Table of contents

Preface 1

Programme Committee 3

Papers presented in the Doctoral Consortium

Integrating Paradigms for Approximate Probabilistic Planning 5
Andrey Kolobov, Mausam and Daniel S. Weld

Integrating a Closed World Planner and an Open World Robot 9
Kartik Talamadupula

Using Interval Automata to Represent Decision Policies with Continuous Variables 13
Alexandre Niveau, Hélène Fargier, Cédric Pralet and Gérard Verfaillie

Learning Policies to Exploit an Extended Domain Model 17
Alan Lindsay, Maria Fox and Derek Long

Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement 21
Hootan Nakhost and Martin Müller

Learning to Combine Admissible Heuristics Under Bounded Time 25
Erez Karpas

Efficient Incremental Search for Moving Target Search 29
Xiaoxun Sun, William Yeoh and Sven Koenig

Revisiting Bounded Suboptimal Heuristic Search 33
Jordan Thayer

A Hybrid Assembly Task Planning System: Where Motion Planning Helps Symbolic Planning Find Good Solutions For Real-World Applications 37
Frederik W. Heger and Sanjiv Singh

Issues in Temporal Planning using Timed Petri Nets 41
Melissa Liew

Market-based Risk Allocation for Multi-agent Systems 45
Masahiro Ono and Brian C. Williams

Hierarchical Skill Learning for High-Level Planning 49
James MacGlashan and Marie desJardins

An Investigation into Using Object Constraints to Synthesize Planning Domain Models 53
M.M.S. Shah, T. L. McCluskey and M.M. West

Abstracts of papers presented in the technical session of ICAPS

Dynamic Controllability of Temporally-flexible Reactive Programs 57
Robert Effinger, Brian Williams, Gerard Kelly and Michael Sheehy

Minimal Sufficient Explanations for Factored Markov Decision Processes 58
Omar Zia Khan, Pascal Poupart and James P. Black

Inference and Decomposition in Planning Using Causal Consistent Chains 59
Nir Lipovetzky and Hector Geffner

Flexible Execution of Plans with Choice 60
Patrick R. Conrad, Julie A. Shah and Brian C. Williams

A Human-Aware Robot Task Planner 61
Marcello Cirillo

An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP 62
Mauro Vallati

Preface

For any field of research, one of the most essential tasks is to attract and to support young researchers. The aim of the Doctoral Consortium series is to provide a forum for students to discuss their work with other students and senior researchers working in the same area.

In ICAPS 2009, the Doctoral Consortium featured a full day workshop on September 19th, where all accepted papers were given a short time slot for oral presentation in small themed clusters of common interest with students and senior researchers. Participants also had further opportunity to present at the poster session of the main conference to receive feedback from other researchers.

This volume contains the 19 papers accepted for presentation at the DC of ICAPS 2009. These papers were selected from a total of 40 submissions. Papers only containing an abstract correspond to papers already accepted for the main technical conference. The remaining papers received two reviews, and preference was given to students who had moderately mature works but still far enough along to have formulated their dissertation topics.

The accepted papers were distributed in five themed clusters around the main area and topics of each paper. Each themed working group was thus formed by four students and, at least, two mentors. Each of the students was assigned a senior researcher to mentor him/her during the group discussions. Students had also the opportunity at the end of the DC session to present and share their roadmap with the students of other clusters.

We thank all the reviewers of the Programme Committee for their work in reviewing the students' papers. Our special thanks to those reviewers who also kindly accepted to participate in the mentoring activity.

Eva Onaindia
Antonio Garrido
Universidad Politécnica de Valencia, Spain

ICAPS 2009 Doctoral Consortium Chairs

Chairs

Eva Onaindía, Universidad Politécnica de Valencia (Spain)
Antonio Garrido, Universidad Politécnica de Valencia (Spain)

Programme Committee

Daniel Borrajo, Universidad Carlos III (Spain)
Adi Botea, NICTA (Australia)
Ronen Brafman, Ben-Gurion University (Israel)
Daniel Bryce, Utah State University (USA)
Andrew Coles, University of Strathclyde (UK)
Yannis Dimopoulos, University of Cyprus (Cyprus)
Minh B. Do, Palo Alto Research Center (USA)
Carmel Domshlak, Israel Institute of Technology (Israel)
Fernando Fernández, Universidad Carlos III (Spain)
Susana Fernández, Universidad Carlos III (Spain)
Hector Geffner, Universitat Pompeu Fabra (Spain)
Patrick Haslum, Australian National University (Australia)
Malte Helmert, University of Freiburg (Germany)
Joerg Hoffmann, SAP Research (Germany)
Ari K. Jónsson, Reykjavík University (Iceland)
Sven Koenig, University of Southern California (USA)
Roman van der Krogt, University College Cork (Ireland)
Ugur Kuter, University of Maryland (USA)
Derek Long, University of Strathclyde (UK)
Tim Lee McCluskey, University of Huddersfield (UK)
Angelo Oddi, ISTC-CNR (Italy)
Federico Pecora, Örebro University (Sweeden)
Kanna Rajan, Monterey Bay Aquarium Research Institute (USA)
Alessandro Saetti, Università degli Studi di Brescia (Italy)
Oscar Sapena, Universidad Politécnica de Valencia (Spain)
Gérard Verfaillie, ONERA, Centre de Toulouse (France)
Petr Vilim, ILOG (Czech Republic)

Integrating Paradigms for Approximate Probabilistic Planning

Andrey Kolobov Mausam Daniel S. Weld
{akolobov, mausam, weld}@cs.washington.edu
Dept of Computer Science and Engineering
University of Washington, Seattle
WA-98195

Abstract

Humans are often able to solve extremely large probabilistic planning problems reasonably well by exploiting problem structure, heuristics, and various approximations. Each of these aspects seems indispensable for achieving high scalability and has been studied in detail by the automated planning community. However, most existing solvers use only a proper subset of them.

In an initial attempt to bridge this gap we introduce RETRASE, a novel MDP solver that derives approximate policies by extracting problem structure and learning its parameters under heuristic guidance. RETRASE uses classical planning to discover basis functions for value-function approximation and applies expected-utility analysis to this compact space. Experiments demonstrate that RETRASE outperforms winners from the past three probabilistic-planning competitions on many hard problems. We outline several extensions of RETRASE and new directions for unifying paradigms in probabilistic planning prompted by RETRASE's success.

1 INTRODUCTION

As humans, we are often able to solve probabilistic planning problems far larger than state-of-the-art planners are capable of. Even though we tend to miss the problems' probabilistic subtleties, our solutions are in many cases reasonably good. We find them by using heuristics and various approximations ("intuition"), and exploiting problem structure (i.e. disregarding weakly relevant information). Each of these aspects has been studied in detail by the automated planning community.

A popular framework for formulating probabilistic planning problems is Markov Decision Processes (MDPs). One of the most popular algorithms for solving MDPs that yields high-quality solutions is RTDP [1], a technique that explores the state space under the guidance of a heuristic. Unfortunately, being based on dynamic programming, it suffers from a critical drawback — it represents the value function extensionally, i.e., as a table, thus requiring memory (and time) exponential in the number of domain features.

Two broad approaches have been proposed to avoid creating a state/value table. One method involves *domain determinization* and uses a classical planner as a subroutine in computing a policy. Such determinization planners, e.g., FFReplan [11], tend to disregard the probabilistic nature of actions and often have trouble with *probabilistically interesting* [7] domains. In other words, their approximation, while computationally efficient, frequently results in poor solution quality.

The other method, *dimensionality reduction*, maps the state space to a parameter space of lower dimension. Typically, the mapping is done by constructing a small set of basis functions, learning weights for them, and combining the weighted basis function values into the values of states. This can be viewed as discovering problem structure while abstracting away unimportant details. Researchers have successfully applied dimensionality reduction to domains after manually defining a domain-specific mapping but automatic basis function discovery in nominal (e.g., "discrete" or "logical") domains, such as those used in the IPPC, remains a challenge.

Thus, each technique has its advantages but also drawbacks that prevents it from dominating others. In fact, all of them seem indispensable for achieving high scalability and acceptable solution quality simultaneously. However, most existing solvers use only a proper subset of them. This paper bridges the gap — proposing a fusion of these ideas that removes the drawbacks of each. Our algorithm RETRASE, which stands for **Re**gressing **Tr**ajectories for **A**pproximate **S**tate **E**valuation, learns a compact value function approximation successful in a range of nominal domains. It discovers problem structure by planning in a determinized version of the domain at hand and thereby automatically obtaining a set of basis functions, learns the weights for these basis functions by heuristically-guided decision-theoretic means, and aggregates them to compute state values. The set of basis functions is normally much smaller than the set of reachable states, thus giving our planner a large reduction in memory requirements as well as in the number of parameters to be learned.

We demonstrate the practicality of our framework by comparing it to the top IPPC-04, 06 and 08 performers and other state-of-the-art planners, on challenging problems from these competitions. RETRASE demonstrates orders of magnitude better scalability than the best optimal planners, and frequently finds significantly better policies than the state-of-the-art approximate solvers. The success of RETRASE

prompts several promising ideas for future work, outlined at the end of the paper.

2 BACKGROUND

MDPs. In this paper, we focus on probabilistic planning problems that are modeled by factored indefinite-horizon MDPs. They are defined as tuples of the form $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ giving the probability of moving from s_i to s_j by executing a, $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ specifying action costs, s_0 is the start state, and $\mathcal{G}$ is a set of (absorbing) goal states. *Indefinite horizon* refers to the fact that the total action cost is accumulated over a finite-length action sequence whose length is unknown.

In factored MDPs, each state is represented as a conjunction of values of the domain variables. Solving an MDP means finding a good (i.e. cost-minimizing) policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that specifies the actions the agent should take to eventually reach the goal. The optimal expected cost of reaching the goal from a state s satisfies the following conditions, called *Bellman equations*:

$$V^*(s) = 0 \text{ if } s \in \mathcal{G}, \text{otherwise} \quad (1)$$

$$V^*(s) = \min_{a \in \mathcal{A}} [\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s')]$$

Given $V^*(s)$, an optimal policy may be computed as follows: $\pi^*(s) = \text{argmin}_{a \in \mathcal{A}} [\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s')]$.

Solution Methods. The above equations suggest a dynamic programming-based way of finding an optimal policy, called *value iteration* (VI), that iteratively updates state values using Bellman equations in a *Bellman backup* and follows the resulting policy until the values converge.

VI has given rise to many improvements. Heuristically-guided methods, e.g. RTDP, explore the state space in a series of *trials* and update the value function over the states in the trial path using Bellman backups. A popular variant, LRTDP, adds a termination condition to RTDP by labeling those states whose values have converged as 'solved' [2].

3 ReTrASE

On a high level, ReTrASE explores the state space in the same manner as RTDP, but, instead of performing Bellman backups on states themselves, backups are performed over *properties* of the visited states. For each property, modified RTDP learns a weight that reflects the quality of the plans enabled by that property. A state's value may then be computed by aggregating the weights of all its properties. Conceptually, there are three kinds of states at runtime: ones that have been deemed dead ends, ones for which some properties are known, and ones not yet assigned to the other two categories. When ReTrASE encounters a state s of the third type, it applies a classical planner (e.g., FF [5]) to a determinized version of the domain starting from s. If no classical plan exists, then every probabilistic policy from s has zero probability of reaching the goal, and s is marked as a dead end. If FF finds a plan, however, ReTrASE regresses the goal conjunction through the plan to generate a logical formula which is a property holding in s. Learning in the property space supports information transfer between similar states (e.g., all states that share a given property) even before some of these states are visited. Our approach is efficient because fast classical planners can quickly derive these properties, and because the number of properties is typically far smaller than the number of reachable states.

Definitions. We define a *trajectory* to be a sequence $t = s, a_1(o_{j_1}), \ldots, a_n(o_{j_n})$ where s is the trajectory's starting state, and each action $a_k(o_{j_k})$ represents the j_k-th outcome of the probabilistic action a_k. We say that t is a *goal trajectory* if s modified by t's action sequence is a goal state. Further, we define a state *property* to be a conjunction of literals[1]. We say that a state s *possesses* property p if p holds in s. With each property p, we associate a unique *basis function* that has value 1 in s iff s possesses p.We say that a property p (and the corresponding basis function b_p) *enables* a set of trajectories T to the goal if the goal can be reached from any state possessing p by following any of the trajectories in T. [2] A *dead-end* is a state with no trajectory to the goal.

Algorithm Intuition. Consider a trajectory $t_g = s, a_1(o_{j_1}), \ldots, a_n(o_{j_n})$ that ends in a goal state. This is an indication that the sequence of *probabilistic* actions $a_1, \ldots, a_n$ is potentially causally important, since their outcomes $o_{j_1}, \ldots, o_{j_n}$ have positive probability. To discover the causal properties $p_1, \ldots, p_n$ that allow the successful execution of $a_1, \ldots, a_n$, we simply regress sequence t from the goal conjunction. We can now claim that action sequence $a_k, \ldots, a_n$ executed starting from *any* state possessing property p_k will lead us to the goal with positive probability, though the magnitude of the probability is yet unknown. Note that t essentially chooses specific outcomes per action and thus the execution of $a_1, \ldots, a_n$ may not always reach the goal. Nevertheless, all properties that enable any positive-probability trajectory to the goal may be important for our purposes because they act as a basis for further planning. In essence, this step can be thought of as unearthing the relevant causal structure necessary for the planning task at hand.

To obtain goal trajectories all we need is to find plans that reach the goal in the *deterministic* version of the domain (by using a classical planner). Every such plan corresponds to a positive-probability trajectory in the original domain.

We can now define a new probabilistic planning problem over a state space consisting of these properties. In practice, the space of properties is much smaller than the original state space, since only the relevant causal structure is retained[3], giving us large reductions in space requirements. Solving this new problem amounts to learning the *weights* for the properties. The weights will be a quantitative measure of each property's importance. There are many imaginable ways to learn them; in this paper, we try one of such methods — a modified version of RTDP. The use of RTDP enables us to leverage the power of heuristic guidance in exploring the state space.

[1] Our algorithm can easily extend to properties specified using general logical formulas.

[2] assuming that the desired outcome is obtained for each action on the trajectory.

[3] We may approximate this further by putting a bound on the

Algorithm 1 ReTrASE

```
Input: probabilistic domain D, problem P = ⟨init. state s_0, goal G⟩, trial length L
declare global map M from basis functions to weights
declare global set DE of dead ends
compute global determinization D_d of D
// Do modified RTDP over the basis functions
for all i = 1 : ∞ do
    declare state s ← s_0
    declare numSteps ← 0
    while numSteps < L do
        declare action a' ← arg min_a {ExpActCost(a, s)}
        ModifiedBellmanBackup(a', s)
        s ←execute action a' in s
        numSteps ← numSteps + 1
    end while
end for

function ExpActCost(action a, state s)
declare array S_o ← successors of s under a
declare array P_o ← probs of successors of s under a
return cost(a) + Σ_i P_o[i] Value(S_o[i])

function Value(state s)
if s ∈ DE then
    return a large penalty // e.g., 1000000
else if some member f' of M holds in s then
    return min_{basis functions f that hold in s} {M[f]}
else
    GetBasisFuncsForS(s)
    return Value(s)
end if

function GetBasisFuncsForS(state s)
declare problem p' ← ⟨init. state s, goal G⟩
declare plan pl ← DeterministicPlanner(D_d, p')
if pl == none then
    insert s into DE
else
    declare basis function f ← goal G
    declare cost ← 0
    for all i = length(pl) through 1 do
        declare action a ← pl[i]
        cost ← cost + cost(a)
        f ← (f ∪ precond(a)) − effect(a)
        insert ⟨f, cost⟩ into M if f isn't in M yet
    end for
end if

function ModifiedBellmanBackup(action a, state s)
for all basis functions f in s that enable a do
    M[f] ← ExpActCost(a, s)
end for
```

The weights reflect the fact that the properties differ in the total expected cost of trajectories they enable, as well as in the total probability of these trajectories. This happens partly because each trajectory considers only one outcome for each of its actions. The sequence of outcomes the given trajectory considers may be quite unlikely. In fact, getting some action outcomes that the trajectory does not consider may prevent the agent from ever getting to the goal. Thus, it may be much "easier" to reach the goal in the presence of some properties than others. Now, given that each state generally has several properties, what is the connection between the state's value and their weights? In general, the relationship is quite complex: under the optimal policy, trajectories enabled by several properties may be possible. Therefore, the exact value of a state is a summation of weights over a subset of the state's properties. However, determining this subset is at least as hard as solving the MDP exactly. Instead, we approximate the state value by the *minimum* weight of all properties that the state possesses. This amounts to saying that the "better" a state's "best" property is, the "better" is the state itself.

Thus, deriving useful state properties and their weights gives us an approximation to the optimal value function. The algorithm's pseudocode is presented in Listing 1.

Theoretical Properties. A natural question about RETRASE is that of convergence. As it turns out, there exist problems on which RETRASE does not converge. We stress, however, that the lack of theoretical guarantees is not indicative of a planner's practical success or failure. Indeed, the experimental results show that RETRASE performs quite outstandingly on many of the planning community's benchmark problems.

4 EXPERIMENTAL RESULTS

Our experiments explore two important aspects of RETRASE – (1) quality of solutions in complex domains and (2) scalability. We ran RETRASE on six probabilistically interesting hard problem sets — Triangle Tire World (TTW) from IPPC-06 and -08, Drive from IPPC-06, Exploding Blocks World (EBW) from IPPC-06 and -08, and Elevators from IPPC-06. The experiments were conducted under the restrictions resembling those of IPPC: for each problem, RETRASE had a maximum of 40 minutes for training and then had 30 attempts to solve each problem. The parameter we measured was success rate (the percentage of 30 trials in which RETRASE managed to solve the given problem) — the factor that decides the winner in IPPC. The runs were performed on a 2.8 GHz Intel Xeon processor with 2GB of RAM.

While analyzing the results, it is important to be aware that our RETRASE implementation is not optimized. Consequently, RETRASE's efficiency is likely even better than indicated by the experiments.

On TTW-06 and -08, RETRASE achieved the perfect 100% success rate across all the problems. For TTW-06, this result is unmatched by the IPPC participants, while on TTW-08 one other planner, HMDPP, achieved the same result. On Elevators, RETRASE's performance was rather average, as it could not solve many of the problems. This is probably due

number of properties we are willing to handle in this step.

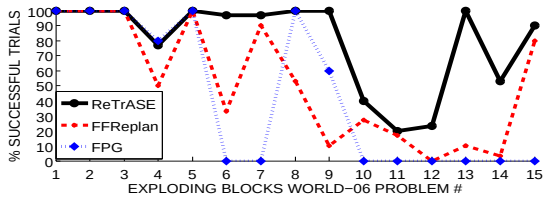


Figure 1: RETRASE dominates on Exploding Blocks World-06.

to the poor set of basis functions it managed to extract for this domain. On Drive, RETRASE's average success rate was at par with the best of IPPC participants (FFReplan and FPG). RETRASE distinguished itself particularly on EBW-08 and EBW-06. On both domains its performance was better than that of all the IPPC competitors. The advantage is especially impressive on EBW-06 because it grows with the complexity of the problem (Figure 1). Space constraints prevent us from presenting the results from other domains in graphical form.

We note also that RETRASE's scalability is far better than that of heuristically guided optimal or suboptimal planners. For instance, LRTDP with FF heuristic ran out of memory on problems 8, 9, and 10 of TTW-08, whereas RETRASE solved them easily.

5 DISCUSSION

RETRASE's potential is clearly indicated by the experimental results, so we would like to extend RETRASE's ideas to other areas of automated planning. One field that would benefit from RETRASE's scalability is POMDP solvers. To be capable of handling POMDPs, RETRASE will need to use a conformant deterministic planner to come up with basis functions. The state of the art in conformant planning will largely determine RETRASE's success as a POMDP solver.

The lack of theoretical guarantees of RETRASE can be perceived by some to be its weakness. To address this concern, we can modify RETRASE's learning mechanism. Space constraints prevent us from specifying the details, but the main idea involves reducing a given MDP to a maximum independent set problem over a "conflict graph" whose vertices are basis functions with associated weights. The independent set problem can then be solved using one of the many theoretically strong approximators for it.

6 RELATED WORK

Besides basis function approximation (discussed in Section 1) other flavors of dimensionality reduction include PCA and algebraic and binary decision diagram (ADD/BDD). In practice algorithms that use ADD/BDD do not scale to large problems. APRICODD [10] is an exception, but it is not clear whether it is competitive with today's top methods. In continuous state spaces, some researchers have applied non-linear techniques like exponential-PCA and NCA for dimensionality reduction [8].

Most basis function based techniques are not applied in nominal domains. A notable exception is FPG [3] but RETRASE outperforms it consistently on several domains.

RETRASE is described in more detail [6]. It is also related in spirit to the probabilistic planners that use determinized domains for probabilistic planning, e.g. FFReplan [11] and FFHop [12].

The idea of using determinization followed by regression has parallels to some research on relational MDPs, e.g., [4; 9].

7 CONCLUSION

Exploiting problem structure, heuristics, and various approximations all seem to be essential components of highly scalable successful probabilistic planners. However, most existing solvers use only a proper subset of them. Our work bridges this gap by introducing RETRASE, an algorithm that combines the power of these approaches. It extracts problem structure in a domain-independent way and learns the parameters in the reduced parameter space under the guidance of a heuristic. We empirically demonstrate that RETRASE outmatches state-of-the-art planners on hard problems from several IPPC competitions and scales drastically better than existing heuristically guided planners. Developing RETRASE's underlying ideas further, we are planning to extend them to POMDPs and add theoretical guarantees on the solution quality by modifying the parameter learning mechanism.

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[2] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS'03*, pages 12–21, 2003.

[3] O. Buffet and D. Aberdeen. The factored policy gradient planner (ipc-06 version). In *Fifth International Planning Competition at ICAPS'06*, 2006.

[4] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection. In *UAI'04*, 2004.

[5] J. Hoffman and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[6] A. Kolobov, Mausam, and D. Weld. ReTrASE: Integrating paradigms for approximate probabilistic planning. In *IJCAI'09*, 2009.

[7] I. Little and S. Thiebaux. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

[8] N. Roy and G. Gordon. Exponential family PCA for belief compression in POMDPs. In *NIPS'02*, pages 1043–1049. MIT Press, 2003.

[9] S. Sanner and C. Boutilier. Practical linear value-approximation techniques for first-order MDPs. In *UAI'06*, 2006.

[10] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *NIPS'00*, 2000.

[11] S. Yoon, A. Fern, and R. Givan. FF-Replan: A baseline for probabilistic planning. In *ICAPS'07*, pages 352–359, 2007.

[12] S. Yoon, A. Fern, S. Kambhampati, and Robert Givan. Probabilistic planning via determinization in hindsight. In *AAAI'08*, 2008.

Integrating a Closed World Planner and an Open World Robot

Kartik Talamadupula
Department of Computer Science
Arizona State University
Tempe, AZ 85287 USA
krt@asu.edu
Advised by: Subbarao Kambhampati
Joint work with: J. Benton, Paul Schermerhorn, Matthias Scheutz

Abstract

In this paper, we present an integrated planning system that actively directs an agent engaged in an urban search and rescue (USAR) scenario. We describe three salient features that comprise the planning component of this system, (1) the ability to plan in a world open with respect to objects, (2) execution monitoring and replanning abilities, and (3) handling soft goals, and detail the interaction of these parts in representing and solving the USAR scenario at hand. We show that though insufficient in an individual capacity, the integration of this trio of features is sufficient to solve the scenario that we present. We test our system with an example problem that involves soft and hard goals, as well as goal deadlines and action costs, and show via an included video that the planner is capable of incorporating sensing actions and execution monitoring in order to produce goal-fulfilling plans that maximize the net benefit accrued.

Introduction

Consider the following problem: a human-robot team is actively engaged in an *urban search and rescue* (USAR) scenario inside a building of interest. The robot is placed inside this building, at the beginning of a long corridor. The human team member has intimate knowledge of the building's layout, but is removed from the scene and can only interact with the robot via on-board wireless audio communication. The corridor in which the robot is located has doors leading off from either side into rooms, a fact known to the robot. However, unknown to the robot (and the human team member) is the possibility that these rooms may contain injured humans (victims). The robot is initially given a hard goal of reaching the end of the corridor by a given deadline based on wall-clock time. As the robot executes a plan to achieve that goal, the team is given the (additional) information regarding victims being in rooms. Also specified with this information is a new soft goal, to report the location of victims.

It is natural to encode this soft goal as a quantified goal, since it is expected that the planner will report the location of as many victims as it can find given its time and cost constraints. The planner must reason about the net benefit of attempting to find a victim, since it is a soft goal and can be ignored if it is not worth the pursuit; it must then direct the robot to sense for the information that it needs in order to determine the presence of a victim in a particular location. This can be modeled as a partial satisfaction planning (PSP) problem, and solved using planners that can handle PSP problems.

Unfortunately, the dynamic nature of the domain coupled with the partial observability of the world precludes complete a priori specification of the domain, and forces the robot and its planner to handle incomplete and evolving domain models (Kambhampati 2007). This fact, coupled with the fallibility of human experts in completely specifying the information that is relevant to the given problem and goals up-front, makes it quite likely that information crucial to achieving some soft goals may be specified at some later stage *during* the planning process. In our USAR scenario, for example, the knowledge that victims are in rooms may be relayed to the planner while it is engaged in planning for the executing robot. In order to handle the specification of such statements in the midst of an active planning process, and enable the use of knowledge thus specified, we need to relax two other crucial invariants that most modern planners rely on. The first is the closed world assumption with respect to the constants (objects) in the problem—the planner can no longer assume that the only objects in the scenario are those that are mentioned in the initial state. The other modification requires that we interleave planning with execution monitoring and, if required, replanning in order to account for the new information.

In this paper, we explore the issues involved in engineering an automated planner to guide a robot towards maximizing net benefit accompanied with goal achievement in such scenarios. We will start by noting that the planning problem that we face here involves partial satisfaction (in that the robot has to weigh the rewards of the soft goals against the cost of achieving them). It also requires replanning ability (in that the robot has to modify its current plan based on new goals that are added). An additional (perhaps more severe) complication is that the planner needs to handle goals involving objects whose existence is not known in the initial state (e.g., the location of the humans to be rescued in our scenario). The system described in this paper was used to guide a Pioneer P3-AT robot as it navigated the scenario

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

presented previously in order to achieve the hard goal of getting to the end of the corridor, while trying to accrue the maximum net benefit possible from the additional soft goal of reporting the location of injured humans (Talamadupula et al. 2009). A video of the robot performing these tasks can be viewed via the following link:
`http://hri.cogs.indiana.edu/videos/USAR.avi`

Planning in an Open World

The planning for the robot is performed by *SapaReplan*, a forward state-space planner based on *SapaPS* (Do and Kambhampati 2004). *SapaReplan* adds the ability to handle updates to the state through the use of a monitor process. We additionally introduce a novel goal construct called an *open world quantified goal* (OWQG) that combines information about the open world and partial satisfaction aspects of the problem.

Goals in an Open World

Our approach seeks to open the world by allowing statements, called open world quantified goals, that label sections of the domain as open with respect to objects. Using these, the domain expert can furnish details about when new objects may be encountered through sensing and include goals that relate directly to the sensed objects. This can be seen as a complementary approach to handling open world environments using *local closed world* (LCW) information produced by sensing actions (Etzioni, Golden, and Weld 1997).

An open world quantified goal (OWQG) is defined as a tuple $\mathcal{Q} = \langle F, \mathcal{S}, \mathcal{P}, \mathcal{C}, \mathcal{G} \rangle$. Here, F and $\mathcal{S}$ are typed variables that are part of the problem Π, where F belongs to the object type that $\mathcal{Q}$ is quantified over, and $\mathcal{S}$ belongs to the object type about which information is to be sensed. $\mathcal{P}$ is a predicate which ensures sensing closure for every pair $\langle f, s \rangle$ such that f is of type F and s is of type $\mathcal{S}$, and both f and s belong to the set of objects in the problem, $\mathcal{O} \in \Pi$; for this reason, we term $\mathcal{P}$ a *closure condition*. $\mathcal{C} = \bigwedge_i c_i$ is a conjunctive first-order formula where each c_i is a statement about the openness of the world with respect to the variable S. For example, `c = (in ?hu - human ?z - zone)` with `S = ?hu - human` means that c will hold for new objects of the type 'human' that are sensed. Finally $\mathcal{G}$ is a quantified goal on $\mathcal{S}$.

Newly discovered objects may enable the achievement of goals, granting the opportunity to pursue reward. For example, detecting a victim in a room will allow the robot to report the location of the victim (where reporting gives reward). Note that reward in our case is for each reported injured person. As such, there exists a quantified goal that must be allowed partial satisfaction. In other words, the universal base (Weld 1994), or total grounding of the quantified goal on the real world, may remain unsatisfied while its component terms may be satisfied. To handle this, we use partial satisfaction planning (PSP) (van den Briel et al. 2004), where the objective is to maximize the difference between the reward given to goals, and the cost of actions. Reward is given for each term $g \in \mathcal{G}$ satisfied, $u(\mathcal{G})$. Additionally each term g is considered soft in that it may be "skipped over" and remain unachieved.

As an example, we present an illustration from our scenario: the robot is directed to "report the location of all victims". This goal can be classified as open world, since it references objects that do not exist yet in the planner's object database $\mathcal{O}$; and it is quantified, since the robot's objective is to report *all* victims that it can find. In our syntax:

```
1 (:open
2   (forall ?z - zone
3     (sense ?hu - human
4      (looked_for ?hu ?z)
5       (and (has_property ?hu injured)
6            (in ?hu ?z))
7   (:goal (reported ?hu injured ?z)
8            [100] - soft))))
```

In the example above, line 2 denotes F, the typed variable that the goal is quantified over; line 3 contains the typed variable $\mathcal{S}$ about which information is to be sensed. Line 4 is the unground predicate $\mathcal{P}$ known as the closure condition (defined earlier). Lines 5 and 6 together describe the formula $\mathcal{C}$ that will hold for all objects of type $\mathcal{S}$ that are sensed. The quantified goal over $\mathcal{S}$ is defined in line 7, and line 8 indicates that it is a soft goal and has an associated reward of 100 units.

Of the components that make up an open world quantified goal $\mathcal{Q}$, $\mathcal{P}$ is required[1] and F and S must be non-empty, while the others may be empty. If $\mathcal{G}$ is empty, i.e., there is no new goal to work on, the OWQG $\mathcal{Q}$ can be seen simply as additional knowledge that might help in reasoning about other goals.

Interleaving Planning and Execution

For most of the sensors on the robot, it is too expensive to sense at every step, so knowing exactly when to engage in perceptual monitoring is of critical importance. Low-level sensing for navigation is handled through action scripts on the robot's end, but for more expensive, high-level operations we use OWQGs. Planning through an open world introduces the possibility of dangerous faults or nonsensical actions. While in some sense, this can be quantified with a risk measure (see (Garland and Lesh 2002), for example), indicating the risk of a plan does nothing to address those risks. A more robust approach in an online scenario involves *planning to sense* in a goal-directed manner. When plans are output to the `ADE` *goal manager*, they include all actions up to and including any action that would result in closure (as specified by the *closure condition*).

Problem Updates and Replanning Regardless of the originating source, the monitoring process receives updates from the robot and correspondingly modifies the planner's representation of the problem. Updates can include new objects, timed events (i.e., an addition or deletion of a fact at a particular time, or a change in a numeric value such as action cost), the addition or modification (on the deadline or

[1] If $\mathcal{P}$ were allowed to be empty, the planner could not gain closure over the information it is sensing for, which will result in it directing the robot to re-sense for information that has already been sensed for.

reward) of a goal, and a time point to plan from. As discussed in (Cushing and Kambhampati 2005), providing for updates to the planning problem allows us to look at unexpected events in the open world as new information rather than faults to be corrected. In our setup, problem updates cause the monitor process to immediately stop the planner (if it is running) and update its internal problem representation. The planner is then signaled to replan on the new problem. In the presence of reward and action cost[2], the replanning process also allows the planner to exploit new opportunities, potentially finding plans that may achieve better net benefit than previous ones.

