University of Alberta

Improving AI Planning and Search with Automatic Abstraction

by

Adi Botea ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2006

Your file Votre référence
ISBN: 0-494-13939-0
Our file Notre référence
ISBN: 0-494-13939-0

University of Alberta

Library Release Form

Name of Author: Adi Botea

Title of Thesis: Improving AI Planning and Search with Automatic Abstraction

Degree: Doctor of Philosophy

Year this Degree Granted: 2006

Adi Botea
8515-112 Street, Apt. 1717
Edmonton, Alberta
Canada, T6G 1K7

Date: ______________

Abstract

Planning is ubiquitous in real life. AI planning and single-agent heuristic search, two major areas of artificial intelligence research, focus on machine-generated solutions to a great range of real-life planning applications. To successfully tackle large planning problems, significant advances in technology are necessary.

This research focuses on speeding up planning and single-agent search. Abstraction, a central idea of this work, is explored in three major application domains, each assuming a different level of application-specific knowledge available beforehand.

The first framework is fully automated AI planning, with no application-specific knowledge provided. The contributions include a family of adaptive techniques that automatically infer new information about a domain. Macro-actions are extracted from previously acquired information. Algorithms for ranking, filtering, and using macros at runtime are introduced. Experiments show an improvement of orders of magnitude, as compared to a state-of-the-art planner such as FF, in domains where structural information can automatically be inferred. Macro-FF, an adaptive planner that implements these ideas, successfully participated in the International Planning Competition IPC-4, taking the first place in 3 out of 7 domains where it competed.

As a second domain, abstraction for path-finding on grid maps is explored. Partial application-specific knowledge is assumed, since path-finding usually takes place in a space with topological structure. The main contribution is

Hierarchical Path-Finding A*, an approach shown to achieve up to a 10-fold speed-up in exchange for a 1% degradation in path quality, as compared to a highly optimized implementation of A*.

The third research domain provides a rich application-specific context: the puzzle of Sokoban. The main contribution is a novel solving approach that combines planning with abstraction. A maze is partitioned into rooms and tunnels, allowing the decomposition of a hard initial problem into several much simpler sub-problems. Experiments show that a prototype implementation of these ideas is competitive with a state-of-the-art specialized solver, on a subset of problems.

Acknowledgements

I have often felt that interacting with and learning from Martin Müller and Jonathan Schaeffer, my co-supervisors, has been an exceptional privilege. Their supervision, an amazing combination of expertize, discreetness, and patience, has decisively supported the progress of this research. More importantly, it has changed me both personally and professionally. Additionally, I would like to thank the other committee members, Simaan AbouRizk, Fahiem Bacchus, Joseph Culberson, and Robert Holte, who allocated valuable time to evaluate this work and provide feedback.

Over the last few years, I have learnt many interesting things from the Planning Reading Group, the Games Group, and the Computer Go Seminar at the University of Alberta. I am thankful to all the people who attended meetings of these groups and contributed to their value with interesting comments and discussions.

Portions of the work described in this thesis have been implemented on top of software initially developed by Mike Ady, Fahiem Bacchus, Froduald Kabanza, Markus Enzenberger, Jörg Hoffmann, and Andreas Junghanns.

All the extremely personable and supportive people around me have made the Department of Computing Science into an essentially perfect work environment. Special thanks go to Edith Drummond, whose dedication in helping graduate students goes well beyond her regular job duties.

I will always remember the good times (and coffee) that I have had with Akihiro Kishimoto, Markus Enzenberger, Markian Hlynka and other friends within the Department of Computing Science. I thank my Romanian friends for playing soccer together and for the great times we have had in Edmonton.

For the strongest emphasis, I have left it to the last to acknowledge the essential role played by family support, upon which I have relied. I am very thankful to my wife Vio, whose moral support helped me complete the challenging journey of a PhD program. Throughout these years, she has made a superb effort to live away from her parents and sister, and to re-start her career in a new country. Finally, I dedicate this thesis to my parents, to whom I owe much of what I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 AI Planning and Heuristic Search

Planning has an ubiquitous presence in real life. Humans need to plan many of their activities, from shopping or driving to a destination to industrial processes or construction projects. As a natural consequence, the exploration of planning has become a major theme in the area of computing science, in an attempt to provide machine-generated answers to humans' day-to-day planning jobs. Formal computer theories such as artificial intelligence (AI) planning [32] and heuristic search [74], which model real-life planning, are well-recognized areas of AI research.

The fields of AI planning and heuristic search share many common ideas. At a high level of interpretation, they both try to provide a sequence of actions that will lead to a goal state starting from the current state. Common solutions are present at a more concrete, algorithmic level too. While several major planning approaches exist (e.g., planning as satisfiability and constraint satisfaction [54], the graphplan algorithm [4], planning with hierarchical task networks [26, 80, 84], etc.), planning as heuristic search has proven to be one of the most effective [5, 42].

Planning and search are hard by their nature, both for humans and computers. To illustrate this, consider the problem of airplane transportation. Specifically, consider scheduling flights on a large network of airports while taking into account routes, aircraft capacity, pilots, attendants, airport availability, fuel costs, airport local configuration, etc. Furthermore, assume that the schedule should optimize some parameters such as operating costs or average waiting time between flights.

Such a computation is next to impossible for a human to do, since it is large, will require lots of time to solve, and errors are very likely to occur. Encoding so many constraints into a computer application would result in a very large problem, which exceeds the capabilities of current technology. Each new variable, or new range of values of an existing variable, can result in a combinatorial blow-up of the problem complexity.

Despite the hardness of planning and search problems, great progress has been achieved in the development of efficient solving techniques. The evolution of the international planning competition over its four editions [1, 40, 61, 67] accurately reflects this. Successive editions introduced more and more realistic and computationally challenging benchmarks, or harder problem instances in the same domain. The top performers could successfully solve a large percentage of the problems each time.

However, many real-life domains still pose great challenges for current techniques. Arguably, the advances in planning technology have yet to reach a point that would allow automated planners to assist humans in *many* daily activities. In effect, the need for more efficient planning methods, which push the boundaries of current technology, is of great importance.

1.2 Abstraction

Abstraction can be a good answer to challenging planning problems. Humans often abstract a problem solving task into higher level representations, and a similar idea can successfully be used in computer applications. Consider again the example of airplane transportation. A human planner would never work at such a low level of detail that considers all variables of the model. They would structure the problem hierarchically and decompose it into much smaller subproblems. For instance, separate the problem of flying between airports (the global problem) from the problems that encode local constraints of each airport. The global problem uses a map of inter-connected airports.

On this map, airports are black boxes that ignore local details such as the terminal/gate where to land, how to refuel the airplane, or how to process the passengers' luggage. These details are solved as a separate problem for each airport. The result is that many smaller problems get solved, with a large reduction in the total solving effort.

When the global problem is still too large, it can further be abstracted. The network of airports can be structured into inter-communicating clusters. Flying from Lethbridge, Canada to Trento, Italy is now a sum of smaller problems:

1. Fly from Lethbridge to a close national-size airport such as Edmonton. This involves searching only in the airport network of Alberta.

2. Fly to Toronto, which has many international connections. This time search is performed only in the network of important Canadian airports.

3. Fly to a major European airport such as Frankfurt, Germany. Now the search explores only major airports on the map.

4. Go down in the hierarchy in a similar fashion until the destination airport is reached.

Solutions to planning tasks often have associated metrics that characterize their quality. Examples include the number of steps in a solution, the total execution time, the resources consumed to achieve a goal, etc. Abstraction-based solutions may not guarantee optimality. But even a near-optimal solution might save millions of dollars per year, be more convenient for customers, and even more environmentally friendly than a hand-made schedule.

1.3 Contributions and Target Applications

This thesis focuses on designing, analyzing and evaluating techniques for speeding up planning and search. Abstraction is a central idea of this research. The term "abstraction" has a multitude of meanings, and many approaches from the AI literature can fit into this category. In this thesis, abstraction refers to the process of changing the level of granularity at which a problem is represented. Two successful strategies that refine this high-level idea are reformulating a problem on several hierarchical levels, and using macro-operators.

A macro-operator abstracts several related actions into a single action. When added to a search space as new transitions, macros can reduce the distance to goal states, at the price of increasing the branching factor. To balance this trade-off in favor of faster search, heuristic rules are introduced that aim to prune macros that are probably not shortcuts towards a goal state. As shown in Sections 3.3.1 and 5.1.1, macros can also improve the accuracy of heuristic state evaluation in fully automated AI planning.

In this thesis, abstraction ideas are explored in a variety of frameworks, each assuming a different level of application-specific knowledge:

- Fully automated AI planning. This is also known as domain-independent planning. No application-specific knowledge is available beforehand, and one single *planner* (i.e., solver application) addresses many classes of problems.

- Path-finding on grid maps. This framework assumes partial application-specific knowledge: application domains in this class contain a topological structure (e.g., a city map, a game level, a building where robots navigate, etc.) which can be exploited by efficient solving methods. In principle, one software application can tackle multiple domains with topological structure.

- An application-specific context, the puzzle of Sokoban. No limitation is imposed on the amount of domain-specific knowledge that can be encoded in the solver.

The following subsections provide more details on each framework. In each subsection, first the corresponding application domain is introduced, and then the contributions are outlined.

1.3.1 Domain-Independent AI Planning

AI planning focuses on solving problems expressed in a given standard language such as PDDL [65]. A domain definition includes general information such as object types in that application, relationships that can exist between objects, and actions, along with their preconditions (i.e., conditions that are required for an action to be applicable) and effects. A problem is defined as an initial state and a goal condition. A solution plan is a sequence of actions that reaches the goal starting from the initial state.

In domain-independent AI planning, a planner must address a large class of previously unknown domains. Hence domain-specific knowledge, which often makes a huge contribution to the success of an AI application, is not available beforehand. The challenge is to design generic methods that work well in many application domains. In particular, systems that can adapt their solving strategy to the particularities of a domain are very appealing.

The contributions of this thesis to the area of AI planning include:

- A family of adaptive techniques that automatically learn new information about a domain and use it to speed up planning in future problems in that domain. Domain structure information is inferred with *component abstraction* and/or *solution abstraction*, briefly introduced in the following paragraphs. Macro-operators are generated based on the previously acquired information. Algorithms for efficient filtering, ranking, and runtime use of macros are introduced.

- The contributions are evaluated both with systematic scientific experiments, and at an internationally recognized competition. When new domain information can automatically be inferred, the performance improves by orders of magnitude, as compared to the state-of-the-art planner FF [42]. Macro-FF, an adaptive planner that implements these ideas, successfully participated in the Fourth International Planning Competition IPC-4 [40], taking first place in 3 out of 7 attempted domains.

Component abstraction groups related low-level constants of a planning problem into more abstract entities called *abstract components.* The idea is similar to how humans can abstract features connected through static relationships into a more complex functional unit. For example, a robot that carries a hammer could be considered a single component, which combines the skills of a robot and a hammer. Component abstraction is a clustering procedure in a *static graph.* Nodes of a static graph are distinct objects such as BILL-THE-ROBOT, JACK-THE-ROBOT, THE-RED-HAMMER, etc. Edges model static relationships between objects. The goal of clustering is to identify small local sub-graphs as patterns that are relevant for the domain structure. In this example, a cluster can model an abstract functional entity such as JACK-THE-ROBOT-WITH-THE-RED-HAMMER.

In solution abstraction, the solution of a planning problem is represented as a *solution graph.* Nodes are solution steps (i.e., actions in the plan). Edges model interactions between actions. Solution abstraction analyzes a solution graph to extract local patterns that are relevant to the structure of the given domain. These local patterns, in fact small sub-graphs corresponding to macro-actions, are used for faster planning in new problems.

1.3.2 Path-Finding on Grid Maps

The objective of path-finding is to plan a route from an initial position to a destination on a map with obstacles. The problem is of crucial importance in applications such as robotics, transportation, traffic optimization in computer networks, and commercial computer games, a fast growing multi-billion dollar industry.

Besides the potentially large search space, path-finding problems often exhibit significant additional challenges. Path-finding problems in robotics and commercial computer games usually have to be solved in real time and under constraints of limited memory and CPU resources. Moreover, a problem environment can change dynamically, and parts of the map may be unknown in advance. For some domains, important criteria regarding the quality of solutions (e.g., "look human-like") can be hard to quantify.

For the problem of path-finding on grid maps, the contributions of this thesis include:

- Hierarchical Path-Finding A* (HPA*), a hierarchical path-finding algorithm that achieves high performance and successfully addresses challenges such as those mentioned before.

- Experimental results for hierarchical search on a variety of game mazes, showing up to a 10-fold speed improvement in exchange for a 1% degradation in path quality, as compared to a highly optimized implementation of A*.

In this thesis, partitioning a map into a set of clusters is called *topological abstraction*. Clustering allows to decompose an original problem into several, much smaller problems: one problem associated with each cluster, and one global problem that models interactions between clusters. Since a grid is usually represented as a graph of atomic locations (tiles) inter-connected by neighborhood relationships, topological abstraction is a graph clustering problem.

HPA* abstracts a map into linked local clusters based on topological abstraction. Each cluster generates a graph with entrance nodes and crossing path edges. At the local level, optimal distances for crossing each cluster are precomputed and cached. At the global level, a whole cluster is traversed in a single big step. Graphs of adjacent clusters are connected through common entrance points. In this way, all cluster graphs are combined into one abstract graph that covers the whole problem map.

A hierarchy can be extended to more than two levels. Small clusters are grouped together to form larger clusters. Computation of graph edges (crossing distances) for a large cluster uses the graphs of the smaller contained clusters.

Path planning starts with a search at the most abstract level of the graph. An abstract solution can gradually be refined until a complete low-level solution is obtained. HPA* is fully automated, needs no domain-specific knowledge other than the assumption of topological structure, and allows great flexibility in execution, solving parts of the problem if and when they are needed.

Name	Target Applications	Application Independent	Utility Scope	Graph Abstraction
Component Abstraction	AI planning	Yes	Domain	Small clusters as local patterns
Solution Abstraction	AI planning	Yes	Domain	Small subgraphs as local patterns
Topological Abstraction	Path-finding	Partially	Map	Clusters that partition a map
	Sokoban	No		

Table 1.1: Abstraction strategies.

1.3.3 Sokoban

Sokoban is a single player game created in Japan in the early 1980s. A man in a maze has to push several stones from their initial locations to designated goal locations [10]. See Section 2.3 for a detailed description of the rules. As shown in Chapter 7, Sokoban is a challenging application for both humans and computers, being characterized by long optimal solutions, a large branching factor, and the presence of dead ends in the search tree. The contributions of this research to this domain include:

- A novel solving approach based on problem decomposition. Similar to HPA*, a topological abstraction strategy decomposes a map into rooms connected through tunnels. This allows for the decomposition of a hard initial problem into several simpler sub-problems, with great potential for reducing the overall search effort.
- Experiments show that, on problem instances that an initial implementation of this approach can tackle, its performance is competitive with a state-of-the-art specialized solver such as Rolling Stone [49].

Table 1.1 summarizes key properties of component abstraction, solution abstraction, and topological abstraction, the three major approaches explored in this thesis work. Topological abstraction in path-finding is listed as partially depending on the nature of a given application domain. The only application-specific knowledge assumed is the existence of a topological space as part of the application definition. The column "Utility Scope" indicates the range where an abstraction remains valid once it is completed. For component abstraction and solution abstraction, the utility scope is a planning domain. Information acquired from training instances in a domain can be used to solve new instances in the same domain. For topological abstraction, the utility scope is a map: several problems on the same map can reuse the map's abstraction.

As pointed out before, each of these abstractions can be seen as a particular case of the more general problem of abstraction in a graph. The last column of Table 1.1 indicates the result of the graph abstraction performed by each technique.