Implementation

To handle open world quantified goals, we developed a methodology that grounds the problem into the closed world using a process similar to Skolemization. More specifically, we generate *runtime objects* from the sensed variable $\mathcal{S}$ that explicitly represent the potential existence of an object to be sensed. These objects are represented with a suffixed exclamation mark on the object type, followed by a number (e.g., `human!1`). One can look at $\mathcal{S}$ as a Skolem function of F, and runtime objects as Skolem entities that substitute for the function. Runtime objects are then added to the problem and ground into the closure condition $\mathcal{P}$, the conjunctive formula $\mathcal{C}$, and the open world quantified goal $\mathcal{G}$. In other words, runtime objects substitute for the existence of S dependent upon the variable F. The facts generated by following this process over $\mathcal{C}$ are included in the set of facts in the problem through the problem update process. The goals generated by $\mathcal{G}$ are similarly added. This process is repeated for every new object that F may instantiate.

We treat $\mathcal{P}$ as an *optimistic closure condition*, meaning a particular state is considered closed once the ground closure condition is true. On every update the ground closure conditions are checked and if true the facts in the corresponding ground values from $\mathcal{C}$ and $\mathcal{G}$ are removed from the problem.

Consider the scenario at hand and its open world quantified goal. Given a known zone, `zone1`, the process would generate a runtime object `human!1`. Subsequently, the facts `(has_property human!1 injured)` and `(in human!1 zone1)` and the goal `(report human!1 injured zone1)` (with reward 100) would be generated and added to the problem. A closure condition `(looked_for human!1 zone1)` would also be created. When the planning system receives an update including `(looked_for human!1 zone1)`, it will update the problem by deleting the facts `(has_property human!1 zone1)` and `(in human!1 zone1)` and the goal `(report human!1 injured zone1)` at the apprioriate time point. Recall that the planner must only output a plan up to (and including) an action that will make the closure condition true. The idea behind the closure condition is that after it becomes true we can expect closure on certain aspects of the world. Once the condition becomes true, the truth values of the facts in $\mathcal{C}$ are known.

[2]In our scenario, the action costs are determined by a combination of the temporal and resource costs incurred by the robot.

Discussion

It has been a regrettable reality in planning ranks that as the time required to generate a complete plan has decreased, so too has the ability to encode interesting details about the world. This has led to a situation where state-of-the-art planners can only deal with a subset of the features necessary to encode any domain of interest with a real world perspective. To be sure, there do exist planners that can handle the expressivity required to model some or all of the problems that delineate our USAR scenario from existing planning benchmarks (Penberthy and Weld 1992; Golden, Etzioni, and Weld 1994); unfortunately, none of these planners combine all the features necessary to solve our problem in the real world. Planning technologies and systems have been analyzed previously (Smith 2003) in order to move these techniques closer to being able to step up and perform in the real world (albeit other-worldly) domains used at NASA.

We took a similar approach towards our problem—we first considered the assumptions we would need to relax in order for a state-of-the-art planner to be able to reason about and solve our problem. In doing so, we found that these assumptions—about the closed world, the separation of planning and execution stages, and all goals being hard—had a direct correspondence to the features that we wished to model. We also discovered the lack of a planner that combines solutions to these problems in one integrated system.

The first (and in some ways most important) assumption we had to relax was one that most modern day planners have come to take for granted—the assumption that the world is closed with respect to facts, objects and operators. Since our scenario involved the specification of new knowledge concerning the world at any stage during the robot's progress, we had to allow for the possibility that there may be objects in the world that are not specified to the robot initially. We retained the closed nature of the world with respect to operator templates, since it is only reasonable that the robot is made aware of its capabilities initially and does not gain any additional powers on the way.

In order to deal with objects that the robot may either discover or *attempt* to discover to achieve some reward, it is essential that the planner not close its possibilities with regard to objects in the world. To enable this, we introduced the open world quantified goals defined in the previous section and equipped the planner with a mechanism to parse and use the information specified within these goals.

The issue of planning with an open versus closed world representation has been dealt with before, notably in the work of Etzioni et al. (Etzioni, Golden, and Weld 1997) via the specification of *local closed world* (LCW) statements. However, there exists at least one major difference between their work and this attempt. We note that the representation used in that work, of closing a world that is open otherwise via the LCW statements, is complementary to our representation. Since our interest in providing support for open world quantified goals (OWQGs) is to relax our planner's assumption of a world closed with respect to object creation, we are *opening* parts of a completely closed world with the aid of OWQGs.

However, the ability to recognize and represent new objects that appear during execution in the world means nothing if the planner cannot actively use these objects in order to output new plans that improve the quality metric (be it time, cost or net-benefit). To support this requirement, we had to endow our planning system with the capability to interleave planning and execution monitoring, so that changes to the world could be transmitted to the planner in the form of updates and subsequently parsed into the planner's database, as outlined in the previous section. This kind of *online planning* seems to be a case-restrictive yet simple solution to the complex problem of dealing with sensing actions. Specifically, this approach seems to be restricted to problems that contain simple reward models where it is reasonable to take a greedy approach.

It may be argued that XII, the planning system used by Etzioni et al. (Etzioni, Golden, and Weld 1997) handles the twin problems of an open world and interleaving planning and execution monitoring described above. However, these two features alone are not sufficient to model the USAR scenario. As described in earlier sections, we wanted the robot (and the planner that creates plans for it to execute) to look at tasks like reporting the presence of victims in rooms as *opportunities*, rather than as hard goals that *must* be achieved. That is to say, we wanted the robot to do its utmost to satisfy the quantification implied in the open world quantified goal, and to report the location of all victims; however, we did not want the pursuit of this objective to cloud the primary goal, which remained getting to the end of the corridor.

To handle this problem, we needed to consider a third relaxation—one that allowed for some goals to be specified as *soft* goals. We used ideas from the field of Partial Satisfaction Planning (PSP) (Smith 2004) and the planner SapaPS (Benton, Do, and Kambhampati 2009) to include support for reasoning about soft goals and net-benefit, as specified in the previous section. Enabling the usage of soft goals mitigates some of the more difficult problems in fully open worlds—in the USAR scenario, when certain rooms are completely undiscoverable, it is infeasible to expect complete satisfaction of certain quantified goals.

Conclusion

In this paper, we presented a novel approach to reconcile a planner's closed world representation with the open world that a robot has to typically execute it. To enable this approach, we presented the integration of techniques which when combined are sufficient to represent and solve the scenario described. We showed that we could handle information about new objects in the world using open world quantified goals, and that our replanning and execution monitoring system is able to handle the new information specified by these goals in order to produce plans that achieve a higher net benefit. We also detailed that our system could support soft goals, thus ensuring that opportunities retain their "bonus" nature, and do not metamorphise into additional hard goals that may constrain existing hard goals. We implemented all novel techniques and algorithms on a robot, and provided a video to demonstrate the robot's new-found abilities. We rounded off with a discussion of work related to our approach.

Acknowledgements

The author wishes to thank J. Benton, Paul Schermerhorn, Subbarao Kambhampati and Matthias Scheutz for their collaboration in this work. Thanks also go to William Cushing for helpful discussions and the generation of the planner *SapaReplan*.

References

Benton, J.; Do, M.; and Kambhampati, S. 2009. Anytime heuristic search for partial satisfaction planning. *AIJ* 178(5-6).

Cushing, W., and Kambhampati, S. 2005. Replanning: A new perspective. In *Proceedings of ICAPS*.

Do, M., and Kambhampati, S. 2004. Partial satisfaction (over-subscription) planning as heuristic search. *Proceedings of KBCS-04*.

Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *AIJ* 89(1-2):113–148.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *Proceedings of the National Conference on Artificial Intelligence*, 461–467. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Golden, K.; Etzioni, O.; and Weld, D. 1994. XII: Planning with universal quantification and incomplete information. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning, KR*, volume 94. Citeseer.

Kambhampati, S. 2007. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain theories. *Proceedings of AAAI 2007*.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL.

Smith, D. 2003. The Next Challenges for AI Planning. Lecture given at the Planet International Summer School.

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, 393–401.

Talamadupula, K.; Benton, J.; Schermerhorn, P.; Kambhampati, S.; and Scheutz, M. 2009. Integrating a Closed World Planner with an Open World Robot: A Case Study. In *ICAPS 2009 Workshop of Bridging the Gap between Task and Motion Planning (BTAMP)*.

van den Briel, M.; Sanchez, R.; Do, M.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proceedings of the National Conference on Artificial Intelligence*, 562–569. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Weld, D. 1994. An introduction to least commitment planning. *AI magazine* 15(4):27–61.

Using Interval Automata to Represent Decision Policies with Continuous Variables

Alexandre Niveau
ONERA Toulouse, France
2 av. Edouard Belin, 31400 Toulouse
alexandre.niveau@onera.fr

Hélène Fargier
IRIT-CNRS, Université Paul Sabatier,
Toulouse, France
fargier@irit.fr

Cédric Pralet and **Gérard Verfaillie**
ONERA Toulouse, France
2 av. Edouard Belin, 31400 Toulouse
{pralet, verfaillie}@onera.fr

Abstract

Knowledge compilation consists in transforming a problem offline into a form which is tractable online. In this paper, we introduce new structures, based on the notion of interval automaton, adapted to the compilation of decision policies involving both discrete and continuous variables. We show that two of these structures support polytime processing of the two main operations needed to use a policy online: conditioning and model extraction.

Introduction

Controlling an autonomous system requires to be able to make decisions automatically, depending on the current observations and goals. If decision-making is completely performed *online* with the embedded computational capabilities, the *reactivity* of the system can be compromised. On the other hand, if all decisions to be made in every possible situation are computed offline, the limited size of embedded memory may not allow the potentially huge set of different alternatives to be recorded on-board.

A possible way of solving this contradiction is to use *knowledge compilation*, which consists in transforming a problem offline in such a way that its online resolution becomes tractable. For example, given a non-deterministic planning problem, it can be possible to solve it offline and then to express the obtained decision policy in some *target compilation language*. The compiled form must then be both as compact as possible, so that embedded memory constraints are respected, and as tractable as possible, so that some relevant queries can be processed efficiently online.

In our planning application domain, the two main queries we are interested in are *conditioning* and *model extraction*. Conditioning consists in focusing the policy on the observed situation; model extraction then provides a suitable decision for this particular situation.

Efficient target compilation languages were proposed for planning domains involving variables with boolean or enumerated domains (*e.g.* OBDDs (Bryant 1986), finite-state automata (Vempaty 1992), DNNFs (Darwiche and Marquis 2002), etc.). However, many real applications involve continuous variables, such as time or energy. The goal of this paper is to define new target compilation languages for expressing mixed policies, involving both continuous and discrete features.

Interval Automata

Structure and Semantics

Definition 1. (Interval automaton) *An* interval automaton *(IA) is a couple $\phi = \langle X, \Gamma \rangle$, with*

- *X (denoted $\mathrm{Var}(\phi)$) a finite and totally ordered set of real variables, whose domains are representable by the union of a* finite *number of* closed *intervals from $\mathbb{R}$;*
- *Γ a directed acyclic graph with at most one root and at most one leave (the* sink*), whose non-leaf nodes are labelled by a variable of X, and whose edges are labelled by a closed interval from $\mathbb{R}$.*

This definition allows an interval automaton to be empty (no node at all) or to contain only one node (together root and sink) and ensures that every edge belongs to at least one path from the root to the sink. Figure 1 gives an example of interval automaton.

For $x \in X$, $\mathrm{Dom}(x) \subseteq \mathbb{R}$ denotes the *domain* of x, which can either be enumerated ($\mathrm{Dom}(x) = \{1, 3, 56, 4.87\}$) or continuous ($\mathrm{Dom}(x) = [1, 7] \cup [23.4, 28]$). For $Y = \{y_1, \ldots, y_k\} \subseteq X$, such that the y_i are sorted in ascending order, $\mathrm{Dom}(Y)$ denotes $\mathrm{Dom}(y_1) \times \cdots \times \mathrm{Dom}(y_k)$, and $\vec{y}$ denotes an *Y-assignment* of variables from Y, *i.e.* $\vec{y} \in \mathrm{Dom}(Y)$. Last, $\vec{y}(y_i)$ denotes the value assigned to y_i in $\vec{y}$.

Let $\phi = \langle X, \Gamma \rangle$ be an interval automaton, N a node and E an edge in Γ. We can then define the following elements:

- $\mathrm{Root}(\phi)$ the root of Γ and $\mathrm{Sink}(\phi)$ its sink;
- $|\phi|$ the *size* of Γ, *i.e.* its number of edges;
- $\mathrm{Out}(N)$ (resp. $\mathrm{In}(N)$) the set of outgoing (resp. incoming) edges of N;
- $\mathrm{Var}(N)$ the variable labelling N (we decide to denote $\mathrm{Var}(\mathrm{Sink}(\phi)) = \emptyset$);
- $\mathrm{Src}(E)$ the node from which E comes and $\mathrm{Dest}(E)$ the node to which E points;
- $\mathrm{Itv}(E)$ the interval labelling E, and $\mathrm{Var}(E)$ the variable associated with E, defined as $\mathrm{Var}(E) = \mathrm{Var}(\mathrm{Src}(E))$.

An interval automaton can be seen as a compact representation of a boolean function over discrete or continuous variables. This function is the *interpretation function* of the interval automaton:

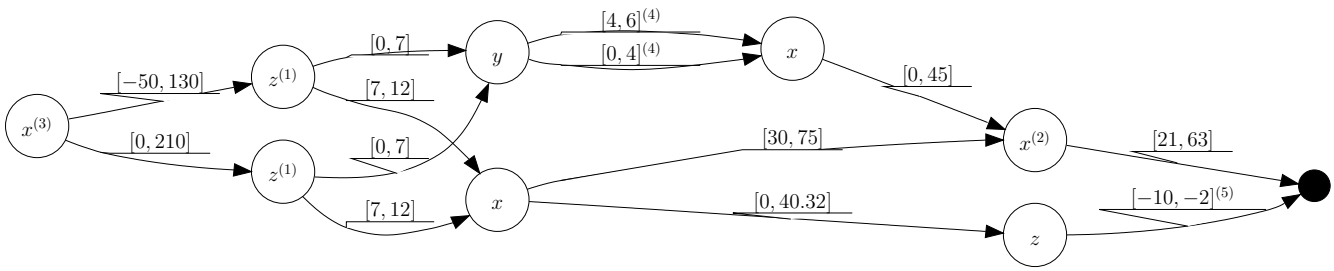


Figure 1: An example of interval automaton. The domains of values of variables are $\mathrm{Dom}(x) = [0, 100]$, $\mathrm{Dom}(y) = [0, 100]$ and $\mathrm{Dom}(z) = \{0, 3, 7, 10\}$. The two nodes marked $^{(1)}$ are isomorphic (Def. 5); the node marked $^{(2)}$ is stammering (Def. 6); node $^{(3)}$ is undecisive (Def. 7); the edges marked $^{(4)}$ are contiguous (Def. 8); and edge $^{(5)}$ is unreachable (Def. 9).

Definition 2. (Semantics of an interval automaton) *An interval automaton ϕ on X (*i.e. $\mathrm{Var}(\phi) = X$*) represents a function from $\mathrm{Dom}(X)$ to $\{\top, \bot\}$. This function, called its* interpretation function $\mathrm{I}(\phi)$*, is defined as follows: for every X-assignment $\vec{x}$, $\mathrm{I}(\phi)(\vec{x}) = \top$ if and only if there exists a path from the root to the sink of ϕ such that for each edge E on this path, $\vec{x}(\mathrm{Var}(E)) \in \mathrm{Itv}(E)$.*

The model set *of ϕ is the subset of $\mathrm{Dom}(X)$ defined by $\mathrm{Mod}(\phi) = \{\vec{x} \in \mathrm{Dom}(X) \mid \mathrm{I}(\phi)(\vec{x}) = \top\}$. Its elements are the* models *or* solutions *of ϕ. ϕ is said to be* equivalent *to another IA ψ (denoted $\phi \equiv \psi$) iff $\mathrm{Mod}(\phi) = \mathrm{Mod}(\psi)$.*

Note that the interpretation function of the empty automaton always returns $\bot$, since an empty IA contains no path from the root to the sink. Conversely, the interpretation function of the one-node automaton always returns $\top$, since in the one-node IA, the only path from the root to the sink contains no edge. We can now introduce useful definitions:

Definition 3. (Consistency, context) *An interval automaton ϕ is said to be* consistent *if and only if $\mathrm{Mod}(\phi) \neq \emptyset$.*

A value $\omega \in \mathbb{R}$ is said to be consistent *for a variable $y \in X$ in an interval automaton ϕ on X if and only if there exists an X-assignment $\vec{x}$ in $\mathrm{Mod}(\phi)$ such that $\vec{x}(y) = \omega$.*

The set of all consistent values for y in ϕ is called the context *of y in ϕ.*

Definition 4. (Conditioning of an interval automaton) *Let ϕ be an interval automaton on X, $Y \subseteq X$ be a set of variables, and $\vec{y}$ be an Y-assignment. An interval automaton ψ is a* conditioning *of ϕ by $\vec{y}$, denoted $\psi \equiv \phi_{|\vec{y}}$, if and only if ψ does not mention any variable from Y, and for every X-assignment $\vec{x}$, $\mathrm{I}(\psi)(\vec{x}) = \mathrm{I}(\phi)(\vec{x}')$, with $\vec{x}'$ the X-assignment such that $\vec{x}'(y) = \vec{x}(y)$ if $y \in Y$, and $\vec{x}'(y) = \vec{x}(y)$ otherwise.*

Reduction

An interval automaton can be reduced in size without changing its semantics by merging some nodes or edges. The reduction operations introduced thereafter are based on the notions of isomorphic, stammering and undecisive nodes, and of contiguous and unreachable edges. Some of these notions are just generalizations of definitions introduced in the context of BDDs (Bryant 1986), while others are specific to interval automata. Each definition is illustrated in Figure 1.

Definition 5. (Isomorphic nodes) *Two nodes N_1, N_2 of an IA ϕ are* isomorphic *if and only if*

- $\mathrm{Var}(N_1) = \mathrm{Var}(N_2)$*;*
- *there exists a bijection σ from $\mathrm{Out}(N_1)$ onto $\mathrm{Out}(N_2)$, such that $\forall E \in \mathrm{Out}(N_1), \mathrm{Itv}(E) = \mathrm{Itv}(\sigma(E))$ and $\mathrm{Dest}(E) = \mathrm{Dest}(\sigma(E))$.*

Definition 6. (Stammering node) *A non-root node N of an IA ϕ is* stammering *iff all parent nodes of N are labelled by $\mathrm{Var}(N)$, and either $|\mathrm{Out}(N)| = 1$ or $|\mathrm{In}(N)| = 1$.*

Definition 7. (Undecisive node) *A node N of an IA ϕ is* undecisive *iff $\forall E \in \mathrm{Out}(N), \mathrm{Dom}(\mathrm{Var}(E)) \subseteq \mathrm{Itv}(E)$.*

Definition 8. (Contiguous edges) *Two edges E_1, E_2 of an IA ϕ are* contiguous *if and only if*

- *they both come from the same node: $\mathrm{Src}(E_1) = \mathrm{Src}(E_2)$;*
- *they both point to the same node: $\mathrm{Dest}(E_1) = \mathrm{Dest}(E_2)$;*
- $\mathrm{Itv}(E_1) \cap \mathrm{Itv}(E_2) \neq \emptyset$.

Definition 9. (Unreachable edge) *An edge E of an IA ϕ is* unreachable *if and only if $\mathrm{Itv}(E) \cap \mathrm{Dom}(\mathrm{Var}(E)) = \emptyset$.*

Definition 10. (Reduced interval automaton) *An interval automaton ϕ is said to be* reduced *if and only if*

- *no node of ϕ is isomorphic to another, stammering or undecisive;*
- *no edge of ϕ is contiguous to another or unreachable.*

In the following, we can consider only reduced IAs since reduction is polynomial.

Lemma 11. (Numbering of an IA) *There exists a polytime algorithm that numbers the nodes of any IA ϕ in such a way that if N_i is a child of N_j, then $i < j$.*

Proposition 12. (Reduction of an IA) *Any IA ϕ can be transformed into an equivalent reduced IA in polytime.*

Proof. Verifying that a node is stammering and eliminating it is linear (suppress it after having created, for each couple (E_I, E_O) of incoming and outgoing edges, an edge labelled by $\mathrm{Itv}(E_I) \cap \mathrm{Itv}(E_O)$, coming from $\mathrm{Src}(E_I)$ and going to $\mathrm{Dest}(E_O)$). Verifying that a node is undecisive and eliminating it is linear (suppress it after having redirected its incoming edges to one of its children N, and created edges labelled by $\mathbb{R}$ going out of N and pointing to each of its other children). Ensuring that a

node has no contiguous or unreachable outgoing edges is $O(|\phi|^2)$, as well as testing whether two nodes are isomorphic.

At any time during process, if an edge has no source or destination, it is suppressed; so are non-leaf nodes without outgoing edges and non-root nodes without incoming edges. All these operations can therefore only decrease $|\phi|$. We have to process them during a traversal from the sink to the root of ϕ. One traversal takes time $O(|\phi|^3)$.

Yet, it is not sufficient, because some operations can modify nodes treated previsouly. As a result, the traversals must be repeated while changes in ϕ occur. But the algorithm remains polynomial: as a traversal can only remove edges and nodes, it will not be repeated more than $|\phi|$ times. □

Operations on Interval Automata

Interval automata can be used to represent decision policies. Indeed, a decision policy can be seen as a function δ over two kinds of variables, the state variables S and the decision ones D. For any S-assignment $\vec{s}$ and any D-assignment $\vec{d}$, $\delta(\vec{s}.\vec{d}) = \top$ if and only if $\vec{d}$ is a suitable decision in state $\vec{s}$.

Suppose that δ is compiled as an IA ϕ. In order to exploit ϕ online, it suffices to condition ϕ by the observed state $\vec{s}$, then extract a model of the resulting IA, whose interpretation function characterizes the set of decisions that suit $\vec{s}$. This process can be applied each time a decision must be made.

Let us now check whether the two required operations, conditioning of an interval automaton and model extraction for an interval automaton, are polytime.

Proposition 13. (Conditioning of an IA) *There exists a polytime algorithm that builds, for any interval automaton ϕ, any set of variables $Y \subseteq \mathrm{Var}(\phi)$ and any Y-assignment $\vec{y}$, an IA that is a conditioning of ϕ by $\vec{y}$.*

Proof. We just have to replace, for each edge E in ϕ such that $\mathrm{Var}(E) \in Y$, $\mathrm{Itv}(E)$ by $\mathbb{R}$ if $\vec{y}(\mathrm{Var}(E)) \in \mathrm{Itv}(E)$, and by $\emptyset$ otherwise. This operation is polynomial. After this step, some nodes may still be labelled by variables in Y; but all the outgoing edges of these nodes are labelled either by $\mathbb{R}$ or by $\emptyset$. They will therefore be suppressed via a (polytime) reduction operation. □

As a result, conditioning is polynomial. As for model extraction, we first need to prove two lemmas.

Lemma 14. (Relation between IAs and BDD) *Any sentence in the BDD language can be expressed in the form of an equivalent IA, in time polynomial in the sentence's size.*

Proof. As boolean variables can be represented by real variables with domain $\{0, 1\}$, a BDD sentence can be transformed into an IA by removing its "false" node, and by recursively removing all edges pointing to no node, and all non-leaf nodes without outgoing edges. The graph obtained then becomes an IA if each "then" edge is replaced by a $[1, 1]$-labelled edge, and each "else" edge is replaced by a $[0, 0]$-labelled edge. □

Lemma 15. (Consistency of an IA) *Unless $\mathsf{P} = \mathsf{NP}$, there exists no polytime algorithm able to check whether any IA ϕ is consistent.*

Proof. If such an algorithm existed, we could check in polynomial time whether a BDD sentence is consistent; yet it is impossible (unless $\mathsf{P} = \mathsf{NP}$), as shown in (Darwiche and Marquis 2002). □

Proposition 16. (Model extraction for an IA) *Unless $\mathsf{P} = \mathsf{NP}$, there exists no polytime algorithm able, for any IA ϕ on X, either to return a X-assignment $\vec{x}$ such that $\vec{x} \in \mathrm{Mod}(\phi)$, or to stop without returning anything if $\mathrm{Mod}(\phi) = \emptyset$.*

Proof. Such an algorithm would invalidate Lemma 15. □

Thereby, interval automata do not satisfy one of the required properties, model extraction. This is why we introduce two particular subsets of interval automata.

Globally Consistent Interval Automata

Let us now introduce a subset of IAs, called *globally consistent IAs* (GCIAs), for which model extraction is easier.

Definition 17. (Global consistency) *An edge E in an interval automaton ϕ is said to be* globally consistent *if and only if every α in $\mathrm{Itv}(E)$ is consistent for $\mathrm{Var}(E)$ in ϕ.*

An interval automaton ϕ is said to be globally consistent *if and only if all its edges are globally consistent.*

Note that except for the particular case of the empty automaton, any GCIA that is reduced is consistent.

We now show that reduction maintains global consistency.

Lemma 18. (Context in a GCIA) *There exists a polytime algorithm that computes, for any GCIA ϕ and any $x \in \mathrm{Var}(\phi)$, the context of x in ϕ.*

Proof. The context of x is simply the union of all labels of edges associated with x, unless ϕ is empty or does not mention x. □

Proposition 19. (Reduction of a GCIA) *Any GCIA ϕ can be turned into an equivalent reduced GCIA in polytime.*

Proof. We only need to slightly modify the algorithm given in the proof of Proposition 12: when processing undecisive nodes, instead of labelling the newly created edges with $\mathbb{R}$, they are labelled with the context of the variable they affect. This ensures that each labelling interval contains only consistent values. □

Both needed operations, conditioning and model extraction, are polytime for GCIAs.

Proposition 20. (Conditioning of a GCIA) *There exists a polytime algorithm that builds, for any GCIA ϕ, any set of variables $Y \subseteq \mathrm{Var}(\phi)$ and any Y-assignment $\vec{y}$, a GCIA that is a conditioning of ϕ by $\vec{y}$.*

Proof. The method proposed for IAs in the proof of Prop. 13 can also be used for GCIAs, since neither changing *all* intervals to $\emptyset$ or $\mathbb{R}$ nor reducing can compromise global consistency . □

The second operation is proved to be polynomial thanks to an important lemma:

Lemma 21. (Consistency of a GCIA) *The consistency of any GCIA ϕ can be checked in time polynomial in $|\phi|$.*

Proof. ϕ is consistent iff it is non-empty when reduced. Indeed, if ϕ is reduced and has at least one edge, there exists a consistent value for some variable in ϕ. The other cases are obvious. □

Proposition 22. (Model extraction for a GCIA) *There exists a polytime algorithm able, for any GCIA ϕ on X, either to return an X-assignment $\vec{x}$ such that $\vec{x} \in \mathrm{Mod}(\phi)$, or to stop without returning anything if $\mathrm{Mod}(\phi) = \emptyset$.*

Proof. First, we can check in polytime whether ϕ is consistent (Lemma 21). If yes, we only have to apply the following algorithm (which is polynomial):

create some X-assignment $\vec{x}$
while $\mathrm{Root}(\phi) \neq \mathrm{Sink}(\phi)$ **do**
  select any value ω in any edge of $\mathrm{Out}(\mathrm{Root}(\phi))$
  $\vec{x}(\mathrm{Var}(\mathrm{Root}(\phi))) := \omega$
  condition ϕ by the $\mathrm{Var}(\mathrm{Root}(\phi))$-assignment (ω)

No variable will be encountered twice, because conditioning removes occurrences of already-assigned variables. When the loop is over, the variables not encountered can be assigned to any value; as it is already the case, there is nothing left to do. □

Tree-Decomposable Interval Automata

The definition of GCIAs is based on the *semantic* property of global consistency; they may be hard to produce. In this section, we propose another tractable subset of IAs that relies on a *syntactic* property called *tree-decomposability*.

Definition 23. (Tree-decomposability) *A non-leaf node N in an interval automaton ϕ is said to be* tree-decomposable *if and only if for all couples (E_1, E_2) of* distinct *edges s.t.*

- $\mathrm{Var}(E_1) = \mathrm{Var}(E_2)$*;*
- *there exists a path from $\mathrm{Dest}(E_1)$ to N not containing E_2 and a path from $\mathrm{Dest}(E_2)$ to N not containing E_1,*

the following statement is true: neither N nor any of its descendants are labelled by $\mathrm{Var}(E_1)$.

An interval automaton ϕ is said to be tree-decomposable *if and only if each non-leaf node in ϕ is tree-decomposable.*

This property is called tree-decomposability, because it implies that for each variable x, if we keep only the x-labelled nodes in ϕ, a tree is obtained.

It is important to note that reducing a tree-decomposable interval automaton (TDIA) does not necessary maintain tree-decomposability, unless we define a strengthened version of isomorphism: denoting A_N the set of all ancestor edges E of a node N s.t. $\mathrm{Var}(E) = \mathrm{Var}(N)$, two nodes N_1 and N_2 are considered as strongly isomorphic iff they are isomorphic and either $A_{N_1} = A_{N_2}$, $A_{N_1} = \emptyset$ or $A_{N_2} = \emptyset$.

Deciding whether an IA is tree-decomposable can easily be done by a traversal from the root to the sink. Moreover, TDIAs satisfy our required queries:

Proposition 24. (Conditioning of a TDIA) *There exists a polytime algorithm that builds, for any TDIA ϕ, any set of variables $Y \subseteq \mathrm{Var}(\phi)$ and any Y-assignment $\vec{y}$, a TDIA that is the conditioning of ϕ by $\vec{y}$.*

Proof. The method proposed for IAs in the proof of Proposition 13 can also be used for TDIAs, if we use a reduction adapted to strong isomorphism. □

Proposition 25. (Model extraction for a TDIA) *There exists a polytime algorithm able, for any TDIA ϕ on X, either to return an X-assignment $\vec{x}$ such that $\vec{x} \in \mathrm{Mod}(\phi)$, or to stop without returning anything if $\mathrm{Mod}(\phi) = \emptyset$.*

Proof. The idea is to associate with each node N in ϕ, $|X|$ empty sets $S^N_{x_1}, \ldots, S^N_{x_{|X|}}$, to assign $S^{\mathrm{Root}(\phi)}_{x_i} := \mathrm{Dom}(x_i)$, and then to traverse ϕ from the root to the sink, doing for each outgoing edge E of each node N the following operations:

if $\mathrm{Itv}(E) \cap S^N_{\mathrm{Var}(E)} = \emptyset$ **then**
  suppress E
else
  assign, for all $x \in X$, $S^{\mathrm{Dest}(E)}_x := S^N_x$
  $S^{\mathrm{Dest}(E)}_{\mathrm{Var}(E)} := \mathrm{Itv}(E) \cap S^N_{\mathrm{Var}(E)}$

At the end, return either any X-assignment $\vec{x}$ s.t. $\forall x \in X$, $\vec{x}(x) \in S^{\mathrm{Sink}(\phi)}_x$, or nothing if $\exists x \in X, S^{\mathrm{Sink}(\phi)}_x = \emptyset$. □

Conclusion

Both structures, GCIA and TDIA, support polytime processing of conditioning and model extraction, and could consequently be used to represent decision policies. Interestingly enough, it should be noted that they both generalize, each in a quite different way, free decision diagrams. Deciding whether they are polynomially equivalent is an open question, left for further research.