1.4 Publications and Thesis Overview

The structure of the remaining chapters is the following: Chapter 2 surveys related work in the AI literature and provides a background for the remaining chapters. Contributions to domain-independent AI planning are the topic of Chapters 3–5. Chapter 3 presents macro-operators created with component abstraction. This work was previously reported in [13] and parts of [15]. Chapter 4, based on [14] and parts of [15], describes macro-operators created with solution abstraction. Chapter 5 presents experiments in AI planning previously discussed in [13, 14, 15]. Chapter 6, based on [12], summarizes research on hierarchical path-finding. With a content similar to [10], Chapter 7 explores how planning can be performed in an abstracted representation of the Sokoban puzzle. Chapter 8 presents the conclusion of the thesis and summarizes directions for future work.

Chapter 2

Literature Review

This chapter surveys related artificial intelligence research and provides a background for the following chapters. Section 2.1 focuses on AI planning research. Section 2.2 surveys related work on abstraction in map navigation. Sokoban is the topic of Section 2.3. Section 2.4 describes work on abstraction in other single-agent search domains. Section 2.5 presents the conclusions of this chapter.

2.1 AI Planning

The planning contributions of this thesis are in the area of *classical* planning. In principle, the same ideas can be applied to extensions of classical planning such as temporal planning, numerical planning, and planning with incomplete information, but this is beyond the focus of this thesis. This section starts with a short introduction to classical planning. Then three approaches, which are often combined in planning research, are discussed:

- Planning as heuristic search, one of the most successful approaches to AI planning.
- Abstracting planning problems based on the *implicit structure* of a domain, which is not part of the standard definition of a problem. Such domain structure can be either automatically inferred, or encoded by hand.

- Macro-operators in AI planning, a topic that needs to be revived and combined with current state-of-the-art technology.

2.1.1 Background of Classical Planning

When solving a planning task, a planner takes as input a *domain* and a *problem instance*. Several problem instances are usually defined for one domain. Figures 2.1 and 2.2 show a domain file and a problem file in ToyLogistics, a simple application where *trucks* can transport *crates* between *places*. Variations of the ToyLogistics' operators DRIVE, LOAD, and UNLOAD are present in several other domains as well. ToyLogistics is a simplified version of Depots, a domain used in the third international planning competition [61]. ToyLogistics is represented in STRIPS, a simple but widely used subset of the standard planning language PDDL. For information on PDDL and subsets such as STRIPS and ADL, see [31, 65].

In PDDL, a domain file contains general information such as *types*, *predicates* and *operators* (*actions*). A predicate has a name and a list of (typed) variables. Each operator o has a name, a set of (typed) parameters, preconditions and effects. In STRIPS, the precondition $Prec(o)$ is a conjunction of predicates. An effect is split into a conjunction of *positive* (*add*) effects $Add(o)$ and a conjunction of *negated* (*delete*) effects $Del(o)$. More complicated subsets of PDDL such as ADL allow precondition formulas that use quantifiers, implications, disjunctions, conjunctions, and negations. Effect formulas can use universal quantifiers, conjunctions, negations, and *conditional effects* [65]. A conditional effect is a pair (c, e), where c is a condition formula and e is an effect formula. See below for details on how conditional effects work when an action is applied to a state.

In PDDL, a problem instance file contains specific information such as the (typed) *objects* (*constant symbols*), the *initial state* s_0, and the *goal condition* G of the instance.

A *state* of a problem is represented by a collection of binary variables called *facts*. Facts are obtained from domain predicates by instantiating their parameters with constant symbols. Only true facts in a state s are explicitly

```
(define (domain ToyLogistics)
  (:types truck location crate)
  (:predicates
    (at ?t - truck ?p - place)
    (at ?c - crate ?p - place)
    (in ?c - crate ?t - truck)
    )
  (:action drive
    :parameters (?t - truck ?p1 - place ?p2 - place)
    :precondition (and (at ?t ?p1))
    :effect (and (not (at ?t ?p1)) (at ?t ?p2))
  )
  (:action unload
    :parameters (?t - truck ?c - crate ?p - place)
    :precondition (and (at ?t ?p) (in ?c ?t) )
    :effect (and (not (in ?c ?t)) (at ?c ?p))
  )
  (:action load
    :parameters (?t - truck ?c - crate ?p - place)
    :precondition (and (at ?t ?p) (at ?c ?p))
    :effect (and (not (at ?c ?p)) (in ?c ?t))
  )
)
```

Figure 2.1: ToyLogistics in STRIPS.

stated:

$$s = \{p | p \text{ is true in } s\}.$$

All unspecified facts are false, according to the so-called *closed world assumption* [65].

Function $\gamma : S \times A \rightarrow S$ models transitions between states. S is the set of states, and A is the set of *instantiated actions.* An instantiated action is obtained from an operator by instantiating all its parameters with constant symbols. If $s \not\Rightarrow Prec(a)$, then $\gamma(s, a)$ is undefined. If $s \Rightarrow Prec(a)$, then a is applicable to s, and $\gamma(s, a)$ is the state s' obtained by applying a's effects to s. For example, in STRIPS this means that all precondition facts of a are

```
(define (problem ToyInstance1)
  (:domain ToyLogistics)
  (:objects
    place0 place1 - place
    truck0 - truck
    crate0 - crate
  )
  (:init
    (at crate0 place0)
    (at truck0 place1)
  )
  (:goal (at crate0 place1)
  )
)
```

Figure 2.2: A problem instance for ToyLogistics.

true in s: $Prec(a) \subseteq s$. The resulting state s' is

$$s' = \gamma(s, a) = (s \cup Add(a)) - Del(a).$$

In ADL, a conditional effect (c, e) is considered only if $s \Rightarrow c$. In such a case, e becomes part of the effect formula that creates s' from s.

A planning task is to find a sequence of instantiated actions called a *solution plan*

$$\pi = a_1 a_2 \ldots a_n$$

that reaches a *goal state* s_n starting from the initial state s_0:

$$s_{i+1} = \gamma(s_i, a_{i+1}), i \geq 0 \wedge s_n \Rightarrow G.$$

When looking for a solution, planners usually explore a *search space* associated with the current planning task. The nature of a search space depends upon the solving strategy chosen. Different planning approaches can explore different search spaces. Forward chaining planning [32] explores the state space defined above. The root is the initial state s_0, and the successors of a state s are the resulting states of all actions applicable in s.

In regression planning, which searches from the goal towards the initial state, a state in the search space is a collection of facts seen as goal conditions.

The root of this search space is G. The successors of a state s in this space are obtained as follows: For simplicity, assume that an action a achieves a condition $p \in s$ and does not delete any condition in s. Then the successor of s corresponding to a is obtained from s by removing p and adding all preconditions of a. The next two subsections contain more details and references about forward and regression planning.

Partial-order planning [73] explores a space of partial plans. The root is an empty plan, and a successor of a partial plan is obtained by resolving a *flaw* (e.g., add a new action that satisfies a goal condition, or add a new ordering constraint between two actions of the partial plan). In SAT planning [54], a problem is represented as a SAT formula, and states in this search space are partial assignments to the formula's variables.

2.1.2 Planning as Heuristic Search

Planning as heuristic search attempts to compute a solution plan with single-agent search techniques. The direction of space exploration can be either from the initial state towards the goal state (*forward-chaining search*) or from the goal state towards the initial state (*regression search*). The most popular search strategies are based on *hill-climbing* or *best-first search.* A few variations of these strategies are described later in this section.

A heuristic state evaluator guides the problem space exploration. Given the generic nature of fully automated planning, the only knowledge about the application at hand that a heuristic function can use comes from the domain and problem formulations in a standard planning language. It is especially challenging to design a heuristic evaluation function that uses no additional hand-coded information, and works well in many classes of problems.

Many successful planners use heuristic search. The following paragraphs focus on fully automated planners that have had a major impact on the development of the ideas in this thesis. Heuristic-search planners that can use hand-coded information (e.g., TLPlan, TALPlanner, SHOP) are described in Section 2.1.3.