GCIAs are interesting theoretically, but may be difficult to build; TDIAs, on the other hand, are probably easier to produce. Indeed, following (Huang and Darwiche 2005), we could use the trace of a search algorithm to build TDIAs. More precisely, we plan to use RealPaver, a CSP solver on real domains (Granvilliers and Benhamou 2006).

In the present work, we considered the simplified problem of exploiting a compiled policy and focused on two operations, conditioning and model extraction. We shall also study the problem of generating the policy, *e.g.* by an approach similar to planning as model checking (Giunchiglia and Traverso 1999). In this perspective, we need to check whether GCIAs and TDIAs support other operations, such as forgetting and conjunction. More generally, a natural continuation of our work is to build a knowledge compilation map for real-valued domains.

If the policy to be exploited is extremely complex, our target languages (and maybe knowledge compilation in general) may not be helpful: the space gain is not important enough to allow the policy to be embedded. For these kind of problems, a solution could be approximate compilation (O'Sullivan and Provan 2006); yet, complete compilation shall nevertheless be used to solve problems of intermediate complexity, such as controller synthesis.

References

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Darwiche, A., and Marquis, P. 2002. A Knowledge Compilation Map. *JAIR* 17:229–264.

Giunchiglia, F., and Traverso, P. 1999. Planning as Model Checking. In *ECP*, 1–20.

Granvilliers, L., and Benhamou, F. 2006. Algorithm 852: RealPaver: an Interval Solver Using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.* 32(1):138–156.

Huang, J., and Darwiche, A. 2005. DPLL with a Trace: From SAT to Knowledge Compilation. In *IJCAI*, 156–162.

O'Sullivan, B., and Provan, G. M. 2006. Approximate Compilation for Embedded Model-Based Reasoning. In *AAAI*.

Vempaty, N. R. 1992. Solving Constraint Satisfaction Problems Using Finite State Automata. In *AAAI*, 453–458.

Learning Policies to Exploit an Extended Domain Model

Alan Lindsay, Maria Fox, Derek Long
University of Strathclyde, Glasgow, Scotland

Abstract

The predicates that are used to encode a planning domain in PDDL often do not include concepts that are important for effectively reasoning about problems in the domain. In particular, the effectiveness of rule-based policies in a domain depend on the concepts that can be expressed in the language used to capture those policies. In previous work we have investigated providing solutions to these limitations in the form of special purpose solvers and in that work we demonstrate how the solvers can lift the language limitations in experiments with hand written policies. In this work we look to integrate these solvers with policy learning technology. We present an architecture that allows certain rule-based policy learners to learn policies that can utilise the features provided by the solvers. We provide evidence to suggest that our architecture could allow control knowledge learners to learn policies that provide control in structurally rich domains.

1 Introduction

The standard language for modelling planning domains, PDDL [4], is an action-centred language. Predicates are used to model relationships between objects in problem instances and actions are modelled in terms of the propositional formulae that must hold before application and the update effects that follow after application. These models are good for capturing state-transition systems cleanly and simply, but are not ideal for expressing certain kinds of meta-structures that are relevant to them. In particular, concepts that influence choices of actions in particular problems are often dependent on richer language than is available in the domain descriptions themselves. This observation has inspired some researchers to explore ways to automatically extend the collection of concepts in a planning domain [8], while others have used hand-crafted concepts to extend the descriptions of specific domains [1, 2].

Control rules are one example of such extended concepts and the success of TLPlan and TALplanner has demonstrated that they offer an effective approach to efficient planning. TLPlan provides a very rich control rule language, extending the expressiveness of the action encodings themselves to include modal formulae that constrain the trajectories plans can follow. This rich language allows abstract concepts and methods of comparison to be constructed and used within the framework of the planning domain description. Unfortunately, encoding control rules demands a thorough understanding of the dynamics of the system being controlled and a challenge for domain-independent planning is to construct similar control knowledge automatically.

There are several approaches for learning control rules [6, 5, 8] and the inclusion of a domain knowledge learning track in the 2008 international planning competition indicates the level of interest in this area within the planning community. A key focus in this field has been to investigate what language features are necessary to capture the concepts required for particular domains. We argue that adding a language feature to the rule language causes the learner to attempt to use that feature in rules for any domain, even if rules can be expressed for a domain without it. As more language features are added to support learning concepts in more domains this could reduce the effectiveness of the learner.

In previous work [7]. we have shown that domain analysis can be used to identify structures in a domain, and that concepts can be provided automatically forming an extended domain model. The key benefit is that the domain model can be extended by concepts that are necessary for reasoning in that domain. We have shown that by expressing the policy's rules using concepts in the extended model, the policy can provide control in the Driverlog and Goldminer domains. Handwritten policies were used in these experiments to demonstrate that control is possible with a limited language. In this work we describe our approach to integrating the machinery to automatically extend the domain model with a policy learner. In this paper we discuss the rule languages for rule based policies, overview the process for extending the domain model, describe our architecture, provide our preliminary results and present our conclusions.

2 The Language of Policy Learners

Many approaches have been used to learn rule-based policies and various languages have been used to capture the rules of these policies. The language features available limit the concepts that can be expressed in the rules. If a concept is not provided in the domain model and cannot be constructed in the language, then the policy rules will not be able to reason with this concept. If the concept is necessary for effective reasoning in a domain, then bad action choices could be mapped to by the policy. In this section we define a policy mapping used in several policy learners and describe

an efficient way of providing the mapping. We then discuss two rule languages and argue that adding language features is not an extendable approach for a domain independent policy learner.

2.1 Policy

A policy is a map from states to actions, and although this can be used in planning to solve a single planning problem, the learning community has focussed on learning partial generalised policies that can be used to solve groups of problems.

Definition 2.1 *A partial generalised policy, π, is a partial map π: States $\times$ Goals $\rightarrow$ Actions.*

We use policy to mean *partial generalised policy* in the rest of this paper.

To use policies effectively in planning their representation must be computable and efficient. A representation that has been used successfully in previous work is to capture a policy as an ordered list of rules [5, 8, 3, 6]. Each rule in the list has two conditions and a corresponding action, in the form: if $\phi \wedge \mathbf{G}\psi$ then do $\mathbf{A}$ where ϕ is a formula that is checked in the current state, while ψ is a formula that is checked against the goal conditions of the problem and $\mathbf{A}$ is an action.

2.2 Language Limitations

In the rule language of the L2Plan [6] and L2Act[5] systems the condition formulae ϕ and ψ are simply conjunctions of literals. The rules are parameterised, so there can be several bindings in a given state. These bindings can be interpreted as an efficient compression of a sequence of policy applications. Concepts captured by this language are constructed directly from the predicates in the domain model and this tightly constrains the policies that can be described. It has been shown that in some cases the language limitations can be lifted by providing hand-crafted support predicates; however, this requires a domain expert to decide which predicates are appropriate.

Martin and Geffner [8] and Fern et al [3] observe that rules of the form 'if an object is in a certain class, then do action a on that object' are often useful in planning. The rule languages developed in the papers are concept languages with extra language features, such as the transitive closure, and are suited to expressing and reasoning with classes of objects. The learning strategy adopted by Fern et al is a variant of approximate policy iteration. In this work they demonstrate that the language can capture concepts necessary for reasoning in several benchmark planning domains.

Individual language features are often suited to building concepts in domains with a particular structure. For example, the transitive closure is particularly useful for expressing concepts in domains with recursive structures. This language feature is not useful however in domains without recursive structures. In the Blocksworld domain, describing the blocks in a stack requires reasoning over the chain of 'on' relationships between blocks and the transitive closure is vital in capturing the necessary concepts, whereas the transitive closure provides no benefit in constructing rules over the original Logistics domain model, as these structures do not exist.

The rules of TLPlan are expressed using a very rich language with many language features. The language features are necessary to express the numerous concepts that support reasoning in the benchmark planning domains. A rule learner using this language would attempt to use all of the features to describe concepts in each domain, however it is likely that the features will not be useful for forming concepts in most of these domains. Adding language features expands the search space for learning policies in all domains, but only provides a useful building block for expressing concepts in a small number of domains. We have investigated learning policies for domains with various structures, however, we have focussed on automatically extending the domain model with concepts that are relevant to the domain.

In this work we focus on the rule language of L2Plan. This allows us to investigate extending the domain model, without also examining the interaction between language features and model extensions.

3 Automatically Extending the Domain Model

The domain model includes the predicates used to describe the state of a planning problem and the actions that can be applied to those states. We have explored automatically extending the domain model in two ways: adding predicates and adding actions. The extra predicates can be used to provide extra state information for reasoning and additional actions can be used to make meta level decisions or to make abstract actions. In this section we describe how special purpose solvers are used to provide these extra features and give an example of how this can support policy control.

3.1 Special Purpose Solvers

A special purpose solver is a hand constructed solution to a specific problem feature. We have developed solvers that provide solutions to problems that are present in several benchmark domains, in particular, graph traversal and resource management. The solvers are automatically invoked when specific features are present in the domain. This ensures that the domain model is extended with actions and predicates that are relevant to the particular domain.

The solvers can introduce predicates to provide extra state information relevant to their particular problem fea-

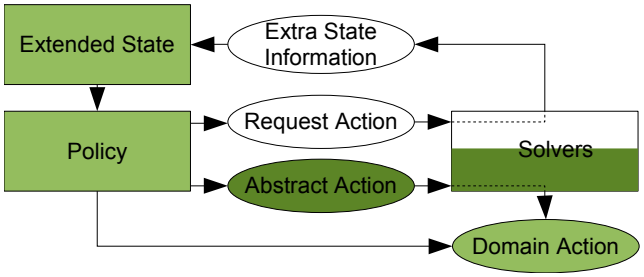


Figure 1: The interaction between the policy and the solvers.

ture. These extra concepts can be used in the rule conditions, providing a view of a complex structure that could not be captured in the rule language. The special purpose solvers can also extend the action set so that the policy can map to actions that make abstract actions or meta level decisions. An abstract action will be translated by the solver into a domain language action, whereas extended actions request further state information and are interpretted as internal actions upon the extended state (Figure 1).

The form of a L2Plan rule is: if $\phi \wedge \mathbf{G}\psi$ then do $\mathbf{A}$. The formula ϕ is evaluated in the current state, however the additional predicates supported by the solvers form an extended state; to make use of this information we evaluate ϕ in the current state and the extended state. An example of a concept that a solver may provide is a resource allocation decision. A predicate is used to represent the allocation and can be included in a rule condition. This will restrict the application of the rule with respect to the allocation.

The actions that the solvers can interpret extend the possible actions fired by a rule. We allow a policy to map to an extended action if there is an associated solver that can interpret that action. The action $\mathbf{A}$ is an action that is either a domain action, or an extended action. An example of an extended action is a request of a solver to make a resource allocation decision. The result of this action will be a predicate that represents the allocation decision being added to the extended state.

3.2 Moving in a Graph Structure

A spatial structure can be encoded as a graph where $(p\, n_1\, n_2)$ holds precisely for connected pairs of nodes, n_1 and n_2. When moving an object through a graph, from n_i to n_j, the destination node n_j is usually identifiable. However, it is unlikely that the nodes along the path between n_i and n_j, have any distinguishing features. For example, a truck might move between these nodes in order to pickup a package at n_j. This location is clearly identifiable by the location of the package, however, there is no information at the nodes in between to indicate that the truck is moving towards the package. To understand the spatial relationship the planner must reason across all of the connecting predicates $(p\, n_i\, n_k), ..., (p\, n_m\, n_j)$. As the number of edges between two nodes depends on the structure defined in a particular problem, this chain of predicates could be of any length.

A solver can be used to provide a solution to this problem. In our experiments our solver extends the action language, allowing the policy to map to actions that move an object through multiple transitions. The solver translates this action as a single transition along the shortest path to the destination indicated in the action. Actions can be used to reason about moving the object to a node that is easily describable, lifting the level of reasoning. The solver then translates this abstracted action back into the domain language.

In our previous work we demonstrated how solvers can allow the policy rules to reason with complex structures, demonstrating this for the dynamic graph in the Goldminer domain and the graph structures and resource management issues in Driverlog.

4 Learning Architecture

The learning architecture that we propose extends the L2Plan learning system to learn rules that are expressed using an extended domain model. The key to this extension is introducing the special purpose solvers into the evaluation function of the policies. In this section we overview the L2Plan learning algorithm and present our approach to support learning rules that harness the functionality of the special purpose solvers.

4.1 L2Plan

L2Plan is a rule-based policy learner that uses a genetic algorithm to evolve generations of policy populations. The initial population is generated randomly using generator functions. Each subsequent population is made by applying operators to policies in the previous population. Policies are selected from the previous population using a selection process that favours better policies, with respect to a given fitness function. The possible fitness functions in L2Plan include evaluating the policy using example states and using the policy to solve problems.

4.2 Initialising the Extended Domain Model

The input to the learner is a domain file and the example states and problems used to evaluate the learned policies. The domain and a problem are analysed using domain analysis. The interesting domain structures are identified and the relevant solvers are selected. As the example states and problems are used many times, we construct the extended state for each example and the initial state of each problem once at the beginning. This is achieved by presenting each of the states to the solvers and giving the solvers the opportunity to respond by extending them with additional concepts.

The generators are used by the learner to provide any predicates or actions that are required for the learned policies. In the L2Plan system these are selected randomly from the available predicates and actions in the domain. We extend the actions and predicates of the domain with the actions and predicates supported by the special purpose solvers. The population of policies in the learner will now have policies represented by rules that use the extended domain model.

4.3 Using the Extended Domain Model

The architecture of L2Plan is particularly suited to this extension. The learning process is separated from the generator functions and the fitness function. The learner uses the fitness of the policies to power its decisions, so the use of the extended domain model has no effect on the learning algorithm. However, the fitness function must be able to evaluate policies captured with concepts in the extended model.

The Fitness Function The fitness function that we use in our experiment evaluates a policy by evaluating it against example states and full problems. In both of these situations the policy is used to map the state and goal into an action. In example state evaluations the action is given a score based on how appropriate it is and in problem evaluations the action forms part of a plan that is evaluated based on quality.

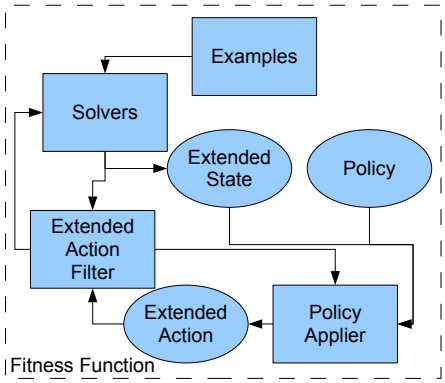


Figure 2: The Fitness Function Architecture.

The policy can map to extended actions, that have effect in the extended state, but do not yield a domain action. Our fitness function architecture, illustrated in Figure 2, uses an abstract action filter that continues to query the policy and solvers until it achieves a domain level action.

The abstract action filter ensures that the action chosen to evaluate is a domain action. The policy is applied to an extended state by the policy applier. If the policy maps to an action in the extended language, provided by the solvers, the abstract action filter uses the appropriate solver to translate the action. If the action is a request for extended state information, such as a resource allocation, the solver will update the extended state and provide nothing to the abstract action filter. In this case the abstract action will apply the policy again, in the updated state. If the action is an abstract action then it is translated by a solver into a domain action. An example of this is an action that moves an object through several transitions in a graph, a solver could translate this as one step in the direction of the actions destination.

The fitness score for a policy is then calculated from the domain actions yielded by the abstract action filter. Any intermediate internal actions are ignored. The use of the solvers is therefore not directly encouraged, however, it is encouraged indirectly in that the features provided by the solvers will assist the policy in mapping to a good domain action.

5 Results

This preliminary experiment was set up to investigate whether a policy learned using our architecture could be used with our solvers to provide effective control. We learned a policy for a transportation domain with multiple trucks and a graph structure using small problems and example states. The domain analysis identified the graph structure in the domain and invoked a graph abstraction solver and the policy uses the concepts provided by this solver. The policy was used to solve 20 transportation problems. These were the Driverlog problems from the 2002 planning competition modified by removing the drivers and paths. Figure 3 shows the quality results of running our policy using breadth first search and the planner FF. The results give some evidence that our policy is providing control in this domain. The problems that the policy failed to solve exhibit properties not present in the example problems. In general the plans that the policy does make are of better quality than FF.

Pfile	1	2	3	4	5	6	7	8	9	10
Policy	1	9	-	11	14	4	9	16	11	14
FF	1	9	9	10	16	5	9	16	15	15
Pfile	11	12	13	14	15	16	17	18	19	20
Policy	17	21	20	21	-	44	58	64	-	-
FF	16	21	20	27	41	46	70	74	103	-

Figure 3: Number of plan steps for transportation problems comparing a learned policy and FF.

6 Conclusion

In this paper we use an approach to automatically extend domain models with concepts that are useful for reasoning in their domains. The approach makes use of special purpose solvers that provide concepts relevant to particular structures. Domain analysis is used to identify if a particular domain contains these structures and in this way the solvers that are relevant to that domain can be chosen to extend the domain model.

We present an architecture that allows the L2Plan rule-based policy learner to learn rules that utilise the concepts in the extended domain. This approach is attractive as the learner does not need to search using many possibly irrelevant language features, but instead is provided with concepts that are useful in the domain. Our preliminary results provide some evidence that policies can be learned that utilise the extended domain model. Our future work is to continue to investigate learning using the presented architecture.

References

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[2] P. Doherty and J. Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22(3):95–102, 2001.

[3] A. Fern, S. Yoon, and R. Givan. Approximate policy iteration with a policy language bias: Solving relational Markov decision processes. *Journal of Artificial Intelligence Research*, 25:75–118, 2006.

[4] M. Fox and D. Long. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *J. Art. Int. Research*, 20:61–124, 2003.

[5] R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[6] J. Levine and D. Humphreys. Learning action strategies for planning domains using genetic programming. In *Proceedings of the 4th European Workshop on Scheduling and Timetabling (EvoSTIM 2003)*, 2003.

[7] A. Lindsay, M. Fox, and D. Long. Lifting the limitations in a rule-based policy language. In *The 22nd International FLAIRS Conference*, 2009.

[8] M. Martin and H. Geffner. Learning generalized policies in planning using concept languages. In *Proceedings of the 7th Internation Conference of Knowledge Representation and Reasoning*, 2000.

Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement

Hootan Nakhost and Martin Müller
Department of Computing Science
University of Alberta
{nakhost,mmueller}@cs.ualberta.ca

Abstract

Current state of the art satisficing planners can produce plans that contain a surprisingly large number of irrelevant actions, redundancies, or unneccessarily complicated or costly action sequences. The recent trend, as in the planning competition IPC-2008, has emphasized plan quality. This paper proposes and evaluates two methods for plan improvement: *Action Elimination* improves an existing plan by repeatedly removing sets of irrelevant actions. *Plan Neighborhood Graph Search* finds an improved plan as the shortest path in a neighborhood graph NG(P) of a given plan P. Both methods are implemented in the ARAS postprocessor. They are empirically evaluated on the output of several leading planners.

The Problem of Plan Quality

In deterministic planning, satisficing planners are able to solve much harder instances than optimal planners. However, they may generate plans that are far from optimal. While previous competitions put much emphasis on the raw speed of planners, the focus of IPC-2008 was on plan quality - achieving the best possible plan within a given time. This paper studies two relatively simple methods for plan improvement, and shows that they are effective for all the planners tested.

Related Work

Weighted A*, or WA* (Pohl 1970), produces plans that are within a constant factor W of optimal.

The LAMA planner, winner of the IPC-2008 competition, uses weighted A* with a large initial value of W to quickly produce an initial plan, then gradually reduces the weight W, while using the best found plan for additional pruning (Richter, Helmert, and Westphal 2008). Other related approaches are Anytime A* (Hansen and Zhou 2007) and Anytime Window A* (Aine, Chakrabarti, and Kumar 2007).

The two similar techniques Joint and LPA* (Ratner and Pohl 1986) improve a given solution by running a shortest path solver between any two selected states which are a fixed distance d apart in the input solution.

In planning by rewriting (Ambite and Knoblock 2001), domain-specific rewriting rules transform a given plan into a better quality plan. Rewriting rules can be given by an expert, or learned from training examples (Upal 1999).

The LPG planner (Gerevini and Serina 2002) uses heuristic local search in plan space. It tries to optimize an objective function that measures the difficulty of resolving the inconsistencies and the cost of a given partial plan. LPG can be used as a post-processor to improve the quality of an input plan. This is done by imposing some random inconsistencies to the input plan. Each time a valid plan with a cost not less than the initial plan is found, new random inconsistencies are injected to the plan.

Several post-processing approaches reduce the makespan of a given totally-ordered plan by converting it into a partially-ordered plan (Do and Kambhampati 2003; Kambhampati 1994; Veloso, Pérez, and Carbonell 1990; Bäckström 1998).

Two Approaches to Plan Improvement

While there is a number of recent algorithms for plan improvement in the weighted A* family, there has been no recent work on such methods for the general case, when the quality of the initial plan is unknown. This is surprising since such plans are arguably most in need of improvement! The two methods presented here take any plan produced by a satisficing planner, and try to improve it. As a pragmatic approach, the methods by themselves produce no global guarantees on the solution quality. Improvement is measured relative to the original plan. However, if a quality bound for the input plan is known, then a corresponding bound on the improved plan can be easily computed.

The plan improvement methods described here are designed to be used as fast postprocessors, independent of any specific planner. For example, the experiments typically use less than 10 seconds for Action Elimination, and less than 4 minutes (out of 30 total under contest conditions) for Plan Neighborhood Graph Search.

Both methods implement different concepts of local optimality: they try to compute the best possible plan within a "neighborhood" of similar plans. They differ in how they define the neighborhood. Action Elimination only removes actions from a given plan. In Plan Neighborhood Graph Search, the neighborhood is a graph consisting of states that are "close" to the states encountered when executing the initial plan.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Action Elimination

Given a plan π, the goal of Action Elimination is to find a shorter plan by removing actions from π.

Definition 1 (Reduction)**.** *Let Π be a planning task, π a solution of Π, and π' a subsequence of π. π' is a* reduction *of π, denoted by $reduct(\pi, \pi')$, iff π' is also a plan for Π.*

Algorithm 1 A greedy algorithm to remove irrelevant actions

Input Initial State s_0, Plan $\pi = (a_1, \ldots, a_n)$, and goal condition G
Output A plan reduction

```
s ← s_0
i ← 1
repeat
   mark a_i {try to remove a_i}
   s' ← s
   for j ← i + 1 to length(π) do
      if a_j is not applicable to s' then
         mark a_j
      else
         s' ← Γ(s', a_j)
      end if
   end for
   if s' satisfies G then
      {commit to the reduction}
      remove marked actions from π
   else
      unmark all actions
      s ← Γ(s, a_i)
   end if
   i ← i + 1
until i > length(π)
return π
```

Action Elimination iteratively improves a given plan $\pi = (a_1, \ldots, a_n)$ by computing a plan reduction in each iteration. The details are given in Algorithm 1. Starting from a_1, the algorithm tries to remove each action in turn. After removing the action, other actions that consequently lose their support - at least one of their preconditions becomes unsatisfied - are removed from the plan. If the reduced sequence remains a solution, the algorithm commits to it as a new plan, otherwise, the removed actions are restored, and the plan remains unchanged. The process terminates when the last action in the remaining plan is tried. Validating a single reduction takes $O(n \times p)$ time, where p is the maximum number of preconditions of an action. The time complexity of the whole algorithm is $O(n^2 \times p)$.

Algorithm 1 does not identify all the removable actions. Consider a planning task with initial state $\{p, q\}$, goal state $\{p, q, r\}$ and the four actions defined in Table 1. In the plan $\pi = (OP_k, OP_p, OP_r, OP_q)$, only OP_r is necessary and OP_k, OP_p and OP_q can be removed. The algorithm first marks OP_k, causing OP_q to lose its support and be marked as well. However, since the remaining sequence (OP_p, OP_r) is not a plan, this step fails and nothing is removed. Next, the algorithm tries OP_p, but again the remaining sequence is not a plan. Therefore, no action is removed from the plan by Algorithm 1.

Name	pre	add	del
OP_k	ϕ	$\{k\}$	$\{p\}$
OP_r	ϕ	$\{r\}$	ϕ
OP_p	ϕ	$\{p\}$	$\{q\}$
OP_q	$\{k\}$	$\{q\}$	ϕ

Table 1: Definition of actions in a planning task example.

Plan Neighborhood Graph Search

Let $\pi = (a_1, \ldots, a_n)$ be a plan and d be a nonnegative integer. Let $S_\pi = \{s_0, \ldots, s_n\}$ be the set of all states visited when executing π. Let S be the set of all states reachable from s_0 by any sequence of actions. Then the *d-neighborhood graph* $NG[\pi, d] = (V, E)$ of π is a directed graph defined as follows: A vertex $s \in S$ is contained in V iff there is a path of length $\leq d$ from some state $s_i \in S_\pi$ to s. An edge (s, s') is in E iff both s and s' are in V and there exists an action a such that $s' = \Gamma(s, a)$.

Algorithm 2 Computation of a Plan Neighborhood Graph

Input $S_\pi = \{s_0, \ldots, s_n\}$, local search method M
Output A graph $G = (V, E)$

```
for i ← 1 to length(π) do
   M.initialize(s_i)
   for j ← 1 to NODES_PER_STEP do
      s ← M.get_next_state()
      V ← V ∪ s
      E ← E ∪ (parent(s), s)
   end for
end for
return G
```

In practice, for efficiency the computation can be restricted by running a local search method M for a fixed number of steps starting from each s_i. Algorithm 2 computes such an approximation. Let G be the resulting graph and d be the furthest distance of any state $s \in V$ from its closest state in S_π. Then G is a subgraph of $NG[\pi, d]$. The size of G is bounded by $NODES_PER_STEP \times n$.

Method M for the experiments reported here is simply the baseline uniform cost search algorithm from the optimal track of IPC-2008 (Helmer, Do, and Refanidis 2008). This algorithm implements an A* search with the following heuristic h: $h = 0$ for goal states and h is equal to the minimum action cost in the problem otherwise. As above, the search is stopped after $NODES_PER_STEP$ expanded nodes.

After building a neighborhood graph, the lowest-cost path from s_0 to s_n is computed.

Experiments

The ARAS postprocessor can be used with any plan. ARAS was used to improve the results for the plans produced by the five top planners in IPC-2008: LAMA, FF_{sa}, FF_{ha}, C3, and SGPLAN6. Results are reported for three configurations of ARAS: E: a single run of Action Elimination using Algorithm 1. S*: repeat running Plan Neighborhood Graph Search until there is no more improvement. (ES)*: run Action Elimination and Plan Neighborhood Graph Search alternatingly until neither can improve the plan.

Tests in Figure 1 were run on a 2.5 GHz machine using 2GB memory. The same scoring function as in the competition was used: cost of best known plan divided by cost of generated plan. For each planner/configuration pair the total score, and the score obtained in each domain are shown. The Openstacks domain is omitted since none of the configurations lead to any improvement in this domain.

While both E and S* are already effective by themselves, the combined (ES)* performs better. Usually, Plan Neighborhood Graph Search alone is more effective than only Action Elimination. E does not find any improvements in a number of domains, but S* is effective in almost all of them. No run took more than 4 minutes. Interestingly, (ES)* is not only superior to S* in terms of plan improvement, but also faster in the worst case.

All three configurations are most effective in problems consisting of several loosely coupled subtasks. In these domains, due to low interaction between different parts of a plan, effective local improvements are possible. For example, all configurations perform very well in the Transport and Elevator domains. Both of these domains are transportation domains in which several objects or persons must be transported to their destinations.

Conclusions and Future Work

Experiments with Action Elimination and Plan Neighborhood Graph Search show substantial improvements of a large variety of plans produced by all tested planners. Action Elimination can quickly remove some useless actions. Plan Neighborhood Graph Search is an anytime algorithm, as long as the underlying local search method M is anytime. Both methods can only find local improvements near the previous plan. There are many promising directions for future work:

- Improve Action Elimination to be more efficient and find more reductions. Action Elimination can be formulated as an optimization problem. Search algorithms such as hill-climbing could be used to find better solutions.
- Experimental results show a rather large variation in the improvability of different plans for the same problem instance. Therefore, using a large number of input plans should increase the chances of the system to find good improved plans. Inputs for ARAS could include many diverse plans as in (Srivastava et al. 2007), or multiple randomized solutions generated by a Monte-Carlo based planner such as ARVAND (Nakhost and Müller 2009).
- Turn ARAS into an anytime system by incrementally controlling the size of the A* searches. Focus on the neighborhood of states with high expected potential for improvements.
- Use ARAS to compute more accurate heuristic values: Most domain-independent heuristic functions use the cost of a solution in an abstract problem as a heuristic value. These solutions could be postprocessed by ARAS to improve their quality.

References

Aine, S.; Chakrabarti, P. P.; and Kumar, R. 2007. AWA* - a window constrained anytime heuristic search algorithm. In *IJCAI*, 2250–2255.

Ambite, J. L., and Knoblock, C. A. 2001. Planning by rewriting. *Journal of Artificial Intelligence Research* 15:207–261.

Bäckström, C. 1998. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research* 9:99–137.

Do, M. B., and Kambhampati, S. 2003. Improving temporal flexibility of position constrained metric temporal plans. In *ICAPS*, 42–51.

Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs with action costs. In *AIPS 2002*, 13–22.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28:267–297.

Helmer, M.; Do, M.; and Refanidis, I. 2008. International Planning Competition-2008, Deterministic Part. Available at `http://ipc.informatik.uni-freiburg.de/`, retrieved April 19, 2009.

Kambhampati, S. 1994. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence* 67(1):29–70.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In *IJCAI*.

Pohl, I. 1970. Heuristic search viewed as pathfinding in a graph. *Artificial Intelligence* 1(3):193–204.

Ratner, D., and Pohl, I. 1986. Joint and LPA*: Combination of approximation and search. In *AAAI*, 173–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

Srivastava, B.; Nguyen, T. A.; Gerevini, A.; Kambhampati, S.; Do, M. B.; and Serina, I. 2007. Domain independent approaches for finding diverse plans. In *IJCAI*, 2016–2022.

Upal, M. A. 1999. Learning rewrite rules to improve plan quality. In *AAAI*, 984.

Veloso, M. M.; Pérez, M. A.; and Carbonell, J. G. 1990. Nonlinear planning with parallel resource allocation. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 207–212. Morgan Kaufmann.

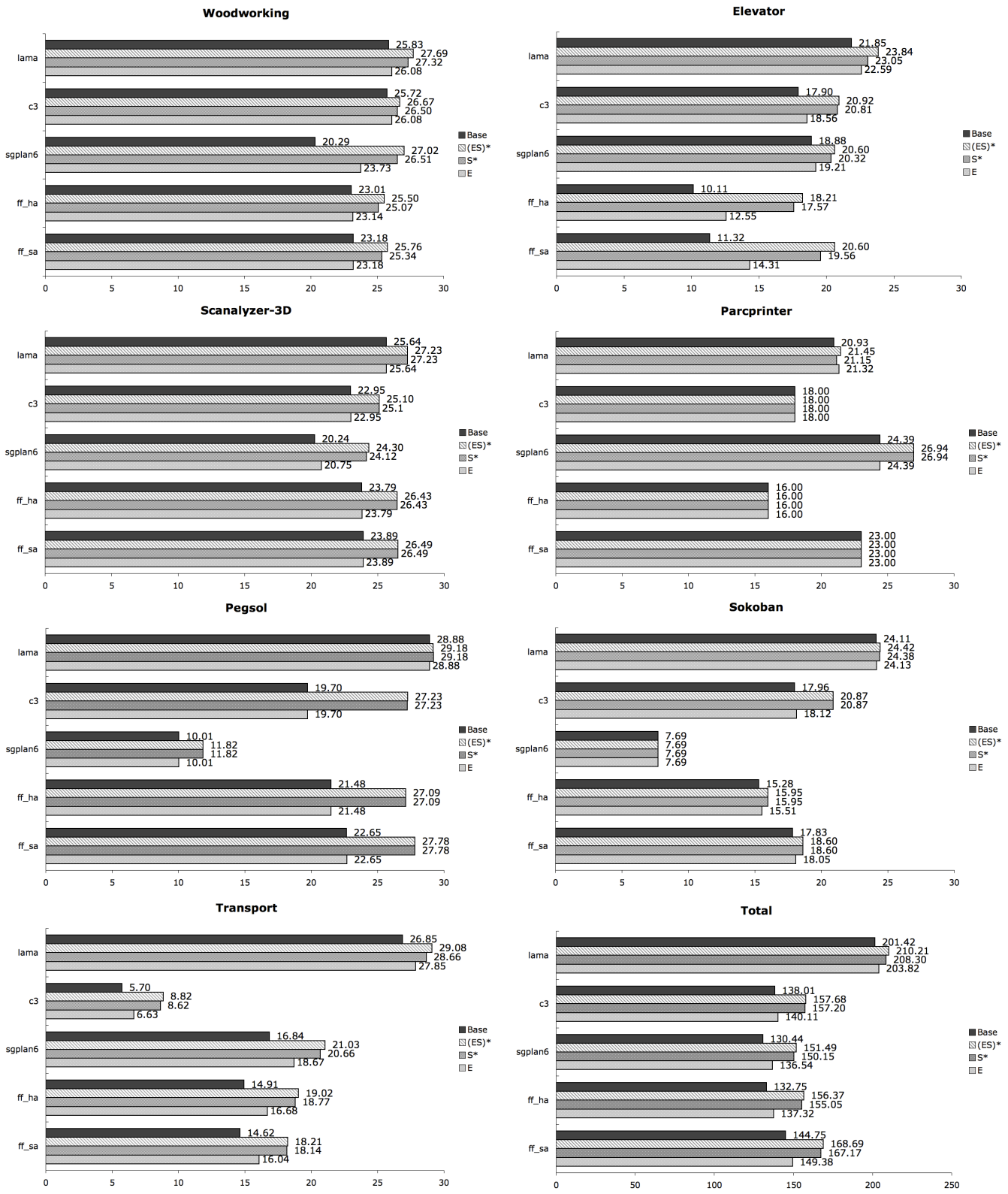


Figure 1: The scores of five base planners (LAMA, FF_{sa}, FF_{ha}, C3, and SGPLAN6) using three different configurations of ARAS (E, S*, (ES)*) in seven IPC-2008 domains. The base results show the score of the planner without using any configuration of ARAS. The bottom right figure shows the total score, including scores gained at Openstacks domain.

Learning to Combine Admissible Heuristics Under Bounded Time

Erez Karpas
Faculty of Industrial Engineering & Management
Technion, Israel

Abstract

Usually, combining admissible heuristics for optimal search (e.g. by using their maximum) requires computing numerous heuristic estimates at each state. In many cases, the cost of computing these heuristic estimates outweighs the benefit in the reduced number of expanded states. If only state expansions are considered, this is a good option. However, if time is of the essence, we can do better than that. We propose a novel method that reduces the cost of combining admissible heuristics for optimal planning, while maintaining its benefits. We first describe a simplified model for deciding which heuristic is best to compute at each state. We then formulate an active learning approach to decide which heuristic to compute at each state, online, during search. The resulting technique, which we call *selective max* is evaluated empirically, and is shown to outperform each of the individual heuristics that were used, as well as their regular maximum, in terms of number of solved instances and average solution time.

Introduction

One of the most prominent approaches to cost-optimal planning (and cost-optimal search in general) is using the A^* search algorithm with an admissible heuristic. Many admissible heuristics for domain-independent planning have been proposed (Bonet and Geffner 2001; Helmert, Haslum, and Hoffmann 2007; Haslum et al. 2007; Coles et al. 2008; Katz and Domshlak 2008; Karpas and Domshlak 2009; Helmert and Domshlak 2009; Katz and Domshlak 2009), varying from cheap to compute and not very informative (Bonet and Geffner 2001) to expensive to compute and very informative (Katz and Domshlak 2008).

Although some heuristics perform better than others on some planning problems, it is difficult to choose a clear-cut "best" heuristic for domain-independent planning in general. Sometimes it is even difficult to choose the best heuristic for a specific planning domain, as different heuristics perform differently on different problem instances in the same domain. Furthermore, it is very hard (if not impossible) to predict how well a given heuristic will perform on a new domain.

One way of producing a more robust planner is by combining several heuristics. The simplest way of doing this is by using their point-wise maximum at each state. Presumably, each heuristic is better (i.e. has a higher estimate) in different regions of the search space, and thus the maximum is more informative than each of the individual heuristics. Another way to combine several heuristics is to use additive heuristics (Katz and Domshlak 2008; Haslum, Bonet, and Geffner 2005; Felner, Korf, and Hanan 2004) or even use both addition and maximum (Coles et al. 2008; Haslum et al. 2007). The problem with both max-based and sum-based approaches is that sometimes the cost of computing numerous heuristic estimates at each state outweighs the benefit in the reduced number of states expanded.

While performing additive combination of heuristics requires computing all of their values, combining heuristics using their maximum does not. Observe that, in every state, the maximum value comes from one of the heuristics. If we had an oracle indicating the most informed heuristic at each state, then computing only that heuristic would result in the same search behavior as max-based combination. However, even if we had such an oracle, it is possible that the extra time spent on computing the more informed heuristic may not be worth the reduction in expanded states. In fact, the results from the last International Planning Competition (IPC-2008) show that it is pretty hard to beat blind search (i.e. A^* with a heuristic which returns 0 for goal states and 1 for non-goal states) – the least informative, and at the same time fastest to compute, heuristic possible.

In this paper we propose a novel method that reduces the cost of combining admissible heuristics, while maintaining its benefits, which we call *selective max*. We first describe a simplified formal model for deciding which heuristic to compute at each state in order to reduce the total search time. We then describe an online active learning scheme that uses a decision rule, derived from our formal model, to decide which heuristic to compute at every state, and can therefore be seen as a *hyper-heuristic* (Burke et al. 2003). An experimental evaluation of our approach using the h_{LA} heuristic (Karpas and Domshlak 2009) and the $h_{\text{LM-CUT}}$ heuristic (Helmert and Domshlak 2009) shows that it solves more planning problems than both of these heuristics individually, as well as their max-based combination, and does it faster on average [1]. Our approach is also shown to exhibit better anytime behavior.

[1] A full version of this paper appears in the workshop on Planning and Learning.

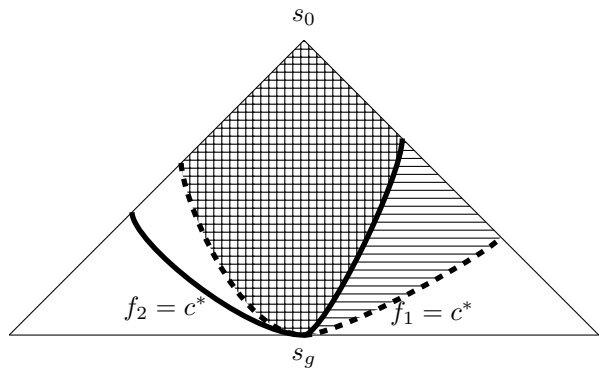


Figure 1: State Space Tree

Notation

We consider planning in the SAS$^+$ formalism (Bäckström and Nebel 1995); a SAS$^+$ description of a planning task can be automatically generated from its PDDL description (Helmert 2009). A SAS$^+$ task is given by a 4-tuple $\Pi = \langle V, A, s_0, G\rangle$. $V = \{v_1, \ldots, v_n\}$ is a set of *state variables*, each associated with a finite domain $dom(v_i)$. Each complete assignment s to V is called a *state*; s_0 is an *initial state*, and the *goal* G is a partial assignment to V. A is a finite set of *actions*, where each action a is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a)\rangle$ of partial assignments to V called *preconditions* and *effects*, respectively.

An action a is applicable in a state $s \in dom(V)$ iff $\mathsf{pre}(a) \subseteq s$, and applying such a changes the value of each state variable v to $\mathsf{eff}(a)[v]$ if $\mathsf{eff}(a)[v]$ is specified. The resulting state is denoted by $s[\![a]\!]$; by $s[\![\langle a_1, \ldots, a_k\rangle]\!]$ we denote the state obtained from sequential application of the (respectively applicable) actions $a_1, \ldots, a_k$ starting at state s. Such an action sequence is a plan if $G \subseteq s[\![\langle a_1, \ldots, a_k\rangle]\!]$.

Model for Heuristic Selection

In order to analyze when computing one admissible heuristic should be preferred to computing another, we need to make certain assumptions about the state space. We assume that the state-space is a tree with a constant branching factor b, uniform cost actions, and that there exists a single goal state (as in the classical results on heuristic-search effort (Pearl 1984)). We also assume that the heuristics are consistent, and that the time t_i required for computing h_i for a state s is independent of s. These assumption do not hold in many planning problems, but we explain how to deal with that in the next section.

A^* expands states according to $f = g + h$, where $g(s)$ is the length of the shortest path to s, and $h(s)$ is the heuristic estimate of the distance from s to the goal. Suppose we have two admissible heuristics: h_1 and h_2. Define $max_h(s) \triangleq \max(h_1(s), h_2(s))$, we then use the notation $f_1 = g + h_1$, $f_2 = g + h_2$, and $\max_f = g + \max_h$.

Assuming the goal state is at depth c^*, let us consider the states satisfying $f_1(s) = c^*$ (the dotted line in Fig. 1) and those satisfying $f_2(s) = c^*$ (the solid line in Fig. 1). The states above the $f_1 = c^*$ and $f_2 = c^*$ contours are those that are surely expanded by A^* with h_1 and h_2, respectively. The states above both these contours (the grid-marked region in Fig. 1) are those that are surely expanded by max_h. These states, which we denote SE (for "surely expanded"), are $SE = \{s \mid \max_f(s) < c^*\}$.

Observe that the optimal decision for any state $s \in SE$ is not to compute any heuristic at all, because all these states are surely expanded anyway. The optimal decision for states that are not surely expanded by max_h is a bit more complicated. Let us consider the states where $f_1(s) < c^*$ and $f_2(s) = c^*$ (that is, in Fig. 1, the states on part of the $f_2 = c^*$ contour that separates between grid-marked and lines-marked areas). Since $g(s)$ is the same in $f_1(s)$ and $f_2(s)$, we know that $h_2(s) > h_1(s)$ (i.e. h_2 is more informative in state s). If we consider only state expansions, h_2 would be the best heuristic to compute at state s. However, h_2 could be more expensive to compute than h_1, so if we consider search time, the choice is not straightforward.

If we compute $h_2(s)$, then s is no longer surely expanded (because $f_2(s) = c^*$, and therefore it may be expanded or not, according to tie-breaking). In contrast, if we compute $h_1(s)$, then s is surely expanded, because $f_1(s) < c^*$. Note that computing h_2 for one of the descendants of s is surely sub-optimal, as we pay the cost of computing h_2, yet only part of the search sub-tree rooted in s is no longer surely expanded. Therefore, our choices can be restricted only to either computing h_2 for s, or computing h_1 for all the states in the sub-tree rooted at s on the $f_1 = c^*$ line (i.e. the leaves of the sub-tree rooted at s and ending at the $f_1 = c^*$ line).

Assume we need to expand l complete levels of the state-space from s to reach the $f_1 = c^*$ line. This means we need to generate on the order of b^l states, and then calculate h_1 for all of the states on the $f_1 = c^*$ line, which would take $b^l t_1$ time. If, instead, we compute h_2, the time required to "explore" the sub-tree rooted in s would be t_2 (assuming favorable tie-breaking). Given that, the optimal decision is thus to calculate h_2 iff $t_2 < b^l t_1$, or if we rewrite this, if $l > \log_b(t_2/t_1)$. As a special case, if both heuristics take the same time to compute, this decision rule boils down to $l > 0$, that is, the optimal choice is simply the more informed (for state s) heuristic.

From Model to Practice

The above simplified formal model for deciding when it is best to evaluate which heuristic makes several assumptions, some of which appear to be problematic to meet in practice. Here we examine these assumptions more closely, suggest pragmatic compromises when possible, and then describe an algorithm, called *selective max*, for speeding up heuristic-search optimal planning according to the principle suggested by our model for exploiting an ensemble of admissible heuristics.

Dealing with Model Assumptions

First, the simplified model assumes that the search space is a tree with a single goal state and uniform cost actions, and that the ensemble heuristics are consistent. Although the first assumption does not hold in most planning problems, and the second assumption is not satisfied by some state of the art heuristics, they do not prevent us from using the decision rule suggested by the model. Considering the exponential growth of the search space "from s" as a function of the heuristic's error, there is some empirical evidence to support the conclusion from the simplified model. For instance, Helmert and Röger (2008) prove for many planning domains that heuristics with a small constant additive error lead A^* to expand an exponential number of states, and the authors also illustrate this phenomenon experimentally. From the perspective of our goals here, from the same empirical results of Helmert and Röger (2008) it is also evident that the number of expanded states increases exponentially as the (still very small and additive) error increases. The latter suggests that the connection between the number of nodes to be expanded in our model and in practice can actually be quite right.

The next step is to somehow estimate the "depth to go" l. For that, we need to make another assumption about the rate at which f_1 grows in the sub-tree rooted at s. Although there are many possibilities here, we will look at two likely options. The first option is that the heuristic estimate h_1 remains constant in the subtree rooted at s (i.e. the additive error increases by 1 for each level below s). In this case, f_1 increases by 1 for each level expanded (because h_1 remains the same, and g increases by 1), and it will take expanding $\Delta_h(s) = h_2(s) - h_1(s)$ levels to reach the $f_1 = c^*$ line. The second option we examine is when the absolute error of h_1 remains constant (i.e. h_1 increases by 1 for each level expanded, so f_1 increases by 2). In this case, we will need to expand $\Delta_h(s)/2$ levels. This can be generalized to the case where the estimate h_1 increases by any constant additive factor c, which results in $\Delta_h(s)/(c+1)$ levels being expanded. In either case, l is linear in $\Delta_h(s)$, and our decision would be to compute h_2 if $\Delta_h(s) > \alpha \log_b(t_2/t_1)$, where α is a hyper-parameter for our algorithm.

Next, the model assumes that the branching factor is constant, and that the times to compute the heuristics are the same across all states. We deal with these two assumptions by estimating and relying upon the average branching factor and average heuristic computation times. These estimates are made on the basis of a random sample of states; the precise estimation procedure is described later on. Third, the model also assumes we know if a state is surely expanded or not. Since we do not, in fact, know this, we treat every state as if it was on the decision border, and thus apply the decision rule at every state.

Learning the Selection Rule Online

Without loss of generality, assume that $t_2 > t_1$. According to the model, the correct decision rule is to use h_2 when $\Delta_h(s) = h_2(s) - h_1(s) > \alpha \log_b(t_2/t_1)$ (since we do not know c^*, we apply this decision rule at every state). In order to do this, we learn a binary classifier that predicts whether, for a given state s, $\Delta_h(s) > \alpha \log_b(t_2/t_1)$. The learning procedure we devise is an *online active learning* procedure. In this procedure, the classifier is presented with a series of examples (states). For every example, the classifier can choose whether to classify the current example using the model learnt so far, or to ask an oracle for the classification of the current example, and then learn from it.

To build a classifier, we first need to collect training examples, which should be representative of the entire search space. One option for collecting the training examples is to use the first k states of the search (where k is the desired number of training examples). However, this method has a bias towards states that are closer to the initial state, and therefore is not likely to represent well the structure of the states in the search space. Hence, we instead collect training examples by sending "probes" from the initial state. Each such "probe" simulates a stochastic hill-climbing search, until a certain depth-limit is reached. All the states generated by such a probe are used as training examples. We send several such probes, until we have collected enough training examples. A more complex sampling method was proposed in (Haslum et al. 2007), but our approach is simpler, and works well in practice.

There are several parameters for this probing procedure. The maximum depth of each probe was set to twice the heuristic estimate of the initial state, that is $2 \max_h(s_0)$. Choosing a successor state to continue the probe from was done according to the inverse of the heuristic value (that is, the probability of choosing a successor s is proportional to $1/\max_h(s)$). The "inverse heuristic" selection biases the sample towards states with a lower heuristic value, which have a higher chance of being expanded by the search.

After the training examples T are collected, they are first used to estimate b, t_1 and t_2 by averaging the respective quantities over T. Once b, t_1 and t_2 are estimated, we can compute the threshold $\alpha \log_b(t_2/t_1)$ for our decision rule. We generate a label for each training example by calculating $\Delta_h(s) = h_2(s) - h_1(s)$, and comparing it to the decision threshold. If $\Delta_h(s) > \alpha \log_b(t_2/t_1)$, we label s with h_2, and otherwise with h_1. If $t_1 > t_2$ we simply switch between the heuristics; our decision is always *whether to evaluate the more expensive heuristic or not* (i.e. the default is to evaluate the cheaper heuristic, unless the classifier says otherwise).

Another requirement for building a classifier is deciding what features characterize states with respect to the decision rule. We decided to start our investigation with the simplest and most accessible features possible, notably the actual state variables of the planning problem, and postpone

evaluate(s)

1. $\langle class, confidence \rangle \leftarrow$ CLASSIFY(s, $model$)
2. **if** ($confidence > \rho$)
   (a) **return** $h_{class}(s)$
3. **else**
   (a) $\Delta_h \leftarrow h_2(s) - h_1(s)$
   (b) $label \leftarrow (\Delta_h > \alpha \log_b(t_2/t_1))$
   (c) Update $model$ with $\langle s, label \rangle$
   (d) **return** $\max(h_1(s), h_2(s))$

Figure 2: The state evaluation algorithm.

feasibility analysis for automatic generation of more complex features for later.

Once we have a classifier (constructed from the initial training set), we start the search from the initial state. For every state we evaluate, we use the classifier to decide which heuristic to compute. If the confidence for one of the possible answers is greater than some confidence threshold ρ, then we follow that decision (i.e. compute the heuristic that was chosen, and use its value). Otherwise, we do not have sufficient information about that state to predict the best heuristic to use, and therefore want to learn from it. This is done by computing the values of both heuristics, generating a label, and learning from this new example[2]. This is described in pseudo-code in Figure 2.

Although many classifiers can be used here, for several reasons we decided to use the Naive Bayes classifier. First of all, both training and classification with Naive Bayes are very fast, which is extremely desirable in the time-bounded setting we are interested in. Another advantage is that it is an incremental classifier, which allows us to learn from a new example very quickly. Finally, when classifying an example, Naive Bayes provides us with a probability distribution over possible classifications, and these probabilities have a natural semantics in terms of the classification confidence.

The Naive Bayes classifier assumes that the features are independent. Although this is not a fully realistic assumption for planning problems, using a SAS$^+$ formulation of the problem instead of STRIPS helps, as instead of many binary variables which are highly dependent upon each other, we have a small number of variables which are less dependent upon each other. The PDDL to SAS$^+$ translator (Helmert 2009) which is used in our implementation, does not generate the "best" possible SAS$^+$ formulation, but gets pretty close to it, by detecting one kind of dependence between propositions in the PDDL representation (mutual exclusion), and creating a single SAS$^+$ state variable that represents all of them. This is most likely part of the reason Naive Bayes works well here.

As a final note, extending selective max to use more than two heuristics is rather straightforward—simply compare the heuristics in a pair-wise manner, and choose the best heuristic by a vote, which can either be a regular vote (i.e. 1 for the winner, 0 for the loser), or weighted according to the classifier's confidence. Although this requires a quadratic number of classifiers, training and classification time (at least with Naive Bayes) appear to be much lower than the heuristic computation time, and thus the overall learning and classification overhead is likely to remain relatively low.

Extending the proposed approach for use with non-uniform action costs remains as future work. Altough the decision rule proposed here can be used "as is" with non-uniform action costs, it is likely that it should be adjusted to account for the different action costs.

[2]We do not change the estimate for b, t_1 and t_2, so the decision threshold remains fixed.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Comp. Intell.* 11(4):625–655.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1–2):5–33.

Burke, E.; Kendall, G.; Newall, J.; Hart, E.; Ross, P.; and Schulenburg, S. 2003. Hyper-heuristics: an emerging direction in modern search technology. In Glover, F., and Kochenberger, G., eds., *Handbook of metaheuristics*. Kluwer Academic Publishers. chapter 16, 457–474.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. Additive-disjunctive heuristics for optimal planning. In *ICAPS*, 44–51.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *JAIR* 22:279–318.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, 1007–1012.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*. In press.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In Fox, D., and Gomes, C. P., eds., *AAAI*, 944–949. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ*. in press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, 174–181.

Katz, M., and Domshlak, C. 2009. Structural-pattern databases. In *ICAPS*. In press.

Pearl, J. 1984. *Heuristics — Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Efficient Incremental Search for Moving Target Search*

Xiaoxun Sun William Yeoh Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{xiaoxuns, wyeoh, skoenig}@usc.edu

Abstract

Incremental search algorithms reuse information from previous searches to speed up the current search and are thus often able to find shortest paths for series of similar search problems faster than by solving each search problem independently from scratch. However, they do poorly on moving target search problems, where both the start and goal cells change over time. In this paper, we thus develop Fringe-Retrieving A* (FRA*), an incremental version of A* that repeatedly finds shortest paths for moving target search in known gridworlds. We demonstrate experimentally that it runs up to one order of magnitude faster than many state-of-the-art incremental search algorithms applied to moving target search.

1 Introduction

Moving target search is the problem where a hunter has to catch a moving target [Ishida and Korf, 1991; Moldenhauer and Sturtevant, 2009]. Motivated by video games, we perform moving target search in known gridworlds with blocked and unblocked cells, where the hunter always knows its current cell and the current cell of the target. The moving target search problem is solved once the hunter reaches the current cell of the target. The hunter needs to determine quickly how to move. A*-based incremental search algorithms reuse information from previous searches to speed up the current search and are thus often able to find the shortest path faster than by solving each search problem independently from scratch [Koenig *et al.*, 2004]. Incremental search algorithms generally fall into two classes:

- The first class uses information from the previous searches to update the h-values of the current search so that they become more informed and focus the current search better. Examples include MT-Adaptive A* [Koenig *et al.*, 2007] and its generalization Generalized MT-Adaptive A* [Sun *et al.*, 2008].
- The second class transforms the previous search tree to the current search tree so that the current search does not need to start from scratch. Examples include Differential A* [Trovato and Dorst, 2002], D* [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2005].

All listed incremental search algorithms from the second class are efficient only if the start cell remains unchanged from search to search. These incremental search algorithms are thus not efficient for moving target search. In this paper, we develop Fringe-Retrieving A* (FRA*), an incremental version of A* from the second class that repeatedly finds shortest paths for moving target search in known gridworlds. We demonstrate experimentally that it runs up to one order of magnitude faster than a variety of state-of-the-art incremental search algorithms applied to moving target search.

2 Search Problem and Notation

We perform moving target search in known gridworlds. We use the following notation: S denotes the finite set of unblocked cells, $s_{start} \in S$ denotes the current cell of the hunter and the start cell of the search, and $s_{goal} \in S$ denotes the current cell of the target and the goal cell of the search. $N(s) \subseteq S$ denotes the set of neighbor cells of cell $s \in S$, namely the at most four adjacent unblocked cells for four-neighbor gridworlds and the at most eight adjacent unblocked cells for eight-neighbor gridworlds. $c(s, s') > 0$ denotes the distance from cell $s \in S$ to cell $s' \in N(s)$. We use one if they are horizontal or vertical of each other and $\sqrt{2}$ if they are diagonal of each other. $d(s, s')$ denotes the distance from cell $s \in S$ to cell $s' \in S$. The h-value $h(s, s')$ needs to be consistent. We use the Manhattan distances for four-neighbor gridworlds and the octile distances for eight-neighbor gridworlds [Bulitko and Lee, 2006]. The objective of each search is to find a shortest path from the start cell to the goal cell.

3 A*

A* is the basis of all incremental search algorithms discussed in this paper. We use the following properties of A*:

- **Property 1**: Every cell s in the CLOSED list satisfies the following conditions: (a) If s is not the start cell, then the parent of s is in the CLOSED list with $g(s) = g(parent(s)) + c(parent(s), s)$. (b) The g-value

*This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number W911NF-08-1-0468 and by NSF under contract 0413196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

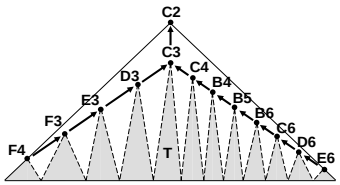


Figure 2: Principle of FRA*.

of s satisfies $g(s) = g(s_{start}) + d(s_{start}, s)$. Property 1 implies that one can identify a shortest path from the start cell to s in reverse by following the parents from s to the start cell. Property 1 also implies that the CLOSED list forms a contiguous area in known gridworlds.

- **Property 2**: The OPEN list contains the following cells: (a) If the CLOSED list is empty, then the OPEN list contains only the start cell. Otherwise, the OPEN list contains exactly all cells that are not in the CLOSED list but have at least one neighbor cell in the CLOSED list. Property 2 (a) implies that the OPEN list contains exactly all cells that on the outer perimeter of the CLOSED list. Every cell s in the OPEN list satisfies the following conditions: (b) The parent of s is the cell s' in the CLOSED list that minimizes $g(s') + c(s', s)$. (c) The g-value of s satisfies $g(s) = g(parent(s)) + c(parent(s), s)$.

4 Fringe-Retrieving A*

Fringe-Retrieving A* (FRA*) is an incremental version of A* that repeatedly transforms the previous search tree to the current search tree (see the pseudo code in [Sun *et al.*, 2009]). Assume that FRA* finds a shortest path from the start cell, which is the current cell of the hunter, to the goal cell, which is the current cell of the target. Properties 1 and 2 hold afterwards since FRA* performs A* searches. Then, the hunter moves along the shortest path. When the target moves off the shortest path, FRA* changes the OPEN and CLOSED lists as well as the parents of the cells in them to guarantee that Properties 1 and 2 hold again and then starts an A* search with these OPEN and CLOSED lists to find a shortest path from the new start cell to the new goal cell. We use the four-neighbor gridworld in Figure 1 to illustrate the steps of FRA*. The current cell C2 of the hunter is labeled S, and the current cell E6 of the target is labeled G. Figure 1(a) shows the very first search of FRA*, which is an A* search from C2 to E6. Cells in the OPEN and CLOSED list are labeled with their g-values in their lower right corners. The outgoing arrows point to their parents. Cells in the OPEN list are labeled OPEN. The shortest path is C2, C3, C4, B4, B5, B6, C6, D6 and E6. The hunter then moves along the shortest path to C3, and the target moves off the shortest path to F6. FRA* now finds a shortest path from C3 to F6 using the following steps.

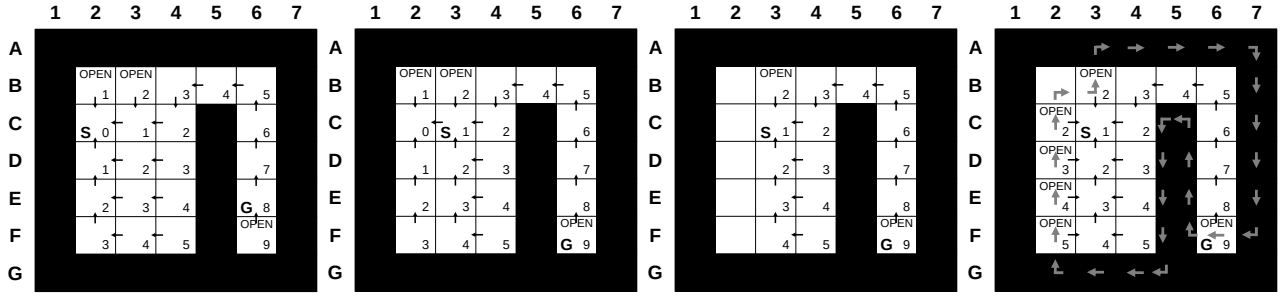
4.1 Step 1: Starting A* Immediately

If the new start cell is the same as the previous start cell and the new goal cell is in the previous CLOSED list, then one can identify a shortest path from the new start cell to the new goal cell in reverse by following the existing parents from the new goal cell to the new start cell according to Property 1. If the new start cell is the same as the previous start cell but the new goal cell is not in the previous CLOSED list, then FRA* starts an A* search with the previous OPEN and CLOSED lists and does not need to execute the following steps. In our example, these conditions are not satisfied and FRA* thus executes the following steps.

4.2 Step 2: Changing Parents

The new CLOSED list needs to satisfy Property 1 with respect to the new start cell. The previous CLOSED list satisfies Property 1 with respect to the previous start cell since FRA* performs A* searches. If the new CLOSED list contains exactly all cells of the subtree of the search tree rooted at the new start cell that are in the previous CLOSED list, then Property 1 holds with respect to the new start cell. Property 1(a) holds trivially, and Property 1(b) holds for the following reason: Consider the shortest path from the previous start cell s'_{start} to any cell s in the new CLOSED list that results from following the parents from s to s'_{start} in reverse. Since the new start cell s_{start} is on this shortest path, it holds that $d(s'_{start}, s) = d(s'_{start}, s_{start}) + d(s_{start}, s)$. Since cell s satisfies Property 1(b) with respect to the old start cell, it holds that $g(s) = g(s'_{start}) + d(s'_{start}, s)$. Since cell s_{start} satisfies Property 1(b) with respect to the old start cell, it holds that $g(s_{start}) = g(s'_{start}) + d(s'_{start}, s_{start})$. Thus, cell s also satisfies Property 1(b) with respect to the new start cell since $g(s) = g(s'_{start}) + d(s'_{start}, s) = g(s_{start}) - d(s'_{start}, s_{start}) + d(s'_{start}, s) = g(s_{start}) + d(s_{start}, s)$. However, the new CLOSED list should be as large a subset of the previous CLOSED list as possible. Cells from the previous CLOSED list often satisfy Property 1(b) with respect to the new start cell but not Property 1(a) since there are typically many alternative shortest paths from the previous start cell to a cell s in the previous CLOSED list. If the new start cell is on the shortest path from the previous start cell to s that results from following the parents from s to the previous start cell, then s is in the subtree of the search tree rooted at the new start cell and thus in the new CLOSED list. If the new start cell is not on the shortest path, then s is not in the subtree of the search tree rooted at the new start cell. FRA* finds such cells and changes their parents to make them part of the search tree rooted at the new start cell so that they can be part of the new CLOSED list. First, FRA* makes the new start cell its current cell, faces the parent of its current cell and performs checks in the counter-clockwise direction. It turns counter-clockwise to face the next neighbor cell s' of its current cell s that is in the previous CLOSED list and checks whether it holds that $g(s') = g(s) + c(s, s')$. If the check is successful, then FRA* sets the parent of s' to s, which is possible because this change does not affect its g-value. Due to this change, all cells in the subtree rooted at s' now belong to the subtree rooted at the new start cell and their g-values remain unaffected. FRA* then makes s' its current cell, faces the new parent of its current cell s and repeats the process of turning and checking the neighbor cell that it faces. If the check is unsuccessful, FRA* repeats the process of turning and checking the neighbor cell that it faces.

If, during the process of turning and checking the neighbor cell that it faces, FSA* faces the parent of its current cell again, then it makes the new start cell its current cell again,



(a) A* Search (b) Steps 1 and 2 (c) Step 3 (d) Steps 4 and 5

Figure 1: Example Trace of FRA* on a Four-Neighbor Gridworld.

faces the parent of its current cell and now performs similar checks in the clockwise direction. The resulting search tree is one that an A* search could have generated if it had broken ties among cells with the same sum of g-value and h-value appropriately. Step 2 traverses at most the inner perimeter of the new CLOSED list, which can be expected to contain fewer cells than the CLOSED list itself.