Bonet and Geffner's Heuristic Search Planner (HSP) [5] first demonstrated the efficiency of heuristic search in modern planning, and generated a lot of research interest on this topic. HSP implements a heuristic state evaluator based on problem relaxation. Given a state s and a fact p, $h(s,p)$ is the length of a shortest sequence of *relaxed* actions that achieves p starting from s. A relaxed action is obtained from a regular action by ignoring its delete effects. Given a collection of goal facts $G = \{g_1, ..., g_k\}$, the heuristic evaluation of state s is

$$h(s) = \sum_{i=1}^{k} h(s, g_i). \tag{2.1}$$

This additive formula introduces yet another approximation, assuming that goal facts are independent. This heuristic can overestimate the real distance and hence is not *admissible*. Admissibility of a heuristic h means that, for each state s, $h(s) \leq h^*(s)$, where $h^*(s)$ is the minimum cost of a path from s to a goal state. This property ensures that an algorithm such as A* [33] produces optimal solutions with respect to their cost.

Several versions of this planner exist that differ mainly in their search strategy. HSP implements an incomplete hill-climbing algorithm. HSP2 implements weighted A* (wA*) [75], a best-first search algorithm [5]. In wA*, nodes in the open queue (i.e., nodes generated but not expanded yet) are sorted according to a value $f(s)$ associated to each state s:

$$f(s) = (1 - w)g(s) + wh(s), 0.5 \leq w \leq 1.$$

This is a weighted sum of $g(s)$, the distance from the root to the current state s, and $h(s)$, the estimated distance to a goal state. When $w = 0.5$, wA* is equivalent to A*. Values of w larger than 0.5 are used with the purpose of achieving a goal state faster, as nodes evaluated closer to a goal state are expanded with increased priority. On the other hand, solutions computed with wA*, $w > 0.5$, are not guaranteed to be optimal, even if the heuristic h is admissible.

In HSP and HSP2, which use forward-chaining search, an important performance bottleneck is that the heuristic evaluation has to be recomputed from scratch in each state. HSPr performs regression (backwards) search using wA*. The benefit of regression search is a much faster computation of heuristic state evaluation. Recall that regression search tries to reach s_0 by exploring a space where states $s = \{p_1, ..., p_n\}$ are collections of facts seen as current goal conditions. The evaluation $h(s_0, s)$ of a state $s = \{p_1, ..., p_n\}$ is

$$h(s_0, s) = \sum_{i=1}^{n} h(s_0, p_i). \qquad (2.2)$$

Since the first parameter of h in Equation 2.2 is fixed (i.e., s_0), $h(s_0, p)$ is precomputed for each fact p in the problem. Then, for each state s, $h(s_0, s)$ is quickly obtained with Equation 2.2. Note that the idea of precomputing $h(s, p)$ is hard to apply in forward search, since s can take arbitrary values (see Equation 2.1).

Hoffmann's planner Fast Forward (FF) [42] significantly advanced the standards established by HSP. Due to its great success, FF's planning strategy is implemented in many current planners. Perhaps the most important contribution of FF is a new domain-independent heuristic function that proved to be very successful in practice. The new method preserves the action relaxation of HSP, but does not assume that goal conditions will be achieved independently. For each state that has to be evaluated, the distance to a goal state is approximated by the length of a relaxed plan that achieves all goal conditions starting from the current state. The relaxed plan is computed with *relaxed graphplan*, a relaxation of the standard graphplan algorithm [4] in which delete effects of actions are ignored. Solving a relaxed problem *optimally* is NP-hard [16], but suboptimal relaxed plans can be found in polynomial time. Despite this significant reduction, FF often spends large amounts of time doing heuristic state computations. FF's heuristic is not admissible, but in practice it acts as a lower-bound for the real value in most cases.

FF implements two forward search algorithms. *Enforced hill climbing* (EHC) is a fast but incomplete algorithm that greedily searches for a goal state in the problem space. If EHC fails, because of either its incompleteness or the absence of a solution, a complete *best-first search* (BFS) algorithm is launched to find a path to a goal state.

EHC starts from the initial state of a problem and performs a local search using a breadth-first strategy. When a state with a better evaluation than the starting state is found, the current local search stops and a new local search is launched starting from the newly found state. In EHC, the relaxed graphplan computation for a state is used not only to find a heuristic evaluation, but also to further prune the search space with *helpful action pruning.* When a state is expanded, only moves that occur in the relaxed plan and can be applied to the current state are considered.

BFS is in fact wA* with $w = 1$, an algorithm also known as *pure heuristic search* [58]. When ordering nodes in the open queue, only h, the estimated distance to a goal state, is taken into account. Nodes evaluated to be closer to a goal state are expanded with higher priority. This strategy tends to find solutions with fewer expanded nodes for the price of sub-optimality.

Vidal's planner YAHSP (Yet Another Heuristic Search Planner) [87, 88] advances FF's approach in two significant directions. First, the use of a relaxed plan is extended. The motivation is that a relaxed plan often contains useful information about the solution of the real problem. If this information is compressed to only one number, the heuristic evaluation of the current state, much useful information might be lost. To address this, Vidal introduces *lookahead policies.* A lookahead policy executes parts of the relaxed plan in the real world. This often provides a path towards a goal state with no search and few states evaluated. This technique heuristically orders the actions in the relaxed plan and iteratively applies them as long as this is possible. When the lookahead procedure cannot be continued with actions from the relaxed plan, a *plan repair* method selects a new action to be applied.

As a second contribution, YAHSP combines EHC, BFS, and lookahead policies into one single algorithm, in an attempt to exploit the benefits of each. EHC is fast and can cut off large parts of the space with helpful action pruning, while BFS is complete. The new algorithm preserves the completeness of BFS, but expands *helpful nodes* generated by helpful actions with higher priority than the remaining nodes.

HSP, FF and YAHSP use some kind of relaxation of the graphplan algorithm for heuristic evaluation of states. Helmert's planner Fast Downward implements a new approach based on causal analysis [34, 35]. The motivation of this work is that graphplan relaxation often loses much of the structure of the problem, and the heuristic becomes inaccurate. This is the case, for example, in transportation domains. When a mobile object such as a truck or an airplane moves from location A to location B, in a relaxed state the object can be in both locations at the same time. In bad cases, the relaxed plan has so little relevance for the real problem that the search becomes blind.

Fast Downward represents problem states with multi-valued variables rather than as in the classical style with propositional logic. In the example above, there will be a variable for each mobile object, with a value for each possible location of that object. Further, a *causal graph* is defined for a problem. Each state variable is a node in the graph. If changing the value of a variable v can depend on changes to a variable u, a causal link (u, v) is defined. One or several subgraphs, called $SAS^+ - 1$ structures, are extracted from the causal graph. A $SAS^+ - 1$ structure has one node (variable) called the *high-level* variable and several nodes called *low-level* variables. There is one edge from each low-level variable to the high-level variable. $SAS^+ - 1$ structures are used to compute the heuristic value of a state. For each structure, a local plan is computed that changes the high-level variable from the current value to the goal value. Local plans of all structures are combined into the heuristic of the global state.

2.1.3 Abstraction in Planning

Automatic discovery and exploitation of the implicit structure of a domain has been explored by Knoblock [55]. In this work, a hierarchy of abstractions is built starting from the initial low-level problem description. A new abstract level is obtained by dropping literals from the problem definition at the previous abstraction level. Planning first produces an abstract solution and then iteratively refines it to a low-level representation. The hierarchy is built in such a way that, if a refinement of an abstract solution exists, no backtracking across abstraction levels is necessary during the refinement process. Backtracking is performed only when an abstract plan has no refinement. Such situations can be arbitrarily frequent, with negative effects on the system's performance.

Bacchus and Yang [3] define a theoretical probabilistic framework to analyze planning in hierarchical models. Abstract solutions of a problem at different abstraction levels are hierarchically represented as nodes in a tree structure. A tree edge indicates that the target node is a refinement of the start node. An abstract solution can be refined to the previous level with a given probability. Hierarchical search in this model is analytically evaluated. The analytical model is further used to enhance Knoblock's abstraction algorithm. The enhancement refers to using estimations of the refinement probabilities for abstract solutions.