In our example, the cells in the subtree of the search tree rooted at the new start cell C3 before Step 2 that are in the previous CLOSED list are B4, B5, B6, C3, C4, C6, D6 and E6. However, D3 could have C3 as parent rather than D2 since this change does not affect its g-value. Then, all cells in the subtree of the search tree rooted at D3 belong to this subtree and their g-values remain unaffected. FRA* starts at the new start cell C3 and performs checks in the counter-clockwise direction. First, FRA* is at C3 facing its parent C2 and checks D3 successfully. It then sets the parent of D3 to C3. Second, FRA* is at D3 facing its parent C3, checks D2 unsuccessfully and then checks E3 successfully. It then sets the parent of E3 to D3. Third, FRA* is at E3 facing its parent D3, checks E2 unsuccessfully and then checks F3 successfully. It then sets the parent of F3 to E3. Fourth, FRA* is at F3 facing its parent E3, checks F2 unsuccessfully and then checks F4 successfully. It then sets the parent of F4 to F3 (which does not change its parent). Fifth, FRA* is at F4 facing its parent F3 and checks E4 unsuccessfully. FRA* then starts again at the new start cell C3 and performs similar checks in the clockwise direction until it reaches E6, see Figure 1(b). The cells in the subtree of the search tree rooted at the new start cell C3 after Step 2 that are in the previous CLOSED list are B4, B5, B6, C3, C4, C6, D3, D4, D6, E3, E4, E6, F3 and F4. The big triangle rooted in C2 in the conceptual Figure 2 represents all cells in the search tree that are in the previous CLOSED list, the triangle T rooted in C3 in the big triangle represents all cells in the subtree of the search tree rooted at the new start cell before Step 2 that are in the previous CLOSED list, and the area in the big triangle below and including the cells traversed by FRA* in the counter-clockwise direction (C3, D3, E3, F3 and F4) and the clockwise direction (C3, C4, B4, B5, B6, C6, D6, and E6) represents all cells in the subtree of the search tree rooted at the new start cell after Step 2 that are in the previous CLOSED list.

4.3 Step 3: Deleting Cells

FRA* now generates the new CLOSED list, which contains exactly all cells of the subtree of the search tree rooted at the new start cell that are in the previous CLOSED list. FRA* uses breadth-first search to traverse the search tree except for the subtree rooted at the new start cell, starting with the previous start cell, and delete the visited cells that are in the previous CLOSED list from the previous CLOSED list to form the new CLOSED list. FRA* remembers the parent of the new start cell (that we call the anchor cell). It then sets the parent of the new start cell to NULL to disconnect this subtree from the search tree and uses breadth-first search to traverse the resulting search tree, starting with the previous start cell, and deletes the visited cells that are in the previous CLOSED list from the previous CLOSED list to form the new CLOSED list. It also deletes the visited cells that are not in the previous CLOSED list from the previous OPEN list to form the new OPEN list. Afterwards, the new CLOSED list is complete. The parents and g-values of all cells in the new CLOSED list satisfy Property 1. In our example, FRA* sets the parent of C2 to NULL, it then deletes C2, D2, E2 and F2 from the previous CLOSED list and deletes B2 from the previous OPEN list, see Figure 1(c).

4.4 Step 4: Terminating Early

If the new goal cell is in the new CLOSED list then one can identify a shortest path from the new start cell to the new goal cell in reverse by following the parents from the new goal cell to the new start cell according to Property 1. In our example, this condition is not satisfied and FRA* thus executes the following steps.

4.5 Step 5: Inserting Cells

The new CLOSED list forms a contiguous area according to Property 1. FRA* thus completes the new OPEN list by circumnavigating the part of the outer perimeter of the new CLOSED list that contains the anchor cell, starting with the anchor cell, and inserting every visited unblocked cell that is not yet in the new OPEN list into the new OPEN list. Afterwards, the new OPEN list is complete. However, the parents and g-values of some cells in the new OPEN list might not satisfy Property 2. FRA* therefore sets the parent of every cell s in the new OPEN list to the cell s' in the new CLOSED list that minimizes $g(s') + c(s', s)$ and then the g-value of s to $g(\mathit{parent}(s)) + c(\mathit{parent}(s), s)$. The parents and g-values of all cells in the new OPEN list now satisfy Property 2. Step 5, different from [Edelkamp, 1998], traverses at most the outer perimeter of the new CLOSED list, which can be expected to

	Four-Neighbor Random Gridworld							Eight-Neighbor Random Gridworld						
	searches per test case	moves per test case	cells expanded per search		cells removed from CLOSED per search		total runtime per search	searches per test case	moves per test case	cells expanded per search		cells removed from CLOSED per search		total runtime per search
A*	387	688	14013	(31.2)			7250	335	475	8418	(19.3)			6098
MT-Adaptive A*	387	688	12351	(28.2)			6002	335	475	8001	(21.9)			6078
Differential A*	387	688	14013	(40.1)			9533	335	475	8418	(19.3)			7355
FRA*	391	688	515	(13.0)	362	(11.6)	397	336	477	325	(3.8)	256	(7.5)	438

Table 1: Experimental Results in Random Gridworlds.

contain fewer cells than the CLOSED list itself. In our example, FRA* circumnavigates the part of the outer perimeter of the new CLOSED list that contains C2, starting with C2, and inserts C2, D2, E2 and F2 into the previous OPEN list and then corrects all of their parents and g-values. For example, the parent of D2 was C2 and its g-value was one. FRA* sets the parent of D2 to D3 and its g-value to three, see Figure 1(d).

4.6 Step 6: Starting A*

FRA* starts an A* search with the new OPEN and CLOSED lists. The g-value of the new start cell is not necessarily zero at this point in time since FRA* has not changed it since its previous search.

5 Experimental Evaluation

We compare FRA* experimentally against A*, MT-Adaptive A*, Differential A* and D* Lite. We use comparable implementations for all search algorithms. For example, all of them use binary heaps as priority queues. We use four-neighbor and eight-neighbor random gridworlds in which 25 percent of randomly chosen cells were blocked. We average over 1000 gridworlds of size 1000×1000 whose start and goal cells were randomly chosen. The target always follows a shortest path from its current cell to a randomly selected unblocked cell and repeats the process once it reaches that cell. The target skips every tenth move, which enables the hunter to catch it. Tables 1 reports two measures for the difficulty of the moving target search problems (that vary slightly among moving target search algorithms due to different tie breaking among several shortest paths), namely the average number of searches and the average number of moves of the hunter until it catches the target. The tables report two measures for the efficiency of the moving target search algorithms, namely the average number of expanded cells per search and the average runtime per search in microseconds on a Pentium D 3.0 Ghz PC with 2 GByte of RAM. The tables also report the average number of cells per search that FRA* removes from the CLOSED list. Finally, the tables report the standard deviation of the mean for the number of expanded cells per search and the number of cells removed from the CLOSED list per search (in parentheses) to demonstrate the statistical significance of our results. D* Lite is not listed in the table because it exceeds our runtime limit of 10 hours for the 1000 gridworlds that we average over and is thus not competitive.

In all cases, the sum of the average number of cells that FRA* expands and the average number of cells that it removes from the CLOSED list is an order of magnitude smaller than the average number of cells that the other search algorithms expand. Accordingly, the average runtime of FRA* is an order of magnitude smaller than the average runtime of the other search algorithms.

6 Conclusions

In this paper, we developed Fringe-Retrieving A* (FRA*), an incremental search algorithm that repeatedly finds shortest paths for moving target search in known gridworlds. We demonstrated experimentally that it runs up to one order of magnitude faster than a variety of state-of-the-art incremental search algorithms applied to moving target search in known gridworlds. FRA* is thus the first incremental search algorithm that efficiently transforms the previous search tree to the current search tree for moving target search.

References

[Bulitko and Lee, 2006] V. Bulitko and G. Lee. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, 25:119–157, 2006.

[Edelkamp, 1998] S. Edelkamp. Updating shortest paths. In *Proceedings of the European Conference on Artificial Intelligence*, pages 655–659, 1998.

[Ishida and Korf, 1991] T. Ishida and R. Korf. Moving target search. In *Proceedings of International Joint Conference in Artificial Intelligence*, pages 204–211, 1991.

[Koenig and Likhachev, 2005] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *Transaction on Robotics*, 21(3):354–363, 2005.

[Koenig *et al.*, 2004] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine*, 25(2):99–112, 2004.

[Koenig *et al.*, 2007] S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1136–1143, 2007.

[Moldenhauer and Sturtevant, 2009] C. Moldenhauer and N. Sturtevant. Optimal solutions for moving target search. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1249–1250, 2009.

[Stentz, 1995] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of International Joint Conference in Artificial Intelligence*, pages 1652–1659, 1995.

[Sun *et al.*, 2008] X. Sun, S. Koenig, and W. Yeoh. Generalized Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 469–476, 2008.

[Sun *et al.*, 2009] X. Sun, W. Yeoh, and S. Koenig. Efficient incremental search for moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2009.

[Trovato and Dorst, 2002] K. Trovato and L. Dorst. Differential A*. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1218–1229, 2002.

Revisiting Bounded Suboptimal Heuristic Search

Jordan T. Thayer
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
jtd7, ruml at cs.unh.edu

Abstract

Although the first algorithm many associate with heuristic search is A*, an algorithm for finding optimal solutions to search problems, many applications of heuristic search only require that we find a good enough solution or a solution quickly enough where good enough and quickly enough are defined by the needs of a user. Both problems have been addressed previously, by bounded suboptimal search and anytime search respectively, but these approaches are not satisfactory. My doctoral work will address these problems by developing new bounded suboptimal algorithms which perform better than the previous approaches. I will show that anytime searches can incorporate these new algorithms to improve their own performance. I will demonstrate that anytime search is not the correct approach when a deadline is known at the beginning of the search, and introduce deadline aware search algorithms which address this problem directly.

The Problem

Heuristic search is used to solve a wide variety of problems, such as pathfinding in a grid or temporal planning. By systematically exploring the space outwards from a starting state, a solution will eventually be found if one exists. When sufficient resources are available, optimal solutions can be found using A* search with an admissible heuristic (Hart, Nilsson, and Raphael 1968). Unfortunately optimal solutions take a large amount of time and memory to calculate. In practical applications this is unacceptable, and a search which finds solutions of reduced quality while consuming fewer resources might be more desirable.

In these practical settings we must accept suboptimal solutions in order to reduce the computation required to find solutions. When abandoning solution optimality we still have ways of retaining control over the search. We can choose to bound the quality of the solution returned by designing searches which return a solution within a given factor of optimal. Alternatively we can limit the time used in finding an the solution. When bounding the quality of a solution our aim then becomes to solve problems as quickly as possible while guaranteeing that the solutions found have a cost within a bounded factor of optimal. When we choose to bound the time, our goal is to produce the best possible solution within the given deadline.

Although the problems can be stated separately, they are closely interwoven. Searches which find solutions of bounded quality quickly are typically applicable in the setting where expedient problem solving is desirable, and searches which find solutions of good, but unbounded, quality quickly can be extended to find solutions of bounded quality. In my thesis, I will investigate improvements to both of these closely related problems.

Bounding Solution Quality

First I consider the setting in which one wants the fastest possible search where the solution is guaranteed to be within a bounded factor of the optimal solution. For a given factor w, we call an algorithm w-admissible if it is guaranteed to return a solution that is no more than w times more expensive than an optimal solution.

Weighted A* is an elegant answer to this problem (Pohl 1970). In weighted A* the traditional node evaluation function of A*, $f_{A^*}(n) = g(n) + h(n)$, is modified to place additional emphasis on the heuristic evaluation function h, as in $f_{wA^*}(n) = g(n) + w \cdot h(n)$. The additional focus on h causes the search to become greedy. Any solution found by weighted A* can be shown to be w-admissible.

Weighted A* often finds solutions within a given quality bound quickly, but it is not the fastest algorithm for a given suboptimality bound. The dissertation will present three approaches for improving bounded suboptimal search in an effort to outperform current state of the art algorithms. These new algorithms will improve performance by focusing on finding solutions, improving the heuristic online, and exploiting the average case performance of existing algorithms.

Focus on Finding the Solution

The purpose of bounded suboptimal search is not really to find a solution whose cost is within a given bound of the optimal, but rather, given a desired quality bound, to produce an acceptable solution as quickly as possible. These solutions should be of high quality but not at the expense of speed. To do this, bounded suboptimal algorithms make a controlled transition between performing like A* when the

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

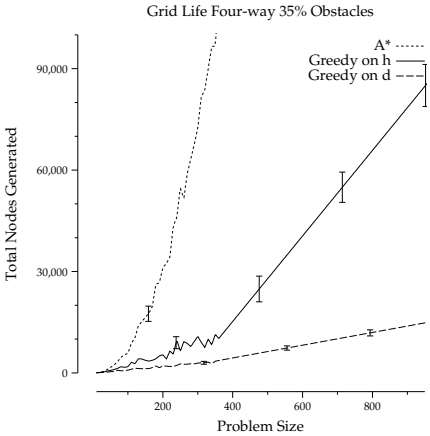


Figure 1: Greedy search on d outperforms search on h.

bound is tight and performance like that of greedy search when the bound is lax.

In some domains all actions have the same cost and becoming greedy with respect to the cost of a solution is equivalent to becoming greedy with respect to the length of a solution. Not all domains have unit cost actions, and so it is useful to separate the idea of pursuing solutions which are cheap and pursuing those which are close. Searches that consider both the estimated cost-to-go, given by a heuristic evaluation function h, as well as the search-distance-to-go, given by a distance estimation function d, should have an advantage over searches which do not make use of this additional information, particularly in domains where h and d differ. In these domains a search that is greedy with respect to solution cost may take longer than a search that focuses on finding the nearest solution. Figure 1 demonstrates the extreme case, where no bound is placed on the quality of the returned solution in a grid pathfinding problem. We see that not only do we find solutions faster when greedily searching on d than when we do on h, but that the performance gap between these two approaches increases with problem size. Domains where solution cost and length differ are common, including popular benchmarks such as temporal planning and many variations of pathfinding.

Many algorithms fail to take advantage of the information provided by d. Although there were several proposals in the 70s and 80s, A^*_ϵ (Pearl and Kim 1982) and dynamically weighted A* (Pohl 1973) among them, using distance estimates has fallen out of favor. This is caused in part by the poor performance of these two algorithms. The root of this poor performance has been identified (Thayer, Ruml, and Kreis 2009), and it will be corrected as part of my thesis.

A^*_ϵ works well for loose bounds, where the algorithm searches greedily on d, but can take longer than A* to find suboptimal solutions for tighter bounds (Thayer and Ruml 2009). A^*_ϵ considers nodes who appear to have cost close to that of the optimal solution, preferring to expand nodes which appear to be close to a goal first. While this sounds like a wonderful idea in principle, in practice these two node orderings can conflict with one another and lead to poor performance. When the orders are conflicting, nodes which are close to the goal do not appear to have costs within a desired bound, and nodes which have desirable costs do not seem close to a solution. The end result is that A^*_ϵ slowly empties the set of nodes which appear to be good candidates for expansion until the lower bound on optimal solution cost is raised. Then, the process starts over again. This repeated flushing and refilling of the focal list causes poor performance.

Dynamically weighted A^* performs poorly in domains where the search proceeds unevenly towards goals. Decreasing the weight by which the heuristic evaluation function is multiplied as depth increases encourages progress in any direction. This is fine in when each step away from the goal, regardless of the action, takes the search equally closer to a goal but is bad in general.

These two algorithms are exemplary of the two approaches for incorporating a focus on search distance into general heuristic search. We can either use the distance-to-go estimate to augment the apparent cost of a node, thereby altering the search order, or we can use the estimates to select from a subset of promising candidates. Methods have already been developed to correct the poor performance of dynamically weighted A* (Thayer and Ruml 2009), and improvements to A^*_ϵ have also been made (Thayer, Ruml, and Kreis 2009). Both approaches deserve further study.

Improve Heuristics

If the heuristics used in the search were perfectly accurate, then we would expand nodes directly along an optimal solution path into the goal. As the heuristics degrade in quality, optimal solutions become harder to find at an alarming pace (Helmert and Röger 2007), but suboptimal solutions can still be easy to find (Davis 1991). As the heuristics typically used are admissible, that is they always underestimate, inflating them by a set factor can be seen as an attempt to correct for an underestimating heuristic. A more principled approach to correction can lead to better performance.

Clamped adaptive search (Thayer, Ruml, and Bitton 2008) is an algorithm which allows for the use of an inadmissible cost-to-go heuristic so long as an admissible heuristic is also provided. In short, it follows a, presumably, more accurate inadmissible heuristic so long as the costs calculated by this heuristic fall within some bound. More specifically, clamped adaptive search is a best first search order on the function $\widetilde{f}(n) = max(g(n), (min(w * f(n), \widehat{f}(n))))$. $g(n)$ is the cost of arriving at node n when traveling from the root. $f(n) = g(n) + h(n)$ where $h(n)$ is some admissible heuristic. Similarly, $\widehat{f}(n) = g(n) + \widehat{h}(n)$ where $\widehat{h}(n)$ is an inadmissible heuristic. The question then becomes how do we derive reasonably accurate inadmissible heuristics automatically. One approach is to use regression techniques to learn the heuristics. There are at least three possible sources of information to learn from: the truth, single step heuristic error, and backwards looking heuristics.

Truth One obvious thing to do when attempting to evaluate the performance of heuristics is to look at the true values which they are trying to estimate. The true values can be calculated by exhaustively searching the space backward from the goal. Error in the admissible heuristic can then be calcu-

lated by means of subtraction. Comparing to the truth is desirable because the error measurements are known to be exactly correct and completely removed from the search being performed. Unfortunately it also requires a large amount of pre-computation which solves the problem as a by-product. As such, learning techniques which rely on the truth are primarily interesting as a benchmark. They show us the limits of a particular model when learned under the best possible conditions. In practice, models learned on the truth of one instance appear to be reasonable for other very similar instances, a phenomena I will explore more thoroughly in my thesis.

Single Step Heuristic Error During the course of a search, nodes are selected for expansion and their children are generated. These single expansions can tell us a great deal about the performance of a heuristic. If the cost-to-go heuristic h were perfect, then the f value of a parent node would be the same as the lowest f among its children. Admissible heuristics underestimate the cost to go, and so the cost of the cheapest child may be larger than that of the parent. The rise in value from a parent to its best child is a measurement of the error in the heuristic. To use the single step error to correct the heuristic, an estimate of the number of steps to go must be provided. This estimate can be derived from the cost to go heuristic, in fact for unit cost domains they are identical, or can be generated separately.

Backwards Looking Heuristics Another possibility requires us to create a separate set of heuristics which, rather than looking forward from a node towards the goal, estimate the cost and distance of arriving at the root from the current node. It is important that they be as similar to the forward heuristics as possible. The reverse heuristics are then compared to the cost of arriving at the current node. If the heuristics are similar then we should be able to infer things about the performance of the forward looking heuristic from the backwards looking heuristic. We can reduce from here to the single step error case by distributing the observed error over the number of steps needed to reach the current node.

Exploit Average Case Performance

We begin by noting that previous approaches to suboptimal heuristic search, including weighted A*, dynamically weighted A*, and A$^*_\epsilon$, are very strict in so much as no node is ever expanded which could not lead to a w-admissible goal. While this strict approach to w-admissibility has a certain conservative appeal, there is also a long history in AI research of exploiting the fact that worst case behavior rarely occurs. This gives rise to an optimistic approach in which we allow ourselves to expand nodes more aggressively, without a strict guarantee that they lead to a w-inadmissible solution. Any fast search method with an open list can be used. In the experiments below, we use weighted A* with a high weight. Crucially, once we have found a solution using the aggressive search, we will then expand additional nodes until we can either prove our solution is w-admissible.

Figure 2 shows the true quality of solutions found by weighted A* compared to the suboptimality bound, and used

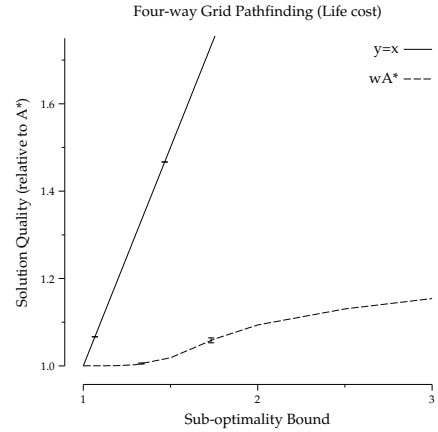


Figure 2: The suboptimality of solutions returned by weighted A* versus the suboptimality bound, averaged over 20 grid-world path-finding problems.

weighting factor, w. The line $y = x$ is also included to show the theoretical limit. Clearly, not only is it possible for weighted A* to return solutions far better than the bound suggests, it is also common in some domains. This gap represents the difference between worst case and average case performance.

The optimistic search algorithm exploits this gap by using a two phased search. First, it employs an aggressively greedy search phase to find a solution that is not guaranteed to be w-admissible, but usually is. This is followed by a 'clean-up' phase, in which the algorithm attempts to prove that the solution obtained by the aggressive part of the search is w-admissible.

Bounding Solving Time

Additionally, I will consider the problem of finding solutions under a time pressure. Here, the user is constrained by time and not by solution cost, though ideally we'd like to find the best solution possible within the deadline. In these problems, only the best solution returned found within the time allotted counts, and not returning a solution within the deadline is unacceptable. Even with these restrictions, these problems still fall into two categories. We may or may not know the deadline ahead of time.

Unknown Deadlines

The current crop of anytime heuristic search algorithms directly address the problem of finding solutions to problems within an unknown deadline. Anytime heuristic search (Hansen and Zhou 2007), ARA* (Likhachev, Gordon, and Thrun 2004), and restarting weighted A* (Richter, Thayer, and Ruml 2009) are a natural fit for problems where a deadline exists, but we do not know what it is at the outset because they greedily search towards an initial solution and then return a stream of ever improving solutions. Both of these algorithms rely on weighted A*, which may not be the best bounded suboptimal algorithm for all domains.

We can adapt our improved bounded suboptimal search

algorithms into anytime algorithms following a technique outlined by Hansen and Zhou, 2007 If the underlying algorithms are better than weighted A^* we would expect the anytime versions of these algorithms to outperform anytime searches relying on previous state of the art techniques.

Fixed Deadlines

Anytime searches naturally work for the case of deadlines which are fixed before the search begins. They race towards an initial solution and then incrementally improve upon it until the deadline arrives and the search is halted. The main flaw in using the anytime approach to search under a specified deadline is that only the final solution is returned. In a sense all of the computational effort spent finding all of the solutions save for the final solution which is returned is wasted.

For problems with known deadlines, we would ideally invest all of the time up until the deadline in finding a single solution. This should allow the search to find a solution of better quality than if it had invested its time in finding a stream of solutions. The problem of search within a fixed deadline has two parts which must be addressed: How do we predict the time it will take a search to complete and how do we quickly find solutions of good quality?

The second question will have been addressed previously by my work in improving bounded suboptimal search. Searches which find solutions of a know quality as quickly as possible are a natural fit for the problem of quickly finding solutions of a good quality as we have simply put a strict limitation on the good qualifier. The first question, how can we predict the time it will take a search to complete, is more complicated. There is previous work for predicting the time it will take to find an optimal solution using IDA* (Korf, Reid, and Edelkamp 2001), and some work has been done on predicting the progress of search as it is being conducted (Hiraishi, Ohwada, and Mizoguchi 1998). Neither of these techniques directly addresses the problem of predicting the time a suboptimal search, and so they will have to be built upon and expanded in order to address the problem at hand.

Conclusions

General heuristic search is one of the cornerstones of more advanced methods in artificial intelligence. There are a great many techniques which rely on systematic search, and improvements to systematic search should extend to techniques which rely on it. Although the prevailing sentiment is that the problem of bounded suboptimal search has been solved, leading approaches to the problem do not directly address the issue of solving problems within the bound as quickly as possible; the loosening of the suboptimality bound simply has the pleasant side effect of reducing the time needed to find a solution. Directly addressing the problem should lead to better solutions.

Improvements to bounded suboptimal search will extend directly to anytime search, by virtue of the fact that bounded suboptimal searches can be directly extended into anytime search. Improvements in anytime search will help better address the problem of search under an unspecified deadline. It will not optimally solve the problem of search under a known deadline however. Doing this will require an extension of existing techniques, or perhaps the development of new techniques, in order to predict the time it will take an algorithm to find a solution. Again, the expectation is that a direct approach to perform better as it is designed to directly address the problem at hand.

References

Davis, S. V. C. H. W. 1991. High-performance a* search using rapidly growing heuristics. In *Proceedings of IJCAI-91*.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *JAIR* 28:267–297.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics* SSC-4(2):100–107.

Helmert, M., and Röger, G. 2007. How good is almost perfect? In *Proceedings of the ICAPS-2007 Workshop on Heuristics for Domain-independent Planning: Progress, Ideas, Limitations, Challenges*.

Hiraishi, H.; Ohwada, H.; and Mizoguchi, F. 1998. Time-constrained heuristic search for practical route finding. In *Pacific Rim International Conference on Artificial Intelligence*, 389–398. Springer.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-a*. *Artificial Intelligence* 129:199–218.

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS 16*.

Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4(4):391–399.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1:193–204.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computation issues in heuristic problem solving. In *Proceedings of IJCAI-73*, 12–17.

Richter, S.; Thayer, J.; and Ruml, W. 2009. The joy of forgetting: Faster anytime search via restarting. In *Symposium on Combinatorial Search*.

Thayer, J., and Ruml, W. 2009. Using distance estimates in heuristic search. In *Proceedings of the 19th International Confyerence on Automated Planning and Scheduling*.

Thayer, J. T.; Ruml, W.; and Bitton, E. 2008. Fast and loose in bounded suboptimal heuristic search. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*.

Thayer, J.; Ruml, W.; and Kreis, J. 2009. Using distance estimates in heuristic search: A re-evaluation. In *Symposium on Combinatorial Search*.

A Hybrid Assembly Task Planning System: Where Motion Planning Helps Symbolic Planning Find Good Solutions For Real-World Applications

Frederik W. Heger and **Sanjiv Singh**
The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Introduction

People's first reaction to the term "robotic assembly" is usually a mental picture of an industrial assembly line where stationary robots perform repetitive tasks at high speeds and with high precision. That is not the kind of "assembly" our work is about. Instead, we consider mobile manipulators retrieving components from a storage location, transporting them through their environment and assembling them into a large structure. We are developing a framework for planning assembly tasks that, given a desired goal structure, automatically decomposes the task and commands robots to execute them. Our system seamlessly reacts to execution-time failures – which inevitably will occur – by repairing and re-planning the task as necessary.

Assembly is naturally a step-by-step process where a sequence of actions transforms an initial (disassembled) state into a desired final (assembled) state. For assembly tasks performed by real robots and in non-trivial environments, symbolic plan solutions at the level of intermediate structure configurations are insufficient to generate desirable plans. To overcome this limitation, the planner requires a way to reason about physical feasibility of the assembly steps it considers. Motion planning techniques are uniquely suited to provide such information, but they require the large assembly problem to be broken down into smaller sub-problems to be able to find solutions.

We present a hybrid planning system that combines a graph-based symbolic representation of an assembly problem with continuous (motion planning) reasoning about physical feasibility of assembly steps during planning. We consider lattices of beams and nodes (see Fig. 1) that have to be assembled by mobile manipulators in constrained environments. Our planner has been successfully applied to assembly problems in simulation and with real robots.

Vision and Motivation

We envision an assembly planning system where the user or operator only has to supply a desired goal structure to be assembled and a place where the final assembly is to be located within the environment, and an automatic planner takes care of the rest. In cases where full autonomy is not desirable, an operator could be informed of key steps along the assembly task and help make decisions, or he could be involved more actively by assisting the robots (on request or proactively) to resolve and avoid problems.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

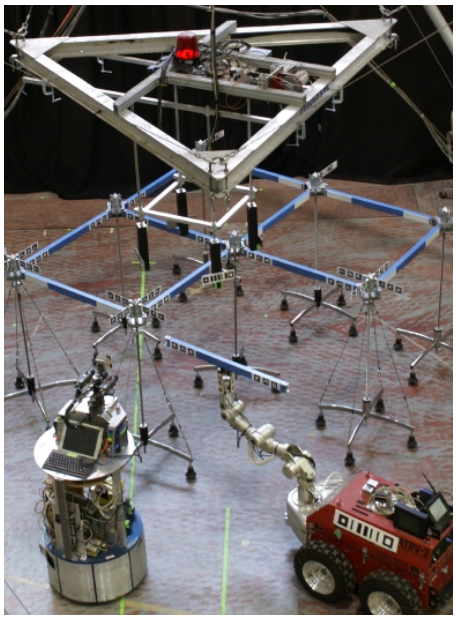


Figure 1: A nearly completed lattice of 21 components. Structural (internal components before external components) and environmental (the workspace is tight – can the robots get to where they need to be to perform their tasks?) constraints need to be considered by the planner.

Such a system will be of great interest to NASA in the context of establishing planetary outposts or habitats where human presence (especially during early phases of construction) is prohibitively expensive, not to mention dangerous for the workers. If robots can be sent ahead to prepare the necessary infrastructure with little supervision or help from remote humans, astronauts can focus on aspects of the mission they are better suited for, such as science experiments.

As a more down-to-earth example, imagine furnishing a new apartment with your favorite assemble-yourself furniture – or rather, have robots assemble all the furniture according to your plan of where the bed/desk/shelf/... go while you are exploring your new neighborhood.

State of the Art

Assembly is a challenging and difficult task (even for humans) with such complicating factors as tight tolerances, heavy and/or large parts to manipulate and more. By definition, robots will have to operate close to other objects, and they have to be able to sense, navigate, manipulate while constructing a growing obstacle in their already constrained workspace. Intertwined with the necessity to reason about motion through the environment, there is inherent structure to an assembly where internal components must be installed before the outer layer is complete.

Existing approaches to assembly planning are limited to highly structured environments (e.g., work cells, (Homem de Mello and Sanderson 1988; Kaufman et al. 1996)). Other applicable work focuses on the motion planning aspects of the problem (Karagöz, Bozma, and Koditschek 2004; Lengyel et al. 1990), but plans produced by those approaches do not always satisfy critical structural constraints imposed by the structure to be assembled. Manipulation planning is faced with similar challenges to assembly planning at a finer level of detail (e.g., dextrous motions to grasp and re-grasp components (Nielsen and Kavraki 2000; Gravot, Alami, and Siméon 2002)). An assembly plan sets up manipulation planning problems for each step in the plan. To our knowledge, there is currently no system that is able to automatically generate assembly plans for mobile manipulators assembling large structures in realistic environments. While various parts of the overall problem have been addressed by previous work, a comprehensive solution has not yet been developed.

We have worked on robotic assembly scenarios for several years. This paper extends our previous work (Heger 2008) with a more efficient implementation to handle structures of up to 21 components (instead of 8 previously), a task executive that takes generated plans and directly turns them into task trees for robots to execute, and plan repair and re-planning capabilities throughout all levels of the planning hierarchy to allow the system to effectively react to and resolve execution-time exceptions. The planner is tied to the simulated or real-robot scenario using a task sequencing executive that also provides capabilities for sliding autonomy interaction with a user. This work is motivated by earlier experiments with real robots (Sellner et al. 2006) where a rather simple assembly plan was scripted by hand – a task that is not feasible for larger and larger structures.

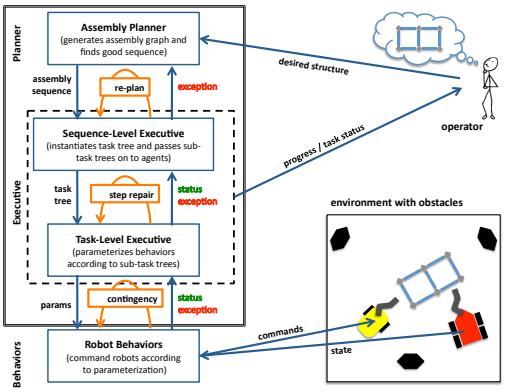


Figure 2: High-level overview of our assembly planning system. The user specifies an assembly goal. The assembly planner generates nominally feasible assembly plans, which the executive converts into a task tree to be executed by robots. Execution-time exceptions are handled to repair/re-plan the assembly sequence as necessary.