Fox and Long propose algorithms that exploit the symmetry present in a planning domain [29, 30]. Symmetries between objects are identified starting from the initial state of a problem [29]. Two objects are symmetrical if swapping them does not change the initial state of the problem. Symmetries are further used to identify equivalent actions that can be applicable to a state. The search space can safely be pruned based on equivalent actions: it is enough to generate only one action that represents an entire equivalence class. In [29], the only symmetries recognized in search are a subset of the initial symmetry group extracted from the initial state of a problem. However, many other symmetries can exist at various levels of a search space. Hence the authors extend their method to dynamically identify larger equivalence classes [30].

Two successful approaches that use hand-crafted information are *temporal logic* control rules and *hierarchical task networks*. These can be seen as forms of abstraction by hand, since planning uses information about the structure of a domain that is not explicitly encoded in the domain definition. In planning with temporal logic control rules, a formula is associated with each state in the problem space. The formula of the initial state is provided with the domain description. The formula of any other state is obtained based on its predecessor's formula. When the formula associated with a state can be proven false, that state's subtree is pruned. The best known planners of this kind are TLPlan [2] and TALPlanner [60].

Hierarchical task networks (HTNs) guide and restrict planning by using a hierarchical representation of a domain. Human experts design hierarchies of tasks that show how the initial problem can be broken down to the level of regular actions. The idea was introduced by Sacerdoti [80] and Tate [84], and has been widely used in real-life planning applications [89]. SHOP2 by Nau *et al.* [72] is a well-known heuristic search planner where search is guided by HTNs.

2.1.4 Macro-Operators in Planning

Early work on macro-operators in AI planning includes Fikes and Nilsson's planner called STRIPS [28]. Macros are obtained from solution plans by replacing constant arguments of actions with generic variables. Minton extended this work by introducing techniques that filter a set of learned macro-operators [68]. In his approach, two types of macro-operators are preferred: S-macros, which occur with high frequency in problem solutions, and T-macros, which can be useful but have low priority in the original search algorithm. Iba proposes generating macro-operators at run-time using the so-called peak-to-peak heuristic [47]. A macro traverses a "valley" between two peaks of the heuristic state evaluation. This can correct problems with the heuristic evaluation. A macro filtering procedure uses both simple static rules and dynamic statistical data. Mooney considers whole plans as macros and introduces partial ordering of operators based on their causal interactions [70].

Work on improving planning based on solutions of similar problems has been reported by Veloso and Carbonell [86]. Large collections of *cases* are stored as more instances are solved in a domain. A case is an entire solution of a solved problem annotated with additional relevant information such as explanations of successful or failed search decisions. When a new problem is fed to the planner, cases corresponding to similar problems are used to guide the current planning process. The same idea of reusing solutions of past instances is explored by Kambhampati [53] in a hierarchical planning framework. In this work, solutions are annotated with causal dependencies between the effects and the preconditions of solution steps. When a new problem is being solved, an old plan is modified into a valid solution to the current problem. In both cited works, similarity metrics are defined that determine what stored solutions should be retrieved and used to solve the current instance.

McCluskey and Porteous focus on constructing planning domains starting from a natural language description [64]. The approach combines human expertise and automatic tools, and addresses both correctness and efficiency of the obtained formulation. Macro-operators are a major technique that the authors propose for efficiency improvement. In this work, a state in a domain is composed of the local states of several variables called dynamic objects. Macros model transitions between the local states of a variable.

Recently, Coles and Smith implemented support for macro-operators in their planner Marvin [18]. Macros are generated with two different techniques. First, action sequences that escape a *plateau* (i.e. reach a state with a better heuristic starting from a local minimum) are discovered *online* and cached for later use. Second, an *offline* method generates a reduced problem by exploiting symmetries in the original instance. The solution of this problem is used to generate macros that will be used in the main search. No macros are stored from one problem instance to another.

2.2 Path-Finding

The problem of path-finding is to compute a path between two given locations on a map that can contain both blocked areas and passable areas. Path-finding is important in numerous applications such as commercial computer games, robotics, transportation, etc. Path-finding problems are usually solved by running a single-agent search such as A* on a graph associated with the problem at hand. A common way to obtain such a graph is to apply a grid onto the problem map. Unblocked grid cells on the map become graph nodes. Graph edges represent adjacency relationships between cells. Other methods of abstracting maps into search graphs, such as visibility points, quadtrees, and navigation meshes, are discussed in the following subsections. This section reviews path-finding in commercial games, abstraction applied to robot navigation, and other relevant work.

2.2.1 Abstraction for Path-Finding in Commercial Computer Games

Path-finding in games using a two-level hierarchy is described by Rabin [77]. The author provides only a high-level presentation of the approach. A map is abstracted into clusters such as rooms in a building or square blocks on other topologies. An abstract action crosses a cluster.

Another important hierarchical approach for path-finding in commercial games uses *points of visibility* [78]. This method exploits the local topology of a domain to define an abstract graph that covers the map. Nodes represent corners of convex obstacles. Edges link all nodes that can see each other (i.e., they can be connected by a straight line). This method is particularly useful when the number of obstacles is relatively small and they have a convex polygonal shape. The efficiency decreases when many obstacles are present and/or their shape is not a convex polygon. Consider the case of a map containing a forest, a dense collection of small obstacles. Modeling such a topology with points of visibility would result in a large graph (in terms of both number of nodes and edges) with short edges. The key idea of abstraction, traveling long distances in a single step, would not work. When the problem map contains concave or curved shapes, the method either has poor performance or needs sophisticated engineering to build the graph efficiently.

A *navigation mesh* (also known as a NavMesh) is a powerful abstraction technique useful for 2D and 3D maps. In a 2D environment, this approach covers the unblocked area of a map with a (minimal) set of convex polygons. A method for building a near optimal NavMesh is presented in [85]. This method relaxes the condition of the minimal set of polygons and builds a map coverage much faster.

Many contributions to the problem of path-finding in computer games come from work on commercial games rather than academic research. Hierarchical search appears to be used by several game companies. The algorithmic details are not public.

2.2.2 Abstraction for Path-Finding in Robotics

Quadtrees have been proposed for hierarchical map decomposition in robot navigation [81]. A map is partitioned into square blocks of different sizes such that a block contains either only empty cells or only blocked cells. The problem map is initially partitioned into 4 blocks. If a block contains both obstacle cells and walkable cells, then it is further decomposed into 4 smaller blocks, and so on. A move in this abstracted framework connects centers of two adjacent blocks. Since an agent always goes to the middle of a box, solutions are sub-optimal.

The solution quality can be improved by *framed quadtrees* [17, 90]. In framed quadtrees, the border of a block is augmented with cells at the highest resolution. An action crosses a block between any two border cells. Since this representation permits many angles of direction, the solution quality improves significantly. However, framed quadtrees use more memory than quadtrees.

2.2.3 Other Relevant Work

Learning macro-operators for path-finding is explored by Markovitch in [62]. Given a fixed map and a distribution of the node pairs for which paths have to be computed, training problems are generated and solved. The solutions are analyzed to extract common patterns that will be stored as macro-operators. Macros are filtered according to the so-called *minimum-to-better* rule, which is very similar to the plateau-escaping rule presented at the end of Section 2.1.4.

Shekhar *et al.* decompose an initial problem graph into a set of fragment sub-graphs and a global boundary sub-graph that links the fragment sub-graphs [82]. Shortest paths are computed and cached for future use. The authors analyze what shortest paths (i.e., from which sub-graphs) to cache, and what information to keep (i.e., either complete path or only cost) for best performance when limited memory is available.

Yap analyzes grid abstractions in path-finding problems [91]. Grid abstraction discretizes a problem map into a grid with a regular structure. The structure of a grid is determined by the shape of its atomic cells (e.g., squares, hexagons, etc.), their relative positioning (e.g., squares can be either aligned or shifted like bricks in a wall), and possible transitions between adjacent cells (e.g., in four straight directions or in eight straight and diagonal directions). The author analyzes how performance in path-finding is affected when these grid abstractions are combined with search algorithms such as A* and IDA* [56].

Reese and Stout classify path-finding problems according to the type of the results that are sought, the environment type, and the amount of information available [79]. Challenges specific to each problem type and solving strategies such as re-planning and using dynamic data structures are briefly discussed.

Moore *et al.* use a multi-level hierarchy to enhance the performance of multiple goal path-planning in a *Markov Decision Process* (MDP) framework [71]. The problem posed is to efficiently learn near optimal policies $\pi^*(x, y)$ to travel from x to y for all pairs (x, y) of map locations. The number of policies that have to be computed and stored is quadratic in the number of map cells. To improve both the memory and time requirements, at the price of losing optimality, a multi-level structure is used – a so called *airport hierarchy*. All locations on the problem map are *airports* that are assigned to different hierarchical levels. The strategy for travelling from x to y is similar to traveling by plane in the real world. First, travel to bigger and bigger airports until a connection exists to the area that contains the destination. Second, go down in the hierarchy by travelling to smaller airports until the destination is reached.

Precup *et al.* use macro-actions to speed up planning in reinforcement learning [76]. In this work, a macro-action is defined as a starting state, a policy that will be followed, and a completion function that tells the probability of completing the macro-action at a given time step. The completion function of a macro models sub-goal achievement. Since policies have probabilistic transitions, execution of a macro-action may end-up in different final states. The authors focus on developing a mathematical foundation of this model. They illustrate the model with an application of navigating inside a building with rooms. Macro-actions model leaving a room and reaching a hallway point. Macros solve the local problems of navigating inside a room, which are

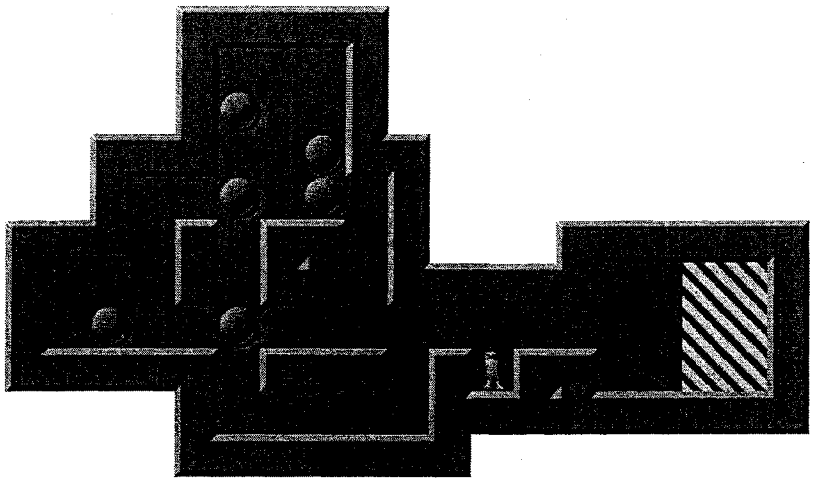


Figure 2.3: Sokoban puzzle.

compiled away from the global problem.

2.3 Sokoban

Figure 2.3 shows the Sokoban problem #1 in the 90-problem test suite available at [48]. It consists of a maze which has two types of squares (also called tiles): inaccessible *wall* squares and accessible *interior* squares. Several *stones* are initially placed on some of the interior squares. A *man* can walk around by moving from his current position to any adjacent stone-free interior position. Consider a row (or column) of three adjacent interior squares such that the man is on one end square, a stone is in the middle, and the other end square is free. A push move is to shift both the man's and stone's positions by one square in the direction of the initially free square. The goal is to push all stones to marked *goal squares.* In Figure 2.3, the six goal squares are the marked ones at the right end of the maze.

Culberson showed that Sokoban is PSPACE-complete [19]. In computer Sokoban, the state of the art is represented by Junghanns' solver *Rolling Stone* [52] and an anonymous Japanese researcher [49]. Little information is available about the latter. Both solvers are able to find solutions for about two thirds of the standard 90-problem test suite [52].

While centered on a classical single-agent search approach, Rolling Stone owes part of its success to abstraction techniques. Two of the most effective concepts in Rolling Stone are *tunnel macros* and *goal macros* [52]. A *tunnel* is a line of adjacent free cells bordered by walls. The ends of the tunnel are connected to the rest of the maze. Tunnel macros are move sequences that push a stone through a tunnel from one end to another. A tunnel macro does not interact with the rest of moves that are legal on the maze. Interleaving single tunnel moves with other moves leads to a blow-up of the search.

A *goal room* is an area that connects with the rest of the maze via a few entrances and contains one or several goal squares. Goal macros are precomputed sequences that arrange stones into a goal room. While losing completeness, goal macros eliminate many deadlocks that could be generated by incorrect filling of a goal room.

2.4 Abstraction in Single-Agent Search

This section summarizes contributions that are not necessarily designed for planning, path-finding, or Sokoban, but still remain relevant for the work reported in this thesis.

Culberson and Schaeffer [20, 21] introduce pattern databases, large collections of abstractions of problem states that represent a heuristic evaluation function. State abstraction is performed by replacing part of the features that characterize a state with a generic "don't care" symbol. Subsequent research significantly extended this idea. Korf used pattern databases to compute optimal solutions to the Rubik's Cube puzzle [59]. Holte *et al.* analyzed how to best use a fixed amount of memory when several pattern databases are available [45]. The authors show that several smaller databases can be better than one single database. Recent work exploits symmetries that pattern databases can exhibit as a result of symmetries in problem states [27].

Edelkamp uses pattern databases in domain-independent planning [23, 24]. Hernádvölgyi applies uses pattern databases to find macro-operators in Rubik's Cube [37]. Korf [57] and Iba [47] apply macro-operators to the sliding-tile puzzle.

Holte *et al.* [43] use *homomorphism abstraction* to analyze how a hierarchical representation of a search space can be exploited in search. In homomorphism abstraction, a graph node at an abstract level represents several nodes from the previous level. Several levels can be built on top of one another until an abstract one-state space is obtained. The authors argue that two strategies for exploiting an abstract solution at the previous level have mainly been considered in the literature. The first strategy uses an abstract level to build a heuristic function at a lower level. The distance between two low-level nodes is approximated by the length of an abstract path between the two abstract nodes that correspond to each low-level node. The second strategy generates a lower-level solution by refining an abstract solution. Nodes in the abstract solution act as sub-goals in the low-level problem. Refinement generates a path between two abstract nodes. The authors show the similarities between the two approaches and develop a unified framework.

Holte *et al.* introduce Hierarchical A* [46], a search method designed for situations when no heuristic function is provided. Hierarchical A* uses homomorphism abstraction to structure an original search space on several levels. The hierarchy is then exploited according to the first strategy mentioned before: search at a given level provides a heuristic function for the previous level. The authors present techniques (e.g., smart caching of search results) that result in expanding less nodes than blind search in the initial space.

Homomorphism abstraction and refinement in explicitly represented graphs are explored by Holte *et al.* in [44]. A clustering algorithm called *STAR* is introduced as an efficient approach for homomorphism abstraction in generic graphs. Several refinement techniques are evaluated, out of which a method called *AltO* is shown to be the overall winner. In AltO, steps of an abstract solution are graph nodes rather than graph edges. A search to find an abstract solution and a search to refine a solution are performed in opposite directions. AltO is opportunistic in the sense that steps in an abstract solution can be skipped when the refinement process finds shortcuts towards the goal state.

Helmstetter and Cazenave use abstraction in a solitaire card game called Gaps [36]. When this puzzle is played with a standard 52-card deck, 4 types of moves exist, one for each suit. If moves of only one suit are considered, they come in a forced sequence, with no branching. Branching is generated when moves of different suits are considered at the same time. At some points, sequences of different suits may interact: the subsequent evolution of a suit can depend on whether a move was made for another suit. The authors exploit these features of the puzzle to abstract an initial search space into *blocks*. A block is a part of the space where move sequences of different suits do not interact with each other. Blocks are efficiently searched, since there is no need to interleave moves of different types.