Contributions

Our work makes the following key contributions toward the goal of a more general and useful assembly planning system:

1. An expressive representation of assemblies that decomposes the problem into sub-problems that can be attacked using existing planning methods.
2. A method to automatically generate plans for mobile manipulators to assemble a desired structure.
3. Seamless repair and re-planning of assembly sequences in response to execution-time failures and exceptions.

Approach and Implementation

Our assembly planning framework consists of four main components (see Fig. 2): the assembly planner itself that generates the assembly sequence, an executive responsible for parameterizing robot behaviors, real or simulated robots to execute those behaviors and exception handling mechanism to recover from execution-time failures.

Assembly Planner

The assembly planner is the workhorse of the system. Given a desired goal structure, an environment, available robots and a final pose for the assembly, its goal is to produce a sequence of assembly steps (i.e., *AssembleComponent 1* → *AssembleComponent 2* → ... → *AssembleComponent N*) that are nominally feasible (assuming some characteristic information about the robots involved and no significant deviations from the plan during execution). Obviously, nominal feasibility is no guarantee for success, but it is usually a good start, and fewer plan repairs are necessary.

Assembly Graph The underlying representation of the assembly planning problem (Heger 2008) is a directed graph with unique assembly states (intermediate configurations of the structure as it is being assembled) at its vertices and the plans necessary to add a new component to the growing structure on its edges. Edges are weighted by the goodness of their associated plans, and vertices are scored on the structural properties of the partial assembly they represent. Assembly plans are traces through this graph from an initial configuration (usually the disassembled state) to a final configuration (usually the assembled state). The chosen graph edges contain all the parameters necessary to enable the executive to create the corresponding task tree. The planner searches the assembly graph using A* to find the assembly sequence to publish to the executive.

The basic version of the assembly graph contains vertices for each structure state between completely disassembled and fully assembled and edges with the appropriate component being added to transition from one state to the next

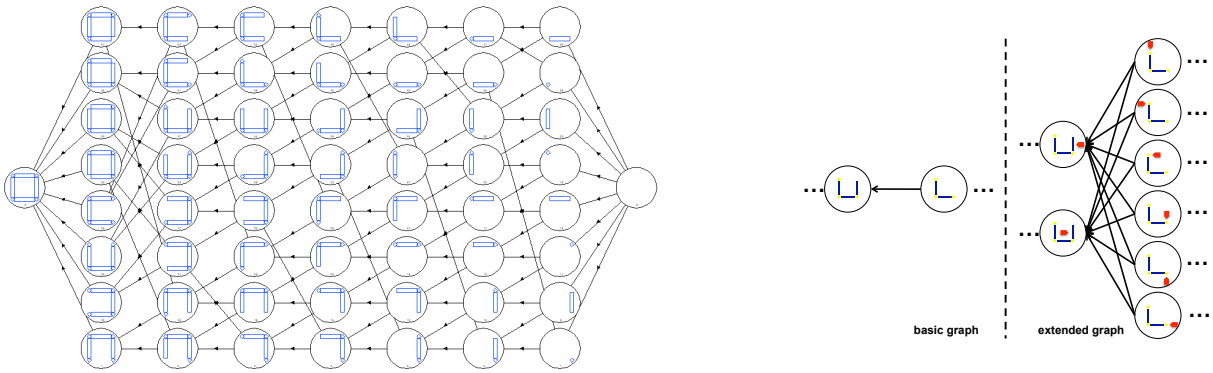


Figure 3: (left) Basic graph representation of a simple assembly problem (a one-square subset of the assembly shown in Fig. 1). Vertices mark structure configurations, edges indicate assembly steps to transition from one configuration to another (by adding a new component to the structure). (right) Graph expansion to include robot state in addition to structure state.

(Fig. 3 (left), see (Heger 2008) for more detail). The graph is constructed goal-to-start by beginning with the fully assembled structure at vertex 0 and then considering all valid components for removal (only components external to the structure are considered – internal components are considered not reachable by robots working around the structure). This graph generation process makes the implicit assumption that removing a component from a structure state is simply the reverse of installing it in the previous state. This is a common assumption in assembly planning, and all assembly steps considered in this paper meet this requirement.

For planning, we use an extended graph(see Fig. 3 (right)) that encodes both the state of the structure as well as the state of of assembling robot(s) around the structure. This expansion allows us to explicitly include costs associated with robots moving through the environment in order to accomplish their tasks as part of the planning process. While it takes more time up-front to construct the extended graph, the greater expressive power of this representation allows the planner to find better plans for larger structures. In addition, the assembly graph is static for a given structure (the graph structure does not change, only the weights of vertices and edges during the search step depends on the specific environment) and can be pre-computed and stored.

Graph Search and Plan Generation With the assembly problem represented as an assembly graph, finding a plan becomes a graph search problem. Any trace through the graph from the vertex marking the fully disassembled structure to the vertex marking the completed assembly is a candidate plan that satisfies all assembly requirements at a symbolic level (i.e., all states are valid along the way, internal components are dealt with while they are reachable, etc.). In addition, motion costs can be taken into account when searching the extended graph. The graph search directly returns a plan that is guaranteed to be at least nominally feasible and fully parameterized for immediate robot execution.

Our work to date has been focused on enabling planning for realistic structures like the ones we can build in our test bed and to provide recovery capabilities for execution-time failures that we must anticipate with any real-robot system. Given that the robots' speed of operation during assembly is fairly slow, optimizing for efficiency has been a lower priority than increasing system robustness. The planner in its current implementation examines approximately 70% of the graph edges and spends 0.11-0.16 seconds per edge.

Plan Repair and Re-Planning

The generated assembly plan is guaranteed to be nominally feasible. As the real robot attempts to follow the planned motions, or as other run-time exceptions occur (e.g., targets out of the field of view of the camera, some error during manipulation tasks, etc.), the system will have to be able to handle various exceptions. Some of these exceptions may be "caused" by the rather simple motion planning methods we are currently using. Our observations of the system in operation suggest that a lot of the time these methods are sufficient, and when they are not, the robots are able to detect that they are in an off-nominal state, stop and request help. The most useful place to introduce more sophisticated planning techniques will be for error recovery where we can leverage additional capabilities without requiring unnecessarily complex computation throughout the entire system.

We consider three distinct levels of failure recovery. At the lowest level, as a first recovery attempt, each robot should have simple contingency responses for things that are known to go wrong from time to time. Often "try again" is a valid recovery strategy. For such contingency recovery actions, the assembly planner never gets involved. However,

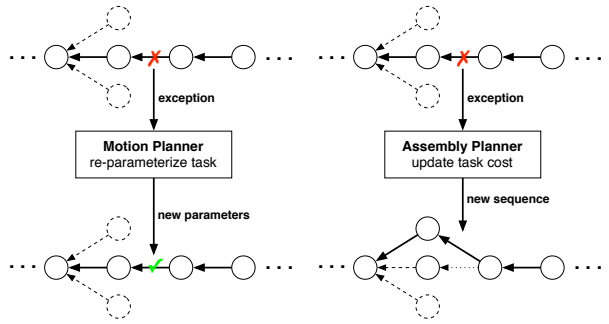


Figure 4: Plan repair (left) and re-planning (right) in response to execution-time failures.

after a number of contingency attempts fail, more work is required to continue on with the task.

Failed assembly operations are associated with assembly graph edges. The next recovery attempt is to repair the failed step only (Fig. 4 (left)). Enforcing continuity with the original plan and taking into account any new information due to the failure, the planner tries to re-parameterize the failed task. Depending on how far along the assembly step the error occurred, the planner may have different options to repair a step. If a repair is possible, the affected assembly step is reparameterized and execution continues.

If no repair is possible the exception jumps up to a higher level in the executive and the planner is queried for a new sequence from the current state of the assembly to the desired goal state. In this case, the offending graph edge is marked impassable, and a new graph search is run from the source state of the failed edge to the original target state (Fig. 4 (right)). Since the failed assembly step left the robot somewhere along its task, the re-planned sequence may need to be prepended with setup tasks that return the robots to a clean state before continuing with the new plan. While the requirement to "physically backtrack" to a clean state can pose challenges to the planner, they provide opportunities where operator input (e.g., via sliding autonomy (Sellner et al. 2006)) can provide great benefit to improve the robustness of the overall system.

The Planner at Work

We use a simulator/visualizer based on OpenRAVE (http://openrave.programmingvision.com/) to examine the output of the planner and to exercise the system's repair and replanning capabilities. The executive interacting with the planner and tasks the robots is the same for the simulated robot case as it will be for real robots.

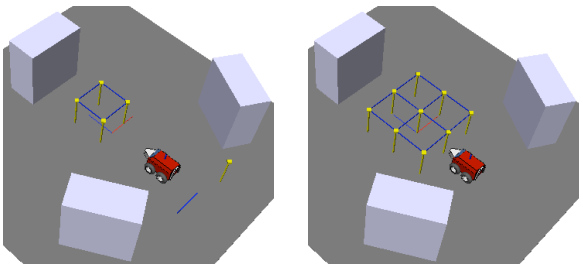


Figure 5: Two simulated example assembly scenarios.

The base case to demonstrate the planner's capabilities is a scenario we had worked with previously: robots assembling a square of four nodes and four beams (Fig. 5 (left)) according to a hand-written script (Sellner et al. 2006). For this scenario, the assembly graph contains 246 vertices, 924 edges and $1.3 \cdot 10^5$ candidate assembly sequences.

We increased the complexity of the structure via a two-square lattice of 13 components up to a four-square lattice (Fig. 5 (right)). That is the largest structure we can build with the hardware we have, and we are working toward being able to show this assembly using real robots. This 21-component assembly is represented by a graph with 66,566 vertices, 460,511 edges and $1.6 \cdot 10^{17}$ candidate traces and is the largest structure the current implementation of the assembly planner can represent.

This progression in graph sizes for still fairly simple structures clearly indicates that just continuing in this manner does not scale sufficiently to be able to plan for structures of 100+ components. In order to solve this problem, we are currently investigating approaches that avoid constructing the entire assembly graph explicitly but rather only construct as much graph as is needed to find a solution.

Summary

We have presented a hybrid planner for assembly problems that combines symbolic graph-search with motion planning techniques to evaluate nominal feasibility of the considered actions. The resulting plan can be executed directly by mobile assembly robots without additional operator intervention. As execution-time exceptions occur, the executive controlling the robots interacts with the planner to repair and replan the assembly sequence as necessary and appropriate.

While we have made some progress toward a system as described in our vision, many issues remain to be tackled. We are evaluating recent advances in motion and manipulation planning to extend the capabilities of our planner.

References

Gravot, F.; Alami, R.; and Siméon, T. 2002. Playing with Several Roadmaps to Solve Manipulation Problems. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*.

Heger, F. W. 2008. Generating Robust Assembly Plans in Constrained Environments. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

Homem de Mello, L. S., and Sanderson, A. C. 1988. Autommatic Generation of Mechanical Assembly Sequences. Technical Report CMU-RI-TR-88-19, Carnegie Mellon University, The Robotics Institute, Pittsburgh, PA.

Karagöz, C. S.; Bozma, H. I.; and Koditschek, D. E. 2004. Feedback-Based Event-Driven Parts Moving. *IEEE Transactions on Robotics* 20(6):1012–1018.

Kaufman, S. G.; Wilson, R. H.; Jones, R. E.; Calton, T. L.; and Ames, A. L. 1996. The Archimedes 2 Mechanical Assembly Planning System. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, volume 4, 3361–3368.

Lengyel, J.; Reichert, M.; Donald, B. R.; and Greenberg, D. P. 1990. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. *Computer Graphics* 24(4):327–335.

Nielsen, C. L., and Kavraki, L. E. 2000. A Two Level Fuzzy PRM for Manipulation Planning. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*.

Sellner, B.; Heger, F. W.; Hiatt, L. M.; Simmons, R.; and Singh, S. 2006. Coordinated Multi-Agent Teams and Sliding Autonomy for Large-Scale Assembly. *Proceedings of the IEEE* 94(7).

Issues in Temporal Planning using Timed Petri Nets

Melissa Liew
Doctorial Consortium Extended Abstract
School of Electrical and Electronics Engineering, University of Adelaide

Abstract

Temporal planning problems with concurrency is the main focus of this paper. An introduction to solving concurrent temporal planning problems using Timed Petri Nets (TPNs) is described. We use an innovative method, whereby temporal planning problems are translated into TPNs and representing the TPNs into an algebraic description. The key approach that is presented here is Policy Evaluation. It details how this approach efficiently obtains optimal temporal plans. In addition, as the definition of concurrency is not uniform in the planning community, we provide the definition of concurrency in our framework.

Introduction

Over the recent years, progress has been made in the area of Temporal Planning, in hope that it will solve real-world problems. However, many complexities arise due to the concurrent and probabilistic nature of real-world planning problems. The interest of this paper is temporal planning problems with concurrency. As a consequence, we will disregard probabilistic planners such as, PROTTLE (Little, Aberdeen, and Thiébaux 2005), PARAGRAPH (Little and Thiébaux 2006), and CoMDP (Mausam and Weld 2005), that handles probabilistic temporal planning problems. Many architectures for temporal planning have been researched. Some of which will be discussed here : State-space search [SGPlan (Chen, Hsu, and Wah 2006), Sapa(Do and Kambhampati 2003), TP4(Haslum and Geffner 2001), TLPlan (Bacchus and Addy 2001) and TALplanner (Doherty and Kvarnstrom 2000)]; Planning Graph [LPGP (Long and Fox 2003)] and Petri nets [Directed Unfolding (Hickmott 2008)].

(Agha 1985) defined concurrency to execute a number of processes concurrently without assuming anything about their concrete realisation. As the key interest of this paper is concurrency, we ask the question: "Is there a uniform semantics of concurrency in Temporal planning?". It is observed from literature that the concurrency semantics is not uniform in the planning community. In our approach, concurrency is defined to be independent of actions. Whereas the planning community defines actions happening at the same time to be concurrency. Hence we will further describe what concurrency means in our approach, and point out the differences in our definition as compared to other planners.

The motivation for our research is from (Hickmott 2008)'s efficient way of solving classical planning problems using Place-transition nets. (Hickmott 2008) is able to efficiently model and solve concurrent planning problem using McMillan's method called unfolding (McMillan 1995). We apply timing constraints to the Place-transition describe in (Hickmott 2008), by introducing durative actions. This leads us to an extension of Petri nets: Timed Petri nets. Hence we present the key difference and indeed innovation of our approach: modelling concurrent temporal plans, using Timed Petri Nets (TPNs), and representing them in an algebraic description. A key aspect of the planning approach that we present here is Policy Evaluation. It is an efficient and innovative method based on solving a set of dynamic equations, described by a (min, max, +) hybrid dynamical system. The solution defines the dynamic evolution of the planning problem under a specified policy. Our approach works in a so-called "plan space" or the space of propositions and durative actions. Thus this avoids the problem of having a combinatorial explosion faced by state-space methods, due to the artificial interleaving of concurrent actions (McMillan 1995). Actions represented in our TPN framework do not depend on the global clock. The duration associated to the actions are not decision epochs (Cushing et al. 2007), that is we do not restrict possible start time of actions to a small set of special time point. We quote (Cushing et al. 2007) that, "by anchoring actions to absolute times appears doomed".

The reader should be aware that the purpose of this paper is to raise problems such as concurrency in temporal planning to the community, discuss our proposed approach and describe its semantics. We also address the issues that the reviewers have brought up. The layout of the paper is as follows: First, various temporal planning methods will be examined. Next, we introduce the notion of concurrency defined in our framework. We then present our approach on temporal planning and Timed Petri nets. Finally, we conclude by establishing future work.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Temporal Planning Methods

State-space Search

State space methods were originally used for classical planning method. But have now been extended to include temporal and logic conditions. Many efficient techniques have been developed to conduct state-space search, both in the classical and temporal domains. In the state-space approach, there is a total ordering of the planning problem where all possible states are enumerated. As cited by (Do and Kambhampati 2003), temporal planning problems have larger planning state spaces. This then results in the state-explosion problem that is commonly faced by state-space methods. Examples of state-space temporal planners are as followed: SGPlan (Chen, Hsu, and Wah 2006), TP4 (Haslum and Geffner 2001) , Sapa (Do and Kambhampati 2003) , TLPlan (Bacchus and Addy 2001) and TALplanner (Doherty and Kvarnstrom 2000). Concurrency in the state space becomes complicated due to all the interleaving of actions. It should be noted that state-space approaches tend to lose the notion of true concurrency. This is due to the fact that they are less expressive and the heuristics used for state-reachability are mainly for classical non-temporal planning problems, as described in (Cushing et al. 2007).

Planning Graph

Long and Fox (2003) produced a temporal planner, LPGP (Long and Fox 2003), to exploit the use of Graphplan (Blum and Furst 1997) in order to handle temporal planning problems. LPGP models the logical structure of the plan by using the layers of the graph. That is a duration is assigned to each state, hence each layer of the planning graph is associated with a duration. Sometimes, longer temporal plans tends to have simpler activity structure than fewer plan graph layers (Long and Fox 2003). This approach retains some concurrency, however optimality is sacrificed.

Petri Nets

Petri nets is a mathematical and graphical tool that is originally used for analysis and modeling of distributed systems (Hickmott 2008). (Hickmott 2008) exploits this representation to solve classical planning problem. As described in (Hickmott 2008), Petri nets encourage a partial ordering on actions in accordance with the cause and effect relation, resulting it to be a plan-space methodology. (Hickmott 2008) solves untimed Petri nets (which describes a classical planning problem) via Directed Unfolding. In her thesis, she suggested and tested using this methodology on temporal planning problems. In this approach, durative action was added as a cost function in the analysis. We shall argue that this approach to solving temporal planning problems is inefficient in terms of computational and memory utilisation.

Timed Petri nets is an extension of the work of (Hickmott 2008), where we assign a duration to every action. TPN models partially ordered concurrent temporal plans, where preconditions are held at the beginning of an action and removed at the end of the duration of the action. In this framework, there is no global clock, which means that we do not anchor actions to any fixed time. TPN is observed to be L_e^s, a temporally expressive language according to the definition in (Cushing et al. 2007). Timed Petri nets, as addressed by (Baccelli, Cohen, and Gaujal 1992), can be described in an algebraic description. A set of dynamic equations is then resulted from the TPN. By solving this set of equations, we are able to efficiently obtain the optimal plan for a given policy. As a result, this solution and the set of equations obtained from the TPN, provide structural insights to planning algebraically.

Concurrency

Petri nets (PN) without timing constraints has a structural property of concurrency. This property is true if and only if independent actions do not change the preconditions of the actions and are not causally connected. As TPN is an extension of the Place-transition nets, TPN inherits this structural property. However, TPNs have an additional condition for actions to be concurrent. Actions in the temporal case, are temporally concurrent if and only if the actions are first structurally concurrent and the actions have to have the same duration. The ordering of the actions in the concurrent temporal plan does not matter, as it will reach the goal regardless. Our definition of concurrent in the temporal case is different to the standard definition of concurrency in the temporal planning community - actions that happen at the same time are defined to be concurrent.

Our Approach

Temporal planning problems are first converted into 1-safe Petri nets. Then it is translated to our TPN framework. With the temporal planning problem represented in our TPN framework, we are able to describe it in an algebraic semantics that is in the form of (min, max, +) hybrid equations. By applying a set of (max,+) operation on the algebraic formula, a solution of the temporal planning problem is obtained. The solution is optimal for a stationary policy. We strongly argue that the approach is efficient in processing and generating an optimal plan for the stationary policy.

Temporal Planning

A temporal plan consists of a set of actions, each having a strictly positive duration, and a set of preconditions and postconditions. Preconditions and postconditions are each sets of logical literals, ie the values taken by a set of propositions. There is an initial state, which is a set of literals which are initially true. In addition, there is a goal state. A feasible plan is a partially ordered set of actions which take the state for the system from the initial state to the goal state. An action is enabled whenever all its preconditions are true. After the action has completed it sets all its postconditions to be true. The plan terminates when all the goal literals are true. Often we are interested

in finding an optimal plan, namely a partially ordered set of actions that results in the goal being achieved in the minimum time over all feasible plans.

TPN Semantics and Algebraic Specification

A timed Petri net (TPN) consist of two sets (of nodes) $\mathcal{P} = \{p_1, \ldots, p_M\}$ and $\mathcal{T} = \{t_1, \ldots, t_N\}$ (called *places* and *transitions* respectively), and a *flow relation* $\rightarrow \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ which relates transition and places. Here M and N are positive integers which denote respectively the number of places and transitions in the TPN. We assume that every transition has at least one input place and one output place, ie for each $t_i \in \mathcal{T}, \exists\, p_j, p_k \in \mathcal{P} \ni (p_j \rightarrow t_i) \wedge (t_i \rightarrow p_k)$. We also assume that each place is connected to at least one transition, ie $\forall p_j \in \mathcal{P}, \exists\, t_i \in \mathcal{T} \ni p_j \rightarrow t_i \vee t_i \rightarrow p_j$. There is a mapping $\tau : \{1, \ldots, N\} \rightarrow (0, \infty)$ where $\tau(i)$ is called the *firing duration* of transition t_i. The state of the system is described by tokens. We say a place p is *marked* when there is a token present in p. We will assume that every TPN under consideration is *1-safe* which means that there is either no token in p or a single token in p. Let $M : [0, \infty) \rightarrow 2^{\mathcal{P}}$ be a mapping which describes which places have a token at a given time, ie the set $M(s) \subseteq \mathcal{P}$ is precisely the subset of places which have tokens at time s. We denote the initial marking $M(0)$ by $M_0 \subseteq \mathcal{P}$. For any transition t, let $\pi(t) \subseteq \mathcal{P}$ be the set of *upstream places*, ie all p such that $p \rightarrow t$, and $\sigma(t) \subseteq \mathcal{P}$ denote the set of *downstream places*, ie all p such that $t \rightarrow p$. We similarly define $\pi(p)$ and $\sigma(p)$ as the *upstream transitions* and *downstream transitions* of a place p respectively.

The semantics of TPNs which we shall adopt here is as follows. A transition t_i is said to be *enabled* if every upstream place of t_i contains a token. When t_i becomes enabled, each upstream token is "reserved" and cannot contribute to the enabling of any other transition. After a time period of $\tau(i)$ elapses, tokens will be removed from all upstream places and a token will be put in all downstream places. A place is said to have *choice* if it has more than one downstream transition. A place is said to have *conflict* if it has more than one upstream transition. Let $\mathcal{P}_{ch}$ and $\mathcal{P}_{co}$ denote the set of places with choice and conflict respectively. We'll denote the number of places with conflict by M_{co}, and the number of places with choice by M_{ch}. Associated with each place having choice is a function which specifies which downstream transition that place will "route" its token to. Thus a place can only contribute to the enabling of one of its downstream transitions. The collection of all such routing functions constitutes a *policy*. We shall, for the moment, only consider *stationary* policies, namely those policies for which the routing function associated with each place in $\mathcal{P}_{ch}$ is constant for all time. In such a case, we can associate a TPN for each policy, where every place has at most one downstream transition. Such a TPN is called *choice free*. The description of the dater equations and the algebraic operations on those equations can be found in more detail in (Liew and White 2007).

The solution of the general equations obtained from the algebraic operation is complicated by the presence of non-causality. Causality in the (max,+) part of the equation can be resolved by use of the *-operator (Baccelli et al. 1992). However resolution of acausality involving (initially unmarked) places with conflict needs to be resolved in an iterative manner until all such places become causal. This occurs when every upstream transition has completed at least one firing. Space limitations preclude a more expansive description of this process. We would however remark that the complete (causal) dynamic evolution of the system is achievable once all such conflicts are resolved.

Converting a Temporal Plan to a TPN

Temporal planning domains are expressed in PDDL2.1 as described in (Fox and Long 2004). To apply our approach, we need to translate the planning problems into 1-safe TPN. This conversion is done so with the aid of an extension of Petrify, a translator described in (Hickmott et al. 2007). This translator takes the planning problems and establish 1-safety by replacing every planning operator (action) by a set of 1-safe operators which lead to 1-safe TPNs. It also eliminates negative preconditions which is a property that cannot be handled consistently using place-transition PNs. It can be shown that any valid partially ordered set of firings of transitions of the TPN leads to a feasible plan to the original planning problem and *vice versa*. 1-safety holds true when preconditions cannot be re-initialised by the fired actions.

Solution to the Dynamic Equations for a TPN

One aspect of our solution approach is to dynamically reduce the size of the state space (by which we mean the number of active transitions) as events evolve. We thus remove all inactive transitions from the state vector $y[k]$. We partition the remaining transitions into the subset of those initially enabled, and the rest. Those transitions which are initially enabled always lead to evolution equations which are causal. The remaining transitions may be associated with places with conflict, and in such a case, will lead to acausality which needs to be resolved. We have a greedy algorithm for conflict resolution which is guaranteed to lead to a correct solution in a finite number of steps. At each step, we also remove transitions which we infer will never fire subsequently. This is done with the aid of the *-operator referred to earlier. We can show that provided every transition has a strictly positive firing duration, the system will reach causality in a finite time. Computational complexity is of the order of N^4 at worst. However we point out that most TPNs resulting from choosing a policy for a planning problem lead to sparse matrices in the system equations, and thus considerable computational savings. We also note that many computational platforms support IEEE floating point standard arithmetic implemented in the hardware, so computations involving infinities can be performed most efficiently if these attributes can be exploited.

Our solution approach differs considerably from that presented in (Baccelli, Cohen, and Gaujal 1992) which is heavily concerned with the causal resolution of switching conflicts. We avoid this by separately generating a choice-free TPN for each policy. As we are concentrating on

stationary policies at present, this iteration (of choice-free nets) without switching functions gives us a causal solution. Hence, our solution is more efficient as compared to (Baccelli, Cohen, and Gaujal 1992).

Generating a Plan

For each policy, the procedure is used to generate the transition firing sequence epochs. In the PNs produced by `petrify`, there is a unique transition, called the goal transition with zero firing duration [1] Generally, we will continue the determination of the solution until the first firing of the goal transition. Under some policies, the goal may be unreachable, so we need to set a maximum plan length, which if exceeded, is taken to mean that the goal is unreachable in a reasonable time under that policy. In some cases, the plan will reach an absorbing state (the TPN becomes *deadlocked*) before the goal is reached. A policy which results in the goal being reached before the specified maximum time is called *admissible*.

This approach may certainly be applied to evaluate the makespan (total duration of the plan) under each admissible policy, and thus select an optimal plan from among these. We also make the observation that the above approach can be used to determine a full plan-space reachability graph for any TPN with each policy choice explicit in the graph.

Concluding Remarks and Future Work

Here we have described some of the temporal planning methods. In addition, we have introduced the application of Timed Petri nets to Temporal planning problems. As there is no uniform definition of concurrency in the temporal planning community, we defined what concurrency is in our proposed framework of using TPNs. We argue that this approach that we have found, is an efficient way of obtaining optimal solution for a given stationary policy. It is efficient as we are able to solve the planning problem for every given policy using an algebraic equation. This efficiency is obtained by reducing the size of the state space (the size of the active transitions) as events evolve. However, we do realise that this approach is too restrictive to real-world problems, as stationary policies are not optimal over all the possible policies. Hence we intend to use heuristic to improve policies generated. In addition, we are interested in extending this approach to non-stationary policy by adding policy switching.

References

Agha, G. 1985. Actors: A model of concurrent computation in distributed systems. *In Artificial Laboratory*.

Baccelli, F.; Cohen, G.; Olsder, G. J.; and Quadrat, J.-P. 1992. *Synchronization and Linearity - An Algebra for Discrete Event Systems*. Wiley.

Baccelli, F.; Cohen, G.; and Gaujal, B. 1992. Recursive equations and basic properties of timed petri nets. *Journal of Discrete Event Dynamical Systems* 4(1):415–439.

Bacchus, F., and Addy, M. 2001. Planning with resources and concurrency: A forward chaining approach. *In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *In Proceedings of International Joint Conference on Artificial Intelligence(IJCAI)*.

Chen, Y.; Hsu, C.; and Wah, B. 2006. Temporal planning using subgoal partitioning and resolution in sgplan. *Journal of Artificial Intelligence Research* 26.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? *In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Do, M. B., and Kambhampati, S. 2003. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research* 20:155–194.

Doherty, P., and Kvarnstrom, J. 2000. Talplaner: A temporal logic based planner. *Invited Submission to AI magazine*.

Fox, M., and Long, D. 2004. Pddl2.1: An extension to pddl for expressing temporal planning domains. *In Proceedings of International Conference on Automated Planning and Scheduling(ICAPS-03)*.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. *In Proceedings of European Conference on Planning*.

Hickmott, S. L.; Rintanen, J.; Thiébaux, S.; and White, L. B. 2007. Planning using petri net unfoldings. In *Proc. Int. Joint Conference on Artificial Intelligence*, 1904–1911.

Hickmott, S. L. 2008. *Directed Unfolding: Reachability Analysis of Concurrent Systems and Applications to Automated Planning*. Ph.D. Dissertation, School of Electrical and Electronic Engineering, The University of Adelaide.

Liew, M., and White, L. B. 2007. Concurrent temporal planning using timed petri nets - policy evaluation. *In Proceedings of Australasia Joint Conferencen on Artificial Intelligence*.

Little, I.; Aberdeen, D.; and Thiébaux, S. 2005. Prottle: A prbabilistic temporal planner. In *Proc. AAAI*.

Little, I., and Thiébaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. *In Proceedings of International Conference on Automated Planning and Scheduling(ICAPS-06)*.

Long, D., and Fox, M. 2003. Exploiting a graphplan framework in temporal planning. *In Proceedings of International Conference on Automated Planning and Scheduling(ICAPS-03)*.

Mausam, and Weld, D. S. 2005. Concurrent probabilistic temporal planning. *In Proceedings of International Conference on Automated Planning and Scheduling(ICAPS-05)*.

McMillan, K. L. 1995. A technique of state space search based on unfolding. *Formal Methods in System Design* 6.

[1] This can be validly assumed without violating a similar condition which should apply to all other transitions.

Market-based Risk Allocation for Multi-agent Systems *

Masahiro Ono and **Brian C. Williams**
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139
hiro_ono@mit.edu, williams@mit.edu

Abstract

This paper proposes *Market-based Iterative Risk Allocation* (*MIRA*), a new market-based distributed planning algorithm for multi-agent systems under uncertainty. In large coordination problems, from power grid management to multi-vehicle missions, multiple agents act collectively in order to optimize the performance of the system, while satisfying mission constraints. These optimal plans are particularly susceptible to risk when uncertainty is introduced. We present a distributed planning algorithm that minimizes the system cost while ensuring that the probability of violating mission constraints is below a user-specified level.

We build upon the paradigm of *risk allocation* (Ono & Williams 2008), in which the planner optimizes not only the sequence of actions, but also its allocation of risk among each constraint at each time step. We extend the concept of risk allocation to multi-agent systems by highlighting risk as a commodity that is traded in a computational market. The equilibrium price of risk that balances the supply and demand is found by an iterative price adjustment process called *tâtonnement* (also known as *Walrasian auction*). Our work is distinct from the classical tâtonnement approach in that we use Brent's method to provide fast guaranteed convergence to the equilibrium price. The simulation results demonstrate the efficiency of the proposed distributed planner.

Introduction

Motivation

There is an increasing need for multi-agent systems that perform optimal planning under uncertainty. An example is planning and control of power grid systems. A power grid consists of a numbers of generators and electric transformers whose control should be carefully planned in order to maximize efficiency. A significant issue in power grid planning is the uncertainty in demand for energy by consumers. As the use of renewable energy, such as solar and wind power, become more popular, uncertainty in supply increases due to weather conditions. Another example is the Autonomous Ocean Sampling Network (AOSN), which consists of multiple automated underwater vehicles (AUVs), robotic buoys, and aerial vehicles. AOSN should maximize science gain while being exposed to external disturbances, such as tides and currents.