2.5 Conclusions

This chapter has presented AI research related to this thesis work. The focus has been on the state of the art in using abstraction in search and planning. This survey indicates that variations of abstraction ideas have generated lots of interest in AI planning and heuristic search. Many of the successful contributions have been applied in a domain-specific context. There are rather few application-independent success stories in this area.

There is much room left for research on how abstraction can be exploited in faster problem solvers. This implies both designing new algorithms and applying abstraction to new domains. In particular, abstraction appears to be necessary in domains where low-level search is often unable to find solutions using the available CPU time and memory resources. Examples of such domains are some current planning benchmarks, path-finding on realistic maps, and Sokoban, the target applications of this research.

Chapter 3

Component Abstraction in AI Planning

In many domains, the performance of a planner can be improved by inferring and exploiting information about the domain structure that is not explicitly encoded in the initial PDDL formulation. The implicit structural information that a domain encodes is, arguably, proportional to how difficult the domain is, and how realistically this models the world. For example, consider driving a truck between two locations. This operation is composed of many subtasks in the real world. To name just a few, the truck should be fueled and have a driver assigned. In a detailed planning formulation, several operators such as FUEL, ASSIGN-DRIVER, and DRIVE would be defined. This representation already contains implicit information about the domain structure. It is quite obvious for a human that driving a truck between two remote locations would be a macro-action where first the truck is fueled and assigned a driver (with no ordering constraints between these two actions) and next the drive operator is applied. In a simpler formulation, one can remove the operators FUEL and ASSIGN-DRIVER. Now driving a truck is modeled as a single action, and part of the original structure has been removed from the model.

In this chapter an automated method is presented that learns such implicit domain knowledge and uses it to simplify planning for new problem instances. This is useful in fully automatic planning, where only a domain and a problem definition are fed into a planner. Additional input that would encode, for instance, human knowledge of the domain at hand is not provided.

1. Analysis – Extract new information about the domain structure.
2. Generation – Build macro-operators based on the previously acquired domain structure.
3. Filtering – Select the most promising macro-operators.
4. Planning – Use the selected macro-operators to improve planning in future problems.

Figure 3.1: A generic planning approach that uses abstraction to generate macro-operators.

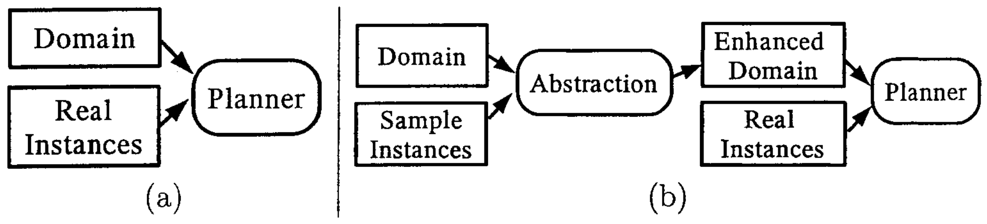


Figure 3.2: (a) Standard planning framework. (b) CA-ED – Integrating component abstraction and macro-operators into the standard planning framework.

As shown in Figure 3.1, this learning approach has four-steps. At step 1, component abstraction is introduced as a novel technique to infer knowledge about the structure of a domain. Then a small set of useful macro-operators is produced in steps 2 and 3. Assume the original domain is expressed in STRIPS, a simple but widely used subset of the standard planning language PDDL. The selected macro-operators are added to the initial domain formulation, resulting in an enhanced domain expressed in the same description language. The definitions of the enhanced domain and new problem instances can be given as input to any STRIPS planner, with no work required to implement step 4. Once the enhanced domain formulation is available in standard STRIPS, a planner makes no distinction between a macro-operator and a normal operator [13]. This approach is called *CA-ED* for Component Abstraction – Enhanced Domain.

Figure 3.2 compares the general architecture of CA-ED with a standard planning framework. In standard planning, a planner takes as input a domain and a problem instance. In CA-ED, abstraction is used to enhance the original definition of a domain. The enhanced domain and problem instances can be fed into a standard planner. The box Abstraction in the figure includes steps 1–3 above. Abstraction is performed within a training session that uses one or several sample problems from a domain. For each sample problem, related low-level objects are grouped together into new components. Macro-operators that group local actions at the level of one component are generated with a local analysis. After all sample instances have been processed, filtering is used to select the "best" macros, which will be added as new operators to the initial domain.

If the above procedure is successful and useful macro-operators are learned, planning will be affected in several important ways, analyzed in detail in Section 3.3:

- New actions are added to the search space, with the effects of reducing the distance to goal states and increasing the branching factor.
- Heuristic state evaluation, using a state-of-the-art method such as Hoffmann's relaxed graphplan [42], can become more accurate.
- Preprocessing cost, as well as run-time cost per node, will often increase.

The rest of this chapter is structured as follows: The next section describes component abstraction. Section 3.2 focuses on generating and filtering macros. Benefits and limitations of CA-ED are analyzed in Section 3.3. The last section contains conclusions and ideas for future work.

3.1 Component Abstraction

Humans often abstract objects connected through permanent (static) relationships into one functional unit. For instance, a robot with a hammer could be considered a single unit, with both mobility and maintenance skills. In this work, such units are modeled with *abstract components* (or, shorter, com-

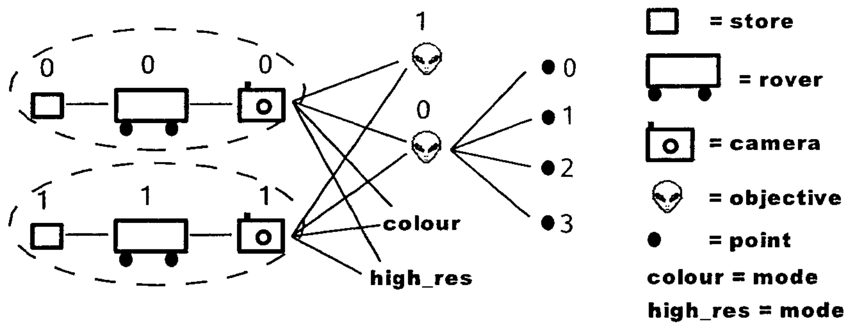


Figure 3.3: Static graph of a Rovers problem.

ponents). Component abstraction automatically identifies components in a two-step process:

1. Build the problem *static graph*, which models *permanent* relationships between constant symbols (objects) of a problem.

2. Build abstract components with a clustering procedure. Formally, an abstract component is a connected subgraph of the static graph.

3.1.1 Building the Static Graph of a Problem

A static graph is constructed from the PDDL representation of a planning problem. Each constant that is an argument of at least one static fact defines a node in the static graph. A fact is *static* for a problem if it is true in the initial state and no action can change its value. All constants in a fact are linked pairwise.[1] All edges in the graph are labeled with the name of the corresponding fact.

A Rovers problem is used as an example of how component abstraction works. See Appendix B for a description of Rovers. Figure 3.3 shows the static graph of the sample problem. Starting from the left of the picture, the nodes include two stores (STORE0 and STORE1), two rovers (ROVER0 and ROVER1), two photo cameras (CAM0 and CAM1), two objectives (OBJ0 and OBJ1), two camera modes (COLOUR and HIGH_RES), and four waypoints (POINT0,...

[1]In other words, a fact of arity $k \geq 2$ will generate a clique between its k constant arguments.

POINT3). The edges correspond to the static predicates (STORE_OF ?S - STORE ?R - ROVER), (ON_BOARD ?C - CAMERA ?R - ROVER), (SUPPORTS ?C - CAMERA ?M - MODE), (CALIBRATION_TARGET ?C - CAMERA ?O - OBJECTIVE), and (VISIBLE_FROM ?O - OBJECTIVE ?W - WAYPOINT).

The two marked clusters on the left are examples of abstract components found by CA-ED. Each component is a rover equipped with a camera and a store. Details about how components are built follow in Section 3.1.2.