In order to deploy AI planning algorithms on such systems, we need a robust plan execution capability. Robust execution often involves 1) handling continuous system dynamics, 2) handling uncertainty in the environment, 3) operating the system at a risk level that the user find acceptable, and 4) scaling to multi-agent system.

To address the four problems, we developed *Market-based Iterative Risk Allocation* (MIRA), a multi-agent optimal planning algorithm that operates within user-specified risk bounds. MIRA optimally allocates risk among agents, and computes optimal control sequence for each agent in a distributed manner.

Approach

Planning under uncertainty, and risk allocation When planning actions under uncertainty, there is always a risk of failure that should be avoided. However, in many cases, performance can be improved only by taking extra risk. For example, we can reach a destination faster by driving a car at a faster speed and accepting a higher risk of an accident.

Without taking any risk, nothing can be done; however, no one dares to take unlimited risk. In many cases, people want to maximize performance, but with an upper-bound on the risk they take (*chance constraint*). For example, a race car driver would like to drive as fast as possible while limiting the probability of a crash to 0.1%. Therefore, we formulate the stochastic planning problem as an optimization problem with a chance constraint. With this formulation, (Ono & Williams 2008) showed that the planner should plan not only the sequence of actions but also the *risk allocation* in order to maximize the performance under a risk bound.

The example shown in Figure 1 illustrates the concept of risk allocation. A race car driver wants to plan a path to get to the goal as fast as possible. However, crashing into the wall leads to a fatal accident, so he wants to limit the probability of a crash to 0.1%. An intelligent driver would plan a path as shown in Figure 1, which runs mostly in the middle of the straightaway, but gets close to the wall at the corner. This is because taking a risk (i.e. approaching the

*This research is funded by The Boeing Company grant MIT-BA-GTA-1
Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

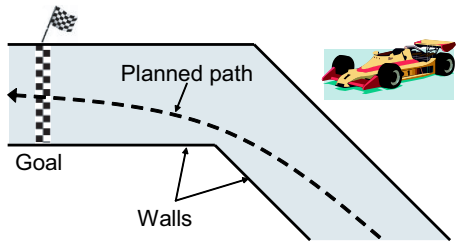


Figure 1: Risk allocation in a race car path planning scenario.

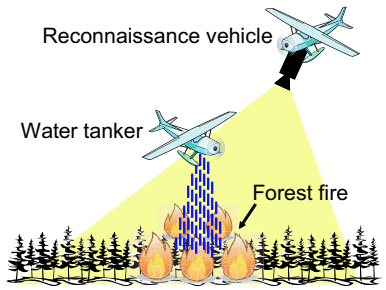


Figure 2: Multi-UAV fire-fighting scenario.

wall) at the corner results in a greater time saving than taking the same risk along the straightaway; in other words, the utility of taking risk is greater at the corner than the straightaway. Therefore the optimal plan allocates a large portion of risk to the corner, while allocating little to the straightaway. As illustrated by this example, *risk allocation* needs to be optimized across the constraints, in order to maximize the performance.

Distributed risk allocation for multi-agent system The concept of risk allocation can be naturally extended to multi-agent systems. Figure 2 shows an example of a multi-agent system with two unmanned air vehicles (UAVs), whose mission is to extinguish a forest fire. A water tanker drops water while a reconnaissance vehicle monitors the fire with its sensors. The loss of either vehicle results in a failure of the mission. Two vehicles are required to extinguish the fire as efficiently as possible, while limiting the probability of mission failure to a given risk bound, say, 0.1%. The water tanker can improve efficiency by flying at a lower altitude, but it involves risk. The reconnaissance vehicle can also improve the data resolution by flying low, but the improvement of efficiency is not as great as the water tanker. In such a case the optimal risk allocation is to allow the water tanker to take a large portion of risk by flying low, while keeping the reconnaissance vehicle at a high altitude to avoid risk. This is because the utility of taking risk (i.e. flying low) is greater for the water vehicle than for the reconnaissance vehicle.

Then, the question is how to find the optimal risk allocation between multiple vehicles in a distributed manner.

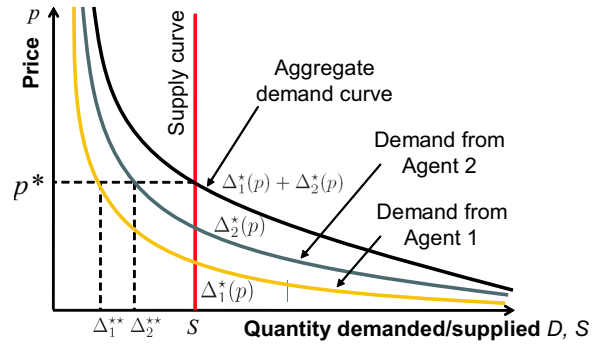


Figure 3: Demand and supply curves of the risk market with two agents. Note that we followed the economics convention of placing the price (independent variable) on the vertical axis.

Tâtonnement: market-based risk allocation Our approach is to use the market-based mechanism to optimize the risk allocation between agents. In a computational competitive market, each agent demands risk in order to improve its own performance. However, it cannot take risk for free; it has to purchase it from the market at a given price.

Agents are price takers. Therefore the demand is a function of the price of risk (*demand curve*). The higher the price is, the less each agent demands. Each agent has a different demand curve according to its sensitivity to risk. On the other hand, the supply of risk is constant, since the upper-bound of total risk is given.

Figure 3 gives the graphical interpretation of the market-based risk allocation in a system with two agents, who have different demand curves. The aggregate demand curve is obtained by adding the two demand curves horizontally. The supply curve is a vertical line since it is constant. The equilibrium price $p^\star$ lies at the intersection of the aggregate demand curve and the supply curve. It can be proven that the optimal risk allocation for the two agents are their demands for the risk at the equilibrium price ($_1^{\star\star}$ and $_2^{\star\star}$ in Figure 3).

However, in practice, the demand curve is not available as a function. Therefore it is not easy to find the intersection of the demand curve and supply curve. Instead, the equilibrium price is found by the following intuitive iterative process, called *tâtonnement* or *Walrasian auction* (Tuinstra 2000):

- Increase the price if aggregate demand exceeds supply,
- Decrease the price if supply exceeds aggregate demand,
- Repeat until supply and demand are balanced.

Related Work

Market-based approach has recently been recognized as an effective tool for decentralized multi-agent systems in AI community (Wellman 1993)(MacKie-Mason *et al.* 2004). Although tâtonnement has drawn less attention than auctions, it has been successfully applied to various problems such as the distribution of heating energy in an office building (Voos 2006), and resource allocation in communication networks (Kelly, Maulloo, & Tan 1998). The convergence of

tâtonnement has been an issue in economics for a long time; with a simple linear price update rule, it can only be guaranteed under a quite restrictive condition (Tuinstra 2000). We solved this problem by applying a root-finding method called Brent's method(Atkinson 1989).

Derivation

Risk Allocation The concept of risk allocation is derived from Boole's inequality:

$$\Pr\left[\bigcup_i F_i\right] \leq \sum_i \Pr\left[F_i\right] \tag{1}$$

Assume that F_i in the above inequality represents the event that the ith agent fails. Then the left hand side means the probability that at least one agent in the system fails (i.e. system failure). It is upper-bounded by the right hand side, which is the sum of the individual probabilities that each agent fails.

The user of the system limits the probability of system failure to S. This constraint is called *joint chance constraint*.

$$\Pr\left[\bigcup_i F_i\right] \leq S \tag{2}$$

Using Boole's inequality Eq.(1), it can be easily shown that the following condition is the sufficient condition of the original joint chance constraint Eq.(2).

$$\forall_i \quad \Pr\left[F_i\right] \quad \leq \quad {}_i \tag{3}$$

$$\wedge \quad \sum_i \quad {}_i \quad \leq \quad S \tag{4}$$

Eq.(3) constrains the probability that each individual agent fails (*individual chance constraints*). Eq.(4) states that the sum of the risk bounds of all individual chance constraints must not exceed the risk bound of the original joint chance constraint S. Here, the analogue to the resource allocation is found; S is the total amount of resource (i.e. risk), which is distributed to agents in the system; ${}_i$ is the amount of resource allocated to the ith agent.

Once the risk is allocated to each agent, a joint chance constraint over multiple agent Eq.(2) is decomposed into individual chance constraints over individual agents Eq.(3).

Distributed Optimization of Risk Allocation The objective of our problem is to minimize the system cost, which is the total of the cost of all agents in the system, while limiting the probability of system failure (joint chance constraint). As explained above, the joint chance constraint Eq.(2) is implied by the individual chance constraints Eq.(3) and the total risk inequality Eq.(4). Therefore, our optimization is formulated as follows:

$$\min_{\Delta_{1:N}} \quad \sum_{i=1}^{N} J_i(\quad {}_i) \tag{5}$$

$$\text{s t} \quad (3)(4)$$

where J_i is the cost of ith agent, and N is the number of agents in the system. We assume that there is no coupling between agents through constraints. This formulation describes a centralized algorithm since the risk allocations of all agents are planned in one optimization problem. We omit the plant model (as linear constraints) and control limit constraints to keep the equations simple. See (Ono & Williams 2009) for the formulation with all constraints.

Solving the centralized optimization problem Eq.(5)(3)(4) is equivalent to solving the following N unconstrained optimization problems, since two formulations have the same Karush-Kuhn-Tucker (KKT) conditions for optimality.

$$\min_{\Delta_i} J_i(\quad {}_i) + p \quad {}_i \qquad (\text{for } \ i = 1 \cdots N) \tag{6}$$

where $p \geq 0$ is the Lagrange multiplier. In order to be optimal, p and ${}_i$ must satisfy the following condition:

$$p\left(\sum_i \quad {}_i \quad S\right) = 0 \tag{7}$$

Since the optimization problems Eq.(6) contains only the variables related to the ith agent, it can be solved by each agent in a distributed manner.

Economic Interpretation The interpretation of these mathematical manipulations becomes clear by regarding the Lagrange multiplier p as the *price of risk*. Each agent can improve the performance by taking risk ${}_i$, but not for free. Note that a new term $p \quad {}_i$ is added to the cost function Eq.(6). This is what the agent has to pay to take the amount of risk ${}_i$. Given the price p, each agent computes the optimal *demand for risk* ${}^\star_i(p)$ by solving the optimization problem Eq.(6). The total amount of risk S can be interpreted as the *supply of risk*, which is given by the user.

In order to minimize the system cost, the price p must satisfy the condition Eq.(7). Such a price $p^\star$ is called the *equilibrium price*. The demand for risk of each agent at the equilibrium price ${}^\star_i(p^\star)$ is the optimal risk allocation for the agent.

Eq.(7) illustrates the relation between the equilibrium price $p^\star$, optimal demand ${}^\star_i(p^\star)$, and supply S; in the usual case where the equilibrium price is positive $p^\star > 0$, the aggregate demand $\sum_i \quad {}^\star_i(p^\star)$ must be equal to the supply S, as illustrated in Figure 3; in a special case where the supply always exceeds the demand for all $p \geq 0$, the optimal price is zero $p^\star = 0$. If the aggregate demand always exceeds the supply for all $p \geq 0$, there is no solution that satisfies the constraint Eq.(4), and hence the problem is infeasible.

Finding Equilibrium Price According to Eq.(7), the equilibrium price is the root of the following equation:

$$\sum_i \quad {}^\star_i(p) \quad S = 0 \tag{8}$$

The classical approach in economics is to iteratively adjust the price with the increment that is proportional to the excess demand $\sum_i \quad {}^\star_i(p) \quad S$, until the price converges.

However, the convergence is guaranteed only under a strong condition called gross substitutability(Tuinstra 2000). The slow convergence is also an issue.

Our breakthrough is to use a root-finding algorithm called Brent's method. It is guaranteed to convergence at a superliner convergence rate, by combining three methods: the bisection method, the secant method, and the inverse quadratic interpolation (Atkinson 1989). The only conditions for convergence is the continuity of the aggregate demand curve, which typically holds. As far as we know, the use of Brent's method for tâtonnement has not been discussed before. This is probably because adjusting price with such a complex method is not a natural model of the real-world economy. Nonetheless, this limitation is not relevant to our computational economy, since our objective is to obtain the optimal plan, not to model the real-world economy.

The Algorithm

The entire algorithm is summarized below. We call this algorithm as Market-based Iterative Risk Allocation (MIRA).

1. Sets the initial price p and announce it to all agents.
2. Each agent computes its optimal demand at the price $^{\star}_{i}(p)$ by solving Eq.(6), and bids it.
3. Terminate the algorithm if the aggregate demand is equal to the supply; otherwise, adjust the price p by computing one step of Brent's method, and announce it.
4. Go to Step 2.

The price converges to the equilibrium price $p^{\star}$ in this iterative process. The optimal risk allocation for each agent is its demand at the equilibrium price $^{\star}_{i}(p^{\star})$.

Simulation

Simulations were conducted on a machine with Intel(R) Core(TM) i7 CPU clocked at 2.67 GHz and 8GB RAM. See (Ono & Williams 2009) for the used parameters.

To evaluate the efficiency of MIRA algorithm, the computation time of the following three methods were compared:

1. Centralized optimization,
2. Distributed optimization (tâtonnement) with a linear price increment, and
3. MIRA: distributed optimization (tâtonnement) with Brent's method.

Table 1 shows the results. The three algorithms were tested with different problem sizes - two, four, and eight agents. Each algorithm was run 10 times for each problem size with randomly generated constraints. The average running time is shown in the table. The computation of the distributed algorithms was conducted parallelly. Communication delay is not included in the result.

The computation time of the centralized optimization algorithm quickly grows as the problem size increases. Distributed optimization with a linear price increment is even slower than the centralized algorithm.

MIRA, the proposed algorithm, outperforms the other two for all problem sizes. The advantage of MIRA becomes clearer as the problem size increases. More simulation result is presented in (Ono & Williams 2009).

Table 1: Comparison of the computation time of three optimization algorithms.

Number of agents	Computation time [sec]		
	Centralized	Distributed (linear increment)	MIRA
2	13.9	80.6	6.4
4	63.8	540.5	18.1
8	318.5	797.8	37.5

Conclusion

We have developed Market-based Iterative Risk Allocation (MIRA), a multi-agent optimal planning algorithm that operates within user-specified risk bounds. The three key innovations that enabled MIRA were:

1. Extension of the concept of risk allocation to multi-agent system.
2. Derivation of distributed optimization method for multi-agent risk allocation using KKT conditions.
3. Introduction of Brent's method to tâtonnement as a price update rule.

The simulation result showed that MIRA achieved substantial speed-up compared to centralized optimization approach, particularly in a large problem.

Acknowledgments

Thanks to Michael Kerstetter, Scott Smith and the Boeing Company for their support.

References

Atkinson, K. E. 1989. *An Introduction to Numerical Analysis, Second Edition*. John Wiley & Sons.

Kelly, F.; Maulloo, A.; and Tan, D. 1998. Rate control in communication networks: Shadow prices, proportional fairness and stability. *J. of the Operational Research Society* 49:237–252.

MacKie-Mason, J. K.; Osepayshvili, A.; Reeves, D. M.; and Wellman, M. P. 2004. Price prediction strategies for market-based scheduling. In *Proc. of ICAPS-04*.

Ono, M., and Williams, B. C. 2008. An efficient motion planning algorithm for stochastic dynamic systems with constraints on probability of failure. In *Proc. of AAAI-08*.

Ono, M., and Williams, B. C. 2009. Risk allocation for multi-agent systems using tâtonnement. Technical report, MIT CSAIL.

Tuinstra, J., ed. 2000. *Price Dynamics in Equilibrium Models: The Search for Equilibrium and the Emergence of Endogenous Fluctuations*.

Voos, H. 2006. Agent-based distributed resource allocation in technical dynamic systems. In *Proc. of IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications (DIS 2006)*.

Wellman, M. P. 1993. A market-oriented programming environment and its application to distributed multicommodity flow problems. *J. of Artificial Intelligence Research*.

Hierarchical Skill Learning for High-Level Planning

James MacGlashan
jmac1@cs.umbc.edu
University of Maryland, Baltimore County

Marie desJardins
mariedj@cs.umbc.edu
University of Maryland, Baltimore County

Introduction

Many of the existing techniques for controlling goal-directed agent behavior fall into two primary approaches: heuristic-search planning (HSP) and reinforcement learning (RL). Each has its advantages and disadvantages. For instance, heuristic-search planning does not traditionally learn from previous experience, and can only be applied in domains for which a complete domain model exists. On the other hand, reinforcement learning often performs poorly in new situations until it has gained enough experience to learn an effective policy, and it is difficult to scale RL up to large, complex domains.

Both RL and HSP also face difficulties in domains that require long action sequences. Heuristic-search state-space planning is intractable in such domains, because of the very large search spaces, and reinforcement learning may require exponentially many execution traces to converge.

In order to address both the unique and the shared problems of HSP and RL, we propose a new research direction called *skill bootstrapping* (SB). The goal of SB is to provide an integrated learning and planning architecture that can improve its performance over time in complex domains. An SB agent starts with a basic set of primitive actions (and their preconditions and effects) as its model of the world. Over the course of solving numerous problems by applying HSP to the primitive actions, SB identifies recurring subgoals, for which it uses RL to create *skills* that can be applied within the HSP process to solve these subgoals more efficiently. The skills behave as partial policies that can be used reactively, without lengthy deliberative reasoning.

Once a new skill is learned, it becomes available for use by the planner along with the other primitive actions, allowing for more compact plans. Additionally, just as future plans can use learned skills, future skills may be built upon lower-level skills. Over the course of the agent's experience, this will eventually result in a *hierarchy* of skills that support high-level reasoning.

The proposed SB approach is still in its very early stages, so all of the design issues have not been solved. We present here our preliminary ideas about how to create SB; we use the present tense throughout the paper, but wish to emphasize that this is proposed research, for which no implementation yet exists. We will detail the architecture of the proposed SB framework, and describe how SB relates to other current research.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

SB Approach

The SB architecture consists of three primary components: *planning*, *memory*, and *skill learning* (Figure 1).

The heuristic-search planning component takes as input the current goal to be achieved and a list of available actions. In a newly created SB agent, the planner simply uses a heuristic to guide a state-space search for the goal. The resulting plan is sent to the memory component, where the plan is stored and indexed in a *plan library*. A *skill identifier* monitors the plan library to find common *types* of goals for which large numbers of plans have been indexed. We say common *type* because goal states need not be identical, but must only share similar properties. For instance, picking up block A is not the same goal as picking up block B, but it is the same *type* of goal. These goal similarities can be determined using the organization of the plan library, similar to techniques used by case-based planning algorithms such as CHEF (Hammond 1989).

Once a common goal type with sufficient relevant plans has been detected, a new skill is created to achieve that goal. *Skill learning* uses the traces stored in the plan library to create a policy for the skill. In other words, each plan retrieved from the plan library is rerun and the Q-values of the state-action pairs (the utility for taking an action given a state) in the plan are learned by the RL process as described later. (The reward value received at each state is set to 0 for all non-goal states and 1 for the goal state.)

Since learned skills become part of plans that are stored in the plan library, over time, skills will be developed that utilize other skills. This property is desirable because the resulting hierarchies of skills permit efficient planning and execution without a lengthy deliberative reasoning process.

The execution process, and possible replanning that may occur if a skill fails, are discussed next.

Plan execution and replanning

After producing a plan, the sequence of actions is executed by the execution modules. Primitive actions are simply applied. Skills, however, require some execution monitoring,

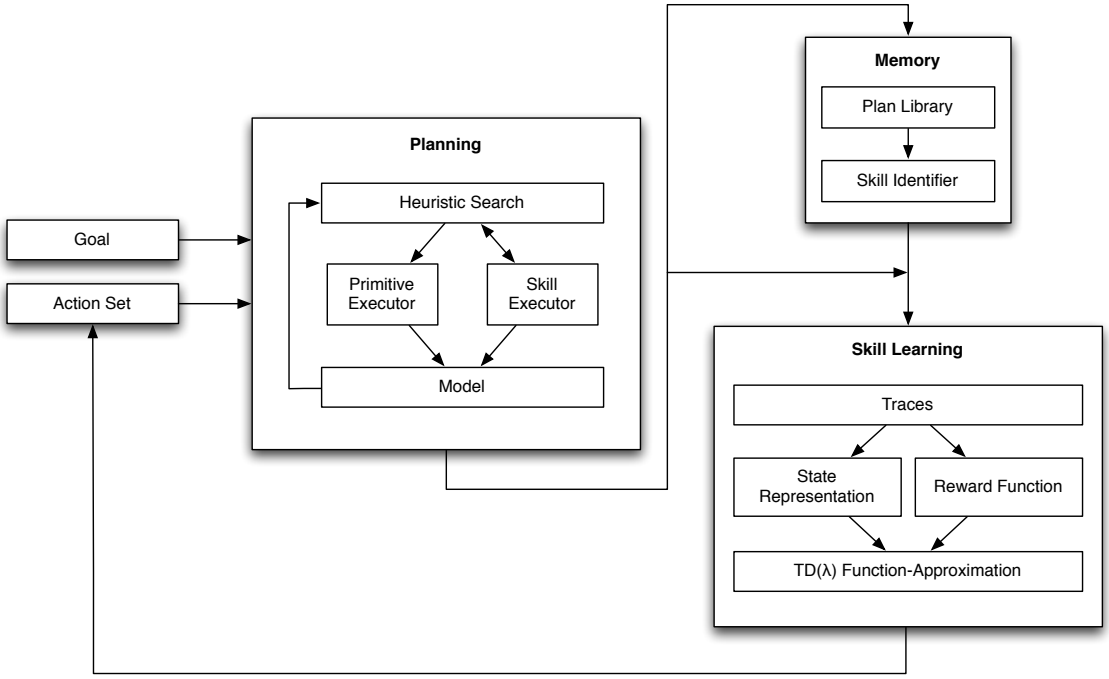


Figure 1: The SB architecture has three primary components: *Planning*, *Memory*, and *Skill Learning*. The planning component takes a set of actions and a goal, and uses heuristic search to find a plan that achieves the goal. Successful plans are stored in a *plan library* which is monitored by a *skill identifier* to find sets of plans that share common types of goals. These sets of plans are used to learn a parametrized skill that accomplishes that goal type. Learned skills augment the action set, and can then be used by the heuristic search planner. This process is applied repeatedly, resulting in a hierarchy of skills.

since their outcomes are not known with complete certainty. Specifically, when a skill is executed from a given state (real, or in the search's model), an ϵ-greedy policy is used. The ϵ-greedy policy is followed until the subgoal associated with the skill has been achieved, at which point control is returned to the top-level planning and execution process. RL updating is applied to the skill, so that performance can be improved even after a skill is first learned.

If at any point a skill becomes "stuck" and cannot reach its goal state, then the skill is terminated and the planner replans the current subgoal from the current state. Such execution failures may be detected either through cycles in the execution, unusually long execution, or perhaps by associating certainty values with each skill that estimate how often each state has been explored in the past.

Skill learning

Planning complex problems can be computationally expensive. In such cases, we would like to restrict planning to an abstract level that is easier to solve. The purpose of a skill is to facilitate this abstraction by learning how to reactively accomplish a *type* of goal, independent of the specific problem formulation. Additionally, it is useful to invoke a skill multiple times within the same problem, but parameterized to different contexts. The planner would then be able to focus on the outcomes of skills, and skills would provide the reactive mapping to achieve the outcomes. To provide this capability, a parameterized representation of the world state must be provided as part of the input to the skill learning algorithm. This parameterized representation would be used both when a skill is initially learned and when it is invoked by future plans.

Consider the previous example of forming a skill for picking up a block. In this case, we may use plans that resulted in picking up Block A and plans that resulted in picking up Block B. In order for the skill to be parameterized—that is, applicable to any kind of block—the world state representation for this particular skill must explicitly indicate which block in the world is the target block. Furthermore, each such object may have relevant attributes (such as its position and size) that should be used when applying the skill. Therefore, a state representation is created for each skill using an appropriate vector format that includes any such parameters.

Even with a skill-specific representation of the world state, the skill still needs to be invariant across all possible goals. In the pickup-block example, there may be a designated position in the state vector that represents the target block's position and size, so the learning must capture a policy that respects differences in position and size. Traditional look-up table RL techniques would quickly become insufficient in this context. Saving an entry for every possible state that may be seen in a domain would be intractable,

in terms of both memory usage and learning time. Instead, skill learning is performed with a TD(λ) function approximation approach (Sutton 1988) that allows for compact storage of skills. Function approximation also enables the learning process to generalize to multiple states, which is necessary to allow the skill to perform well on variants of the goal type that have not previously been explored.

There are a number of different techniques for function approximation with TD(λ). Two methods we are currently exploring are locally weighted linear regression (LWLR) techniques, such as those used in the Hedger algorithm (Smart and Kaelbling 2000), and artificial neural networks (ANN). LWLR function approximation allows for fast non-linear value function approximation, and is guaranteed to converge when values are not extrapolated. A potential drawback of LWLR is that it traditionally stores every data point explored, which may prove to be too expensive. To avoid this, a mechanism to store only crucial points may be needed.

ANNs provide compact storage of the function, and can also approximate non-linear value functions. Historically, ANNs were not guaranteed to converge with off-policy TD(λ), and algorithms that did guarantee convergence were either computationally complex, or required a large number of episodes to learn. Recently, Sutton et al. have proposed a new algorithm for gradient-based function approximation that is guaranteed to converge and learns quickly (2009). While these new advances may make ANNs viable, a drawback of using ANNs is that it is unclear how many hidden nodes would be required to capture the expressiveness of the value function. For any given domain, the number of hidden nodes could be specified in advance by the designer, but this approach may be infeasible in complex or poorly understood domains. LWLR techniques, on the other hand, can dynamically control the expressivity of the representation by adjusting the kernel bandwidth and the number of stored data points.

Traditionally, RL algorithms can require a very large number of trials to converge to a good policy, and will perform very poorly in the early learning stages. However, we would like to learn skills without having to exhaustively iterate through many trials. RL techniques may seem to be insufficient for these demands. However, because every skill in SB is learned *after* a number of source plans have been produced, the skill's learning time can be reduced by seeding the process with the planning traces. A similar approach is taken by the Hedger RL algorithm which seeds learning with example traces provided by a human (Smart and Kaelbling 2001). By combining HSP and RL in this way, we eliminate the poor early performance exhibited by RL, at the cost of incurring the additional deliberation time required for planning.

Skill-driven goal selection

The SB architecture assumes that the agent will be tasked with numerous goals to achieve over the course of its life, and that through these experiences, a hierarchy can be built that allows the agent to scale to even more complex tasks. Ideally, there would be meaningful tasks that are tractably solvable by the agent, even when only the primitive actions available. This would allow the agent to perform meaningful work from its creation, while also being able to perform more complex activities later in its life.

In some domains, however, the first meaningful tasks may be intractable to plan using primitive actions alone. One way to solve this problem would be to provide teacher guidance (from a human, or perhaps even from another agent) during the early stages of the agent's existence. The teacher could provide progressively more complex tasks for the agent that were designed to give the agent the needed experience to solve the desired higher-level tasks. This approach places some of the burden of structure implementation back onto the designer, which of course is what learning is intended to avoid. However, a teacher who provides progressively more complex tasks is providing only partial knowledge about the domain, at a level of abstraction that is relatively natural to express. The agent still provides most of the knowledge modeling through its learning process: discovering the relationship of the skills to each other, determining the necessary subskills for each skill, and identifying any unnecessary subskills.

To further automate the process of skill development and remove the need for a teacher, an agent can proactively generate its own set of goals. To choose these goals, the agent can begin by random walking from its current state. The random walk can be augmented to choose less frequently used actions when they become possible. When a unique (statistically unusual) change in a state space feature occurs after some sequence of random walking, the agent can identify this as a goal state of interest. For example, if randomly acting in the world resulted in a box being in an agent's hands, this could be identified as a goal of interest. The agent can then enter a trial phase where it again randomly walks, and continues to do so for some time after the goal state is no longer valid. The agent can then specify the same type of goal for it to solve, and the SB architecture continues as described. The agent can repeatably try variants on the goal of interest, using random walking to reset into new start states before each trial. As skills are developed, this random walking can quickly take the agent to goal shifts that require much longer action sequences, facilitating the development of higher-level skills.

Related Work

The concept of policy control that builds on lower-level action primitives to achieve a goal is not new to agent control. In planning fields, this notion is usually referred to as *macro-operators* or *macro-actions*. Macro-actions are generally constructed as a fixed sequence of primitive actions. Botea et al. (2005) present an algorithm called Macro-FF that examines a planning domain for potential sequences of actions to create macro-actions, and then filters that list based on heuristics and experience in training problems. Newton et al. (2005) use genetic algorithms on training problems to determine useful sequences of actions for use as macro-actions in planning. Marvin (Coles and Smith 2007) is a learning algorithm that uses macro-actions to escape heuristic plateaus. Coles et al. (2007) extended Marvin to allow

macro-actions learned in previous problems of the same domain to be applied to solve future problems. This large collection of macro-actions is stored in a macro-action library that is managed and pruned.

Two key commonalities of these approaches are that macro-actions are a fixed sequence of actions, and that the list must be pruned to avoid large collections of macro-actions. The SB approach proposed here differs in that skills are *not* fixed sequences of actions. Instead, they are policy control mechanisms that vary the action sequence depending on the particular state of the world. Because skills can vary their action sequence depending on the situation, a single skill could effectively represent a *collection* of macro-actions as one succinct unit.

Marthi et al. propose a hierarchical planner, AHA* (2008) using high-level actions (HLAs) to specify partial plans that are refined in the planning process. Planning can take place at a high level by constructing abstract lookahead trees. HLAs may result in different primitive action sequences, depending on the specific problem, unlike the macro-operators described above. A key difference from the SB approach is that SB creates plans using skills — reactive learned policies for solving sub goals — whereas AHA* must plan at every step. Further, AHA* requires a designer to specify the full HLA hierarchy structure, whereas SB can learn its skill hierarchies through experience, by partial design from a teacher, or in an automated exploratory manner.

Using hierarchies of actions in reinforcement learning has also been an area of active research. The MaxQ algorithm (Dietterich 2000) used a designed hierarchy of subtasks to efficiently solve more complex problems. These subtasks are often referred to as *temporally extended actions*. More recent work has focused on automatically identifying the action hierarchy. Jonsson and Barto (2005) presented the VISA algorithm, which uses a Dynamic Bayesian Network (DBN) to assist in construction of the action hierarchy. The HI-MAT algorithm (Mehta et al. 2008) is similar to the VISA algorithm, but couples a DBN with a successful trajectory of a source reinforcement learning problem to determine the hierarchy.

With these algorithms, the action hierarchies are fixed structures that have a defined root structure and are specific to a single problem. With the SB architecture, skills form hierarchies, but are not fixed in structure and can grow over time. Additionally, skills learned in the SB architecture are not explicitly structured. That is to say, skills that are referenced by parent's skills do not have to be invoked by the parent. Rather, any skill can be independently invoked if it is pertinent to the problem at hand. This also means that skills can be shared among different problems, and their structure does not have to be relearned.

Other work on forming action abstraction comes from Simsek and Barto (2007). They use the same terminology of *skills* to represent policies that achieve some subgoals in a task. To identify skills, they examine the graph structure of reinforcement learning problems to identify states that are likely to be important in the problem. The SB architecture differs in that instead of explicit states being used to identify places for skills, *types* of goals are identified that may result in different states, but states that share similar properties and may be parameterized. SB also differs in that skills can be hierarchical, building on the abilities of lower-level skills.

References

Botea, A.; Enzenberger, M.; Muller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

Coles, A., and Smith, A. 2007. MARVIN: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research* 28:119–156.

Coles, A.; Fox, M.; and Smith, A. 2007. Online identification of useful macro-actions for planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.

Hammond, K. 1989. *Case-based planning: Viewing planning as a memory task*. Academic Press Professional, Inc. San Diego, CA, USA.

Jonsson, A., and Barto, A. 2005. A causal approach to hierarchical decomposition of factored MDPs. In *ICML '05: Proceedings of the 22nd International Conference on Machine Learning*, 401–408. New York, NY, USA: ACM.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic hierarchical planning: Optimal and online algorithms. Technical Report UCB/EECS-2008-150, EECS Department, University of California, Berkeley.

Mehta, N.; Ray, S.; Tadepalli, P.; and Dietterich, T. 2008. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the 25th International Conference on Machine Learning*, 648–655. ACM New York, NY, USA.

Newton, M.; Levine, J.; and Fox, M. 2005. Genetically evolved macro-actions in AI planning problems. *Proceedings of the 24th UK Planning and Scheduling SIG* 163–172.

Simsek, O., and Barto, A. 2007. Betweenness centrality as a basis for forming skills. Technical report, University of Massachusetts, Department of Computer Science.

Smart, W., and Kaelbling, L. 2000. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning*.

Smart, W., and Kaelbling, L. 2001. Reinforcement learning for robot control. *Mobile Robots XVI (Proc. SPIE 4573)*.

Sutton, R.; Maei, H.; Precup, D.; Bhatnager, S.; Silver, D.; Szepesvari, C.; and Wiewiora, E. 2009. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th International Conference on Machine Learning*.

Sutton, R. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3(1):9–44.

An Investigation into Using Object Constraints to Synthesize Planning Domain Models

M.M.S.Shah and **T. L. McCluskey** and **M.M.West**
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
email s.shah@hud.ac.uk

Abstract

This thesis work concerns the area of automated acquisition of planning domain models from one or more examples of plans within the domain under study. It assumes that an adequate domain model for a domain can be composed of objects arranged in collections called object sorts. Recently, two systems have had success in using this underlying assumption: the Opmaker2 system (McCluskey et al. 2009), and the $LOCM$ system (Cresswell, McCluskey, and West 2009). The former requires only one solution plan as input, as long as it contains at least one instance of each operator schema to be synthesized. It does require a partial domain model as well as the example plan, and the initial and goal states of the plan. In contrast $LOCM$ requires no background information, but requires many instances of plans before it can synthesize domain models. Our aim is to build on these systems, and establish an experimental and theoretical basis for using object - centred assumptions to underlie the automated acquisition of planning domain models.

Introduction

This thesis work concerns the area of automated acquisition of full or partial domain models from one or more examples of plans within the domain under study. One motivation is that the knowledge engineering of such domain models by hand into languages such as PDDL is inefficient and laborious. Another is to help planning agents become more autonomous. Agents that have planning capabilities may need the ability to acquire and refine domain models, if they encounter new domains.

Our work assumes that an adequate domain model for a domain can be composed of objects arranged in collections called object sorts. We use the assumptions of Simpson et al's object-centric view of domain models (Simpson, Kitchin, and McCluskey 2007). Here a planning domain model consists of sorts of object instances, where each object behaves in the same way as any other object in its sort. Sorts have a defined set of states that their objects can occupy, and an object's state may change (called a state transition) as a result of a domain action's execution.

Recently, two systems have had success using this underlying assumption: the Opmaker2 system (McCluskey et al. 2009), and the $LOCM$ system (Cresswell, McCluskey, and West 2009). The former requires only one solution plan as input, as long as it contains at least one instance of each operator schema to be synthesized. It does require much background knowledge, however, including a partial domain model, as well as the training plan itself, and the initial and goal states of the plan. In contrast $LOCM$ requires no background information, but requires many training plans before it can synthesize domain models. We aim to build on these systems, and establish an experimental and theoretical basis for using object - centred assumptions to underlie the automated acquisition of planning domain models.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Related Work

Work in the general area of knowledge acquisition for plannig is growing, with current events such as ICKEPS 2009, and the ICAPS 2009 workshop in "Learning Structural Knowledge From Observations", both of which are very relevant to the thesis study. One can classify work by considering what inputs the systems require, and what their outputs are, as given in the list below. Let us assume a system "observes" training plans containing action instances, and tries to synthesize operator schema representing groups of these action instances. The features of the systems may include the following.

- Observations(s): Are many sequences of training plans required or just one? Are the plans assumed to be correct? How are they represented e.g. as an action name followed by a list of parameters representing objects affected?
- Training help: As well as the observed plan, is there other data supplied? e.g. a system might be supplied with the initial and goal state that the plan solves, as well as partial or full intermediate state information.
- Constraints: is there a background set of constraints assumed about the world which help in the synthesis stage? For example, in the object-centric approach we assume that objects are affected by actions, and an object is input to an action in the state that it leaves the previous action.
- Partial Model: is there a pre-defined specified language of predicates or fluents? e.g. most systems assume that the user has specified a set of fluents in which the world is described, thus constraining the system's range of applications.

- Algorithm: this refers to the characteristics of the algorithm used to synthesize operators: does it use deduction and/or induction, or hill climbing or other search techniques?
- Output: this is normally a set of operator schema or methods, but could also include a set of heuristics to be used to speed up planning. The expressiveness of the output schema language is important in determining where a system is capable of adequately synthesizing a domain model. One way of characterising this is to use the levels in PDDL as a form of measure.

There has been a great deal of related work in this area under the assumption that the output is a literal-based STRIPS-like domain model. In this work, it is generally the case that predicates or fluents are predefined, and the aim of the system is to use example plans, either incrementally or all at once, to induce or deduce operator schema in terms of these fluents. We will mention several recent systems. In the ARMS system (Wu, Yang, and Jiang 2005), a domain model is output in the form of STRIPS-type operator schema, with the input being multiple examples of training plans. ARMS does not require intermediate state information, but requires background knowledge such as types, relations,initial and goal states, and relies on many training plans containing valid solution sequences. In the $SLAF$ approach (Shahaf and Amir 2006) expressive operator schema can be output (at least to ADL level), but $SLAF$ requires as input specifications of fluents, as well as partial observations of intermediate states between action executions. Learning expressive theories from examples is a central goal in the Inductive Logic Programming (ILP) community, and some systems build on this considerable body of work. For example, in his thesis (Benson 1996), Benson describes an ILP method for learning very expressive operator schema, using multiple examples, in an incremental fashion.

There is also a body of research aimed at learning hierarchical domain models, mainly in the "HTN" variety. Practical planning domains are based on 'hierarchical task network' decomposition, and it could be claimed that these systems learn heuristics encapsulated within an HTN method. HTNs can be very difficult to construct manually and authors have worked in producing these using methods from machine learning. In (Nejati, Langley, and Konik 2006) the authors describe how they induce teleoreactive logic programs from expert traces. The teleoreactive programs index methods by the goals they achieve. They use methods derived from explanation based learning to chain backwards from the end result of the sample trace. Theoretical work on HTN planning is presented in (Ilghami et al. 2005). This paper introduces a formalism whereby situations are modelled where general information is available of tasks and subtasks, together with some plan traces. In the early work all information about methods was required except for the preconditions. This limitation is overcome in later work by the same group (Ilghami, Nau, and Munoz-Avila 2006) a new algorithm 'HDL' (HTN Domain Learner) is presented which learns HTN domain descriptions from plan traces. Between 70 and 200 plan traces are required to induce the descriptions. HTN-MAKER is presented in (Hogg and Munoz-Avila 2007). This receives as input a STRIPS domain model, a collection of STRIPS plans and task definitions and produces an HTN domain model. The experimental hypothesis is that after a few problems have been analysed an HTN domain model will be ultimately obtained able to solve most solvable problems. A version of the logistics-transportation domain is chosen for the experiment and good results are obtained. However these good results are not replicated for the blocks-world domain. One problem is the large number of methods which have to be learned, where one method might subsume another.

A motivating example

The Ring World

To motivate the work on synthesizing domain models from an object perspective, we use an example from computer games, called the "Ring World". The idea is, for each group of characters in the game, we need to create a domain model specifying the actions that they can take. After this, a planning engine can then be used to generate plans and create the illusion of goal directed behaviour. The game was designed as follows: there is one overall 'database' which can completely describe any state of the game. Different sets of characters have different sets of capabilities: one is a set of knights that attempt to bring about a set of goals (eg acquire the Ring, kill orcs). Another is a set of orcs, with goals such as blocking doors, disabling knights etc. Each set of characters can use deliberative planning to try to achieve their goals, execute their plans and replan when their plans are no longer executable. The player character within the game could, for example, play the role of a 'wizard' that observed one force's plans and attempted to help achieve them.

A step towards implementing the game might be to construct a planning domain model with operator schema representing the capabilities of the characters, and the effects of their actions. Sets of operator schema, representing the views of the overall domain model, have to be generated for this, one for each set of characters. These views would constitute distinct operator sets, whereas the object structures would be common to all and constitute the central database.

An example snapshot (or state) of the Ring World is illustrated in Figure 1. For example, from the point of view of the force controlling the knights:

- knights can move between rooms and passages. Movement will take place via propositional steps such as "move-into-room" and "move-to-passage";
- passages can be blocked by orcs or locked doors;
- locked doors need to be opened by the correct key;
- knights can acquire keys from trolls if they are near them and offer them some treasure in exchange;
- orcs that are sleeping can be killed by a knight if it is near them and has a weapon (however the weapon is then spent);
- orcs that are awake are similar to sleeping orcs except the knight needs to pick up protection first (a shield or a spell) before fighting them;

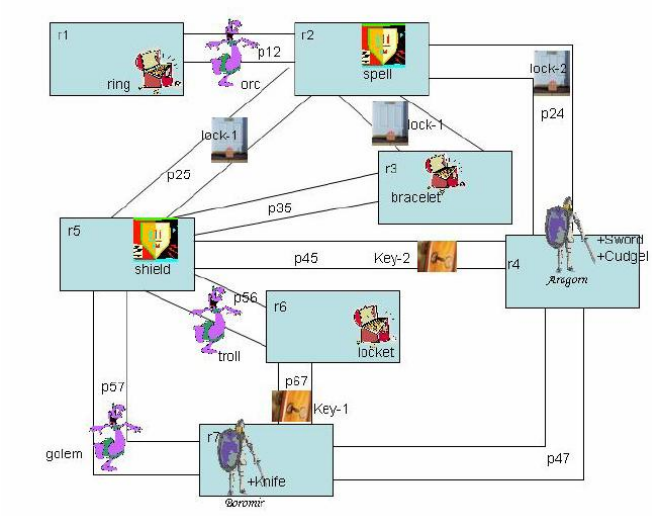


Figure 1: A state of the Ring World

- hidden treasure can found by a knight if it is in the same room as the treasure.

For example, consider the state in Figure 1. Assume that a goal to be achieved is for the knight "Aragorn" to acquire the ring. Then the planner would have to generate a plan for him to move through rooms and passages to acquire a key to unlock one of the passages leading to r2 (the only route to the ring in r1); and to acquire some treasure so that he can trade it for protection, so that he will be able to overcome the orc guarding room r1, when he eventually arrived there.

Use of Object Centred Constraints in Determining Object Transitions

We will use the Ring World example to illustrate the idea of how object constraints can help with the domain model synthesis process. In this case, we would like to use constraints to determine *the complete set of transitions of individual objects* as they are effected by actions in a training plan.

We assume that a correct training plan is input, with initial and goal state as training help. Additionally, assume that we have partial domain model that includes only a propositional encoding of the Ring World, with the treasure object necklace being described as in one of 4 states: hidden, held_by_knight, taken_by_troll, offered_for_barter. Assume a training plan of

"Find necklace, Offer necklace, Exchange necklace"

The initial state is "hidden", and the final state is taken_by_troll. As shown in Figure 2, there are 7 possible paths through the object state space, using the assumption that the object changes state at each step. In this case it would not be possible to trace a deterministic path through the space of propositions, from the training sequence, as the constraints on initial and goal states are insufficient.

If we introduce a relational, object representation, then this introduces extra constraints. Assume that any piece of object treasure (?t) can occupy one of four abstract states, where it might be related to a ring (?r), a knight (?k), or a troll (?m), as follows:

```
hidden(?t,?r)
holding(?t,?k)
taken(?t,?m)
offered(?t,?k,?m)
```

Returning to the example with the new representation, assume the initial state is "hidden(necklace,r7)", the final is "taken(necklace,troll1)", and we use the training sequence from the example above. We reproduce this in the syntax used by the *Opmaker* system (McCluskey et al. 2009), below (where the character '@' is used to distinguish an object that does not change state as a result of an action):

```
find_treasure(@knight1,@r7,necklace),
move_to_passage(knight1,p67,r7),
present(@george,@p67,@troll1,necklace),
exchange(necklace,key1,@george,@p67)
```

We can now trace a unique path through the state space of the necklace, using object constraints, from the initial state "hidden(necklace,r7)". The first four possible transitions are:

```
hidden(necklace,r7)=>
          holding(necklace,knight1)
hidden(necklace,r7)=>
          offered(neckalce,knight1,?troll)
hidden(necklace,r7)=>
          taken(necklace,?troll)
hidden(necklace,r7)=>
          hidden(necklace,?r)
```

Given that the first step in this sequence does not refer to a troll, or two different rooms, only the first of the four possible forms of transition below would be possible, as it is the only one which leads to an instantiated transition. The last transition would only be possible if two distinct rooms had been specified in the sequence step. The transitions effected by *move_to_passage* do not affect the necklace, so this step can be ignored. The next step leads to the following transitions:

```
holding(necklace,knight1)=>
           holding(necklace,?knight)
holding(necklace,knight1)=>
           offered(neckalce,knight1,troll1)
holding(necklace,knight1)=>
           taken(necklace,troll1)
holding(necklace,knight1)=>
           hidden(necklace,?room)
```

Hence the second and third transitions are possible. The first is ruled out as there is no new knight object specified to instantiate "?knight", and in the last there is no value specified for "?room". Keeping both these options open, we can examine the last transition, which must change the necklace into state "taken(necklace,troll1)". This constraint eliminate the third option in the step above, leaving transition

```
offered(neckalce,knight1,troll1) =>
           taken(necklace,troll1)
```

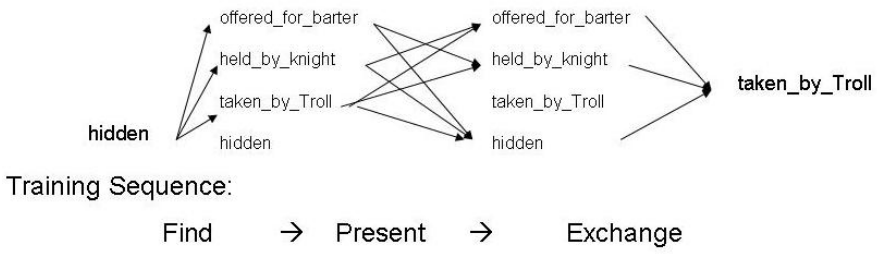


Figure 2: Seven paths through object space

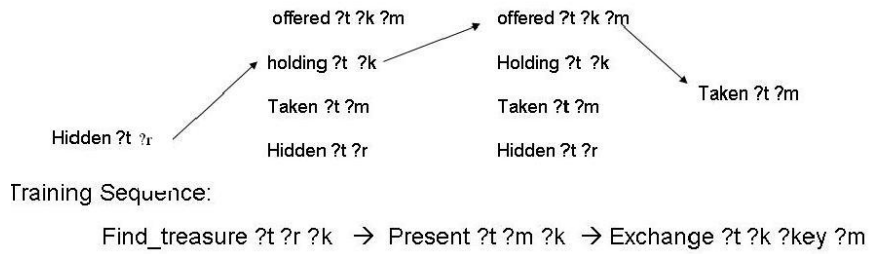


Figure 3: A unique path through object space

as the only option available. This unique path of transitions through the object state space of sort treasure is illustrated in Figure 3.

Summary

The aim of this thesis work is to explore the scope and potential of using object-centred constraints to assist the acquisition of planning domain models from examples. We plan to build on the recent work embodied in the object-centred systems of Opmaker2 (McCluskey et al. 2009), and $LOCM$ (Cresswell, McCluskey, and West 2009); whereas they have produced promising experimental results, there is a need to analyse them theoretically, to find the limits of their output domain models, and the conditions under which they are successful.

In this paper we have reviewed some of the large amount of related work, and introduced a classification for systems that learn planning operator schema. We introduced an example domain to illustrate the kind of constraints that an object centred approach can bring, and stepped through an example sequence showing how a unique object transition path can be determined.

References

Benson, S. S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation, Dept of Computer Science, Stanford University.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proceedings of ICAPS 2009*.

Hogg, C., and Munoz-Avila, H. 2007. Learning Hierarchical Task Networks from Plan Traces. In *Proceedings of the ICAPS'07 Workshop on Artificial Intelligence Planning and Learning*.

Ilghami, O.; Nau, D. S.; Muoz-Avila, H.; and Aha, D. W. 2005. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence* 21(4):388–143.

Ilghami, O.; Nau, D. S.; and Munoz-Avila, H. 2006. Learning to do htn planning. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 390 – 393.

McCluskey, T.; Cresswell, S.; Richardson, N.; and West, M. M. 2009. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, 93–100.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, 665–672. New York, NY, USA: ACM Press.

Shahaf, D., and Amir, E. 2006. Learning partially observable action schemas. In *AAAI*. AAAI Press.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *Journal of Knowledge Engineering* 1.

Wu, K.; Yang, Q.; and Jiang, Y. 2005. ARMS: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.

Dynamic Controllability of Temporally-flexible Reactive Programs

Robert Effinger*[+], Brian Williams[+], Gerard Kelly[‡], and Michael Sheehy[‡]

*Charles Stark Draper Laboratory, Inc.
Autonomous Mission Control Group
555 Technology Square
Cambridge, MA 02139-3563
reffinger@draper.com

[+]Massachusetts Institute of Technology
Model-based Embedded and Robotic Systems
CSAIL, 32 Vassar St. Room 32-226
Cambridge, MA 02139
effinger@mit.edu, williams@mit.edu

[‡]University of Limerick
CTVR, Stokes Institute
Engineering Research Bld. ERO-006,
Limerick, Ireland
gkelly@mit.edu, msheehy@mit.edu

Abstract

In this paper we extend dynamic controllability of temporally-flexible plans to temporally-flexible *reactive programs*. We consider three reactive programming language constructs whose behavior depends on runtime observations; conditional execution, iteration, and exception handling. Temporally-flexible reactive programs are distinguished from temporally-flexible plans in that program execution is conditioned on the runtime state of the world. In addition, exceptions are thrown and caught at runtime in response to violated timing constraints, and handled exceptions are considered successful program executions. *Dynamic controllability* corresponds to a guarantee that a program will execute to completion, despite runtime constraint violations and uncertainty in runtime state. An algorithm is developed which frames the dynamic controllability problem as an AND/OR search tree over possible program executions. A key advantage of this approach is the ability to enumerate only a subset of possible program executions that guarantees dynamic controllability, framed as an AND/OR solution subtree.

Minimal Sufficient Explanations for Factored Markov Decision Processes

Omar Zia Khan, Pascal Poupart, and James P. Black

David R. Cheriton School of Computer Science
University of Waterloo
200 University Avenue West, Waterloo, ON, N2L 3G1 Canada
{ozkhan, ppoupart, jpblack} @ cs.uwaterloo.ca

Abstract

Explaining policies of Markov Decision Processes (MDPs) is complicated due to their probabilistic and sequential nature. We present a technique to explain policies for factored MDP by populating a set of domain-independent templates. We also present a mechanism to determine a minimal set of templates that, viewed together, completely justify the policy. Our explanations can be generated automatically at run-time with no additional effort required from the MDP designer. We demonstrate our technique using the problems of advising undergraduate students in their course selection and assisting people with dementia in completing the task of handwashing. We also evaluate our explanations for course-advising through a user study involving students.

Inference and Decomposition in Planning Using Causal Consistent Chains

Nir Lipovetzky
Departament of Technology (DTIC)
Universitat Pompeu Fabra
08018 Barcelona, SPAIN
nir.lipovetzky@upf.edu

Hector Geffner
Departament of Technology (DTIC)
ICREA & Universitat Pompeu Fabra
08018 Barcelona, SPAIN
hector.geffner@upf.edu

Abstract

Current state-of-the-art planners solve problems, easy and hard alike, by search, expanding hundreds or thousands of nodes. Yet, given the ability of people to solve easy problems and to explain their solutions, it seems that an essential inferential component may be missing. The reasons expressed by people for selecting actions appear to be related to causal chains: sequences of causal links $a_i \rightarrow p_{i+1}$, $i = 0, \ldots, n-1$, such that a_0 is applicable in the current state, p_i is a precondition of action a_i, and p_n is a goal. Some of these causal chains or paths appear to be good, some bad, others appear to be impossible. In this work, we focus on such paths and develop three techniques for performing inference over them from which a path-based planner is obtained. We define the conditions under which a path is consistent, provide an heuristic estimate of the cost of achieving the goal along a consistent path, and introduce a planning algorithm that uses paths as decomposition backbones. The resulting planner, called C3, is not complete and does not perform as well as recent planners that carry extensive but extremely efficient searches such as LAMA, but is competitive with FF and in particular, with FF running in EHC mode which yields very focused but incomplete searches, and thus provides, a more apt comparison. Moreover, many domains are solved backtrack-free, with no search at all, suggesting that planning with paths may be a meaningful idea both cognitively and computationally.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Flexible Execution of Plans with Choice

Patrick R. Conrad and **Julie A. Shah** and **Brian C. Williams**
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory, MERS
32 Vassar St. Room 32-D224, Cambridge, MA 02139
prconrad@mit.edu, julie_a_shah@csail.mit.edu, williams@mit.edu

Dynamic plan execution strategies allow an autonomous agent to respond to uncertainties while improving robustness and reducing the need for an overly conservative plan. In previous work, Williams et al. introduced the Reactive Model-Based Programming Language (RMPL), which is designed to allow engineers to simply and intuitively express the desired behavior of the system (2003). Then the agent's *executive* determines the correct sequence of actions to accomplish this behavior. This paper presents a novel system called Drake that makes steps towards executing an expanded set of choices and temporal constraints dynamically, without significant latency.

RMPL programs can include simple interval constraints on durations and include choices between possible threads of execution. These choices require disjunctive constraints, making the temporal constraints of a special case of the Disjunctive Temporal Problem (DTP).

One effective technique for handling uncertainty is to follow a strategy of least commitment and to delay decision making until execution time. Previous work has developed efficient execution strategies by breaking the executive into a *dispatcher* and a *compiler*. The off-line *compiler* exposes all the implicit constraints in the original plan. The *dispatcher* uses the dispatchable form to quickly make dynamic scheduling decisions.

However, developing flexible executives for plans with choices has been difficult, especially for tightly coupled systems, for example robots working with humans or walking robots with fast dynamics. Recently, Shah and Williams extended the *compiler* and *dispatcher* model to Temporal Constraint Satisfaction Problems (TCSPs), a type of temporal problems with choice (2008). Their executive improved upon the response time of previous systems by reducing the storage and propagation of redundant information. We generalize their algorithm to work with DTPs by recording the minimal differences between the plans required by the dispatcher.

We evaluated Drake's performance on randomly generated structured problems. In Figure 1, we compare the spaced requirements for the compiled form of Drake's compilation of Tsamardinos et al.'s approach of enumerating all the consistent component STPs (2001). Especially on larger problems, Drake uses much less space than prior art, by two orders of magnitude when compared for disjuncts of order two and three. These encouraging results suggest that Drake will support fast flexible execution of plans with choice.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

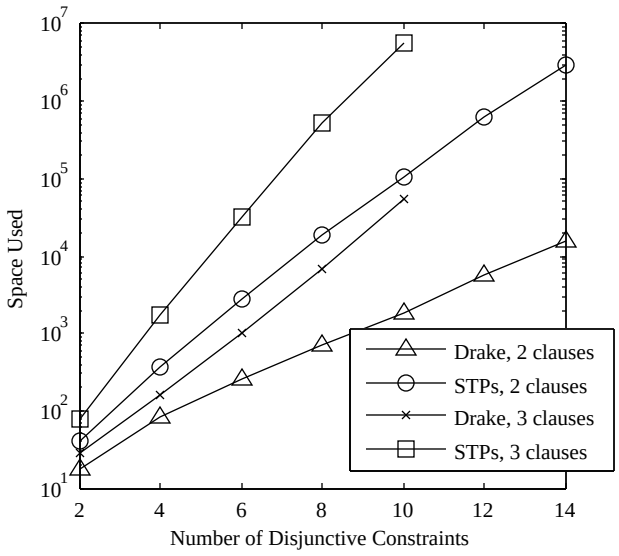


Figure 1: The space used by the dispatchable graphs compiled with Drake's method and by listing component STPs with disjunctive constraints with two and three clauses each.

References

Shah, J. A., and Williams, B. C. 2008. Fast Dynamic Scheduling of Disjunctive Temporal Constraint Networks through Incremental Compilation. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Tsamardinos, I.; Pollack, M.; and Ganchev, P. 2001. Flexible dispatch of disjunctive plans. In *6th European Conference on Planning*, 417–422.

Williams, B.; Ingham, M.; Chung, S.; and Elliott, P. 2003. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE* 91(1):212–237.

A Human-Aware Robot Task Planner

Marcello Cirillo
AASS Research Center
Örebro University, Sweden
marcello.cirillo@aass.oru.se

The growing presence of household robots in inhabited environments arises the need for new robot task planning techniques. These techniques should take into consideration not only the actions that the robot can perform or unexpected external events, but also the actions performed by a human sharing the same environment, in order to improve the cohabitation of the two agents, e.g., by avoiding undesired situations for the human.

This article presents a planning technique able to deal with the presence of the human in the environment. This technique is applied to situations in which there is a controllable agent (the robot) whose actions we can plan, and an uncontrollable agent (the human) whose action we can only try to predict. For simplicity, we assume there are only one robot and one human. Multiple alternative human plans (agendas) are considered, and the domain is partially observable. Therefore the planner generates policies conditional on what the human is observed doing. Actions have duration, but for the purpose of this paper we assume that durations are deterministic.

The planner is progressive, and it searches over belief situations, starting from a given initial belief situation. Each belief situation is a probability distribution over situations. A situation is a tuple $\langle s, rt, ht, ha \rangle$ where s is a state, $rt \in \mathbb{R}^+$ is the time when the robot's latest action ended, $ht \in \mathbb{R}^+$ is the time when the human's latest action ended, and ha is the remaining agenda of the human. An *agenda* is a finite list of consecutive human actions: $(a_1, a_2, ..., a_n)$. By associating different situations with different agendas, the planner can maintain alternative hypotheses about what the human will do.

Each time the planner applies an action of the robot in a belief situation, one or more new belief situations are generated. To compute these new belief situations, the planner takes into account:

- what actions in the human's agenda will finish before or at the same time as the robot's action
- the effects of those human actions and the robot's action
- the observations generated both by the human's and the robot's actions.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Figure 1: The home environment used in our experiment.

The planner searches for a policy that achieves the given goals. In addition, the user can include interaction constraints, which are logical conditions which the policy is not allowed to violate. For instance, the robot may be forbidden to vacuum clean a room where the human is watching TV.

The planning system has been tested on an advanced distributed robotic system called the PEIS home — a small apartment equipped with actuators and sensors of various complexity, including mobile robots (Figure 1). In this setting, the human agendas were produced by a human plan recognition module taking input from a vision system. These agendas were then given to the planner together with other information (world state, goals etc), and the planner generated policies that were then executed on the distributed robotic system (in particular the mobile robot).

In our experiment, the user was executing daily activities in the morning, from 8 am to 1 pm, following one of six predefined agendas. During the same time span, a robotic vacuum cleaner had the task to clean the floor in all the rooms that needed it, minimizing the interference with the user. In our case, this means that the robotic vacuum cleaner had to both operate and wait in rooms that the system had predicted as not occupied by the human at that time. Initially, two possible agendas with identical prefixes were identified, and a policy was generated and successfully executed.

An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP –Abstract–

Mauro Vallati*

Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia, Via Branze 38, 25123 Brescia, Italy
mauro.vallati@ing.unibs.it

The field of automated plan generation has recently significantly advanced. However, while several powerful domain-independent planners have been developed, no one of these clearly outperforms all the others in every known benchmark domain. It would then be useful to have a multi-planner system that automatically selects and combines the most efficient planner(s) for each given domain.

The performance of the current planning systems is typically affected by the structure of the search space, which depends on the considered planning domain. In many domains, the planning performance can be improved by deriving and exploiting knowledge about the domain structure that is not explicitly given in the input formalization. In particular, several approaches encoding additional knowledge in the form of macro-actions have been proposed, e.g., (Botea, Müller & Schaeffer 2005; Newton *et al.* 2007). A macro-action is a sequence of actions that can be planned at one time like a single action. When using macro-actions there is a trade-off to consider. While their use can reduce the number of search steps required to reach a solution, it also increases the search space size; moreover, the effectiveness of a set of macro actions can depend on the particular planner using it.

In this abstract paper we sketch a planner, called PbP (*Portfolio-based Planner*), which automatically configures a portfolio of domain-independent planners. The configuration relies on some knowledge about the performance of the planners in the portfolio and the observed usefulness of automatically generated sets of macro-actions. This configuration knowledge is "learned" by a statistical analysis and consists of: an ordered selected subset of the planners in the initial portfolio, which are combined through a round-robin strategy; a set of useful macro-actions for each selected planner; and some sets of planning time slots. A planning time slot is an amount of CPU-time to be allocated to a selected planner (possibly with a set of macro-actions) during planning.

When PbP is used without this additional knowledge, all planners in the portfolio are scheduled by a simple round-robin strategy where predefined and equal CPU-time slots are assigned to the (randomly ordered) planners. When PbP uses the knowledge computed for the domain under consideration, only the selected cluster of planners (and relative sets of macro actions) is scheduled, their ordering favors the fastest planners for the domain under consideration, and the planning time slots are defined by the learned knowledge.

It should be noted that in PbP the computed macro-actions are not always used by a planner. Assume that a planner P performs very well in a domain D, and thus it is in the set of the planners selected by PbP for solving the problems in D. If in the learning phase PbP observes that the set of macro-actions computed for planner P does not improve the performance of P in D, then PbP schedules the run of P without using macro-actions.

PbP has two variants: PbP.s focusing on speed, and PbP.q focusing on plan quality. PbP.s entered the learning track of the sixth international planning competition (IPC6), and was the overall winner of this competition track (Fern, Khardon & Tadepalli 2008). However, as observed by the IPC6 organizers, for the IPC6 problems the use of the learned knowledge does not speedup the competition version of PbP.s significantly. This behaviour depends on some implementations bugs contained in the preliminary version of PbP.s concerning both the learning phase and the planning phase, and the inefficient use of some Linux shell scripts (evident especially for small or easy problems).

An experimental analysis about an improved implementation of the competition version of PbP.s and about PbP.q confirms the effectiveness of PbP.s, indicate that PbP.q performs better than the IPC6 planners, and show that, contrary to the preliminary version of PbP.s, the learned configuration knowledge is useful for PbP.

For a more detailed description about PbP, the interested reader can see Gerevini, Saetti & Vallati 2009.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2005. Learning Partial-Order Macros from Solutions. In *Proc. of ICAPS-05*.

Newton, M.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In *ICAPS07*.

Gerevini, A. E.; Saetti, A.; and Vallati, M. 2009. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proc. of ICAPS09*.

Fern, A.; Khardon, R.; and Tadepalli, P. 2008. 6th IPC – Learning Track http://eecs.oregonstate.edu/ipc-learn/

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

*The work described in this abstract is a joint work with Alfonso E. Gerevini and Alessandro Saetti.