To identify static facts necessary to build the static graph, the set of domain operators $\mathcal{O}$ is used to partition the predicate set $\mathcal{P}$ into two disjoint sets, $\mathcal{P} = \mathcal{P}_F \cup \mathcal{P}_S$, corresponding to *fluent* and *static* predicates. A predicate p is fluent if p is part of an operator's effects (either positive or negative). In STRIPS, this translates to

$$p \in \mathcal{P}_F \Leftrightarrow \exists o \in \mathcal{O} : p \in Add(o) \cup Del(o).$$

Otherwise, p is static, denoted by $p \in \mathcal{P}_S$.

Facts that are true in the initial state of the problem and correspond to static predicates are static. Static predicates that are unary[2] or contain two or more variables of the same type will be ignored. The latter kind of facts are often used to model topological relationships, and can lead to large components. Appendix A.1 provides the pseudocode of the method that builds the static graph of a problem. Details about identifying static facts in domains with hierarchical types, which need additional care, are presented in Appendix A.2.

3.1.2 Building Abstract Components

Abstract components are built as connected subgraphs of the static graph of a problem. Clustering starts with abstract components of size 1, containing one node each, that are generated based on a randomly selected domain type t, the *seed* type. For each node of type t in the static graph, a new abstract component is created. Abstract components are then iteratively extended with a greedy approach.

[2]In many current domains, unary static facts have been replaced by types associated with variables.

Step	Current Predicate	Used Pred.	COMPONENT0		COMPONENT1	
			Consts	Facts	Consts	Facts
1			CAM0		CAM1	
2	(SUPPORTS ?C - CAMERA ?M - MODE)	NO	CAM0		CAM1	
3	(CALIBR_TARGET ?C - CAMERA ?O - OBJECTIVE)	NO	CAM0		CAM1	
4	(ON_BOARD ?C - CAMERA ?R - ROVER)	YES	CAM0 ROVER0	(ON_BOARD CAM0 ROVER0)	CAM1 ROVER1	(ON_BOARD CAM1 ROVER1)
5	(STORE_OF ?S - STORE ?R - ROVER)	YES	CAM0 ROVER0 STORE0	(ON_BOARD CAM0 ROVER0) (STORE_OF STORE0 ROVER0)	CAM1 ROVER1 STORE1	(ON_BOARD CAM1 ROVER1) (STORE_OF STORE1 ROVER1)

Table 3.1: Building abstract components for the Rovers example.

Next the clustering procedure is run on the Rovers example. A more formal description, including pseudo-code, is provided in Appendix A.3. As said before, Figure 3.3 shows the two abstract components built for the example. The steps of applying the clustering procedure to the example are summarized in Table 3.1, and correspond to the following actions:

1. Randomly choose a seed type (CAMERA in this example), and create one abstract component for each constant of type CAMERA: COMPONENT0 contains CAM0, and COMPONENT1 contains CAM1. Next, iteratively extend the components created at this step. One extension step uses a static predicate that has at least one variable type already encoded into the components.

2. Choose the predicate (SUPPORTS ?C - CAMERA ?M - MODE), which has a variable of type CAMERA. Since ending up with a large component containing the whole graph is not desired, this method does not allow merging two existing components. Hence a test is performed whether the static facts based on this predicate keep the existing components

separated. These static facts are (SUPPORTS CAM0 COLOUR), (SUPPORTS CAM0 HIGH_RES), (SUPPORTS CAM1 COLOUR), and (SUPPORTS CAM1 HIGH_RES). The test fails, since constants COLOUR and HIGH_RES would be part of both components. Therefore this predicate is not used for component extension.

3. Similarly, the predicate (CALIBRATION_TARGET ?C - CAMERA ?O - OBJECTIVE) is not used, as it would add the constant OBJ1 to both components.

4. Predicate (ON_BOARD ?C - CAMERA ?R - ROVER) is used for component extension. The components are expanded as shown in Table 3.1, Step 4.

5. Predicate (STORE_OF ?S - STORE ?R - ROVER), whose type ROVER has previously been encoded into the components, is considered. This predicate extends the components as presented in Table 3.1, Step 5.

After Step 5, no further component extension can be performed. There are no other static predicates using at least one of the component types to be tried for further extension. At this moment the quality of the decomposition is evaluated. In this example it is satisfactory (see discussion below), and the process terminates. Otherwise, the decomposition process restarts with another domain type.

The quality of a decomposition is evaluated according to the size of the built components, where size is defined as the number of low-level types in a component. In experiments, the size was limited to values between 2 and 4. The lower limit is trivial, since an abstract component should combine at least two low-level types. The upper limit was set heuristically, to prevent the abstraction from building just one large component. These relatively small values are also consistent with the goal of limiting the size and number of macro operators. More details on this issue are provided in Section 3.2.

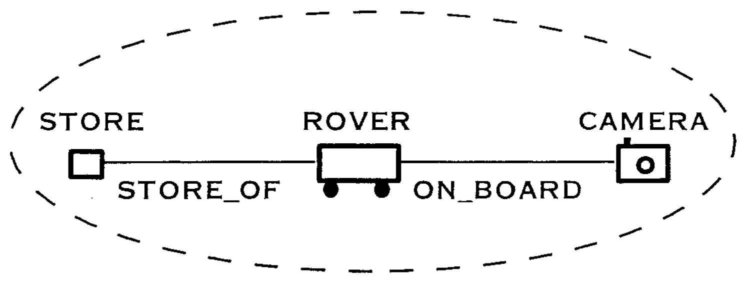


Figure 3.4: Abstract type in Rovers.

3.1.3 Assigning Types to Abstract Components

Following the standard of typed planning domains, where each object has a type, abstract components are assigned *abstract types*. Figure 3.4 shows the abstract type assigned to the components of the Rovers example. As shown in this figure, the abstract type of a component is a graph obtained from the component graph by changing the node labels. The constant symbols used as node labels have been replaced with their low-level types (e.g., constant CAM0 has been replaced by its type CAMERA).

The example also shows that components with identical structure have the same abstract type. As shown below, the concept of identical structure is a strong form of graph isomorphism, which preserves the edge labels as well as the types of constants used as node labels. Assume $Nodes(ac)$ is the set of constants (subgraph nodes) and $Facts(ac)$ is the set of static facts associated with a component ac. The arguments $c^1, ..., c^k$ of a fact $f(c^1, ..., c^k) \in Facts(ac)$ are nodes of ac: $c^i \in Nodes(ac)$.

Two abstract components ac_1 and ac_2 have identical structure if:

1. $|Nodes(ac_1)| = |Nodes(ac_2)|$; and

2. $|Facts(ac_1)| = |Facts(ac_2)|$; and

3. there is a bijective mapping $p : Nodes(ac_1) \longrightarrow Nodes(ac_2)$ such that

   - $\forall c \in Nodes(ac_1) : Type(c) = Type(p(c))$;
   - $\forall f(c_1^1, ..., c_1^k) \in Facts(ac_1) : f(p(c_1^1), ..., p(c_1^k)) \in Facts(ac_2)$;
   - $\forall f(c_2^1, ..., c_2^k) \in Facts(ac_2) : f(p^{-1}(c_2^1), ..., p^{-1}(c_2^k)) \in Facts(ac_1)$.

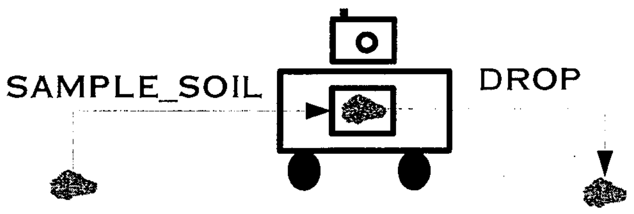


Figure 3.5: Example of a macro in Rovers.

For each abstract type a local analysis is performed with the goal of improving planning at the component level. In CA-ED, local analysis is used to generate macro operators. This is only one possible way to exploit component abstraction. Other ideas are mentioned in the last section of this chapter.

3.2 Creating Macro-Operators

Similar to a regular STRIPS operator, a CA-ED macro-operator m has a name $N(m)$, a set of variables $V(m)$, a set of preconditions $Prec(m)$, a set of add effects $Add(m)$, and a set of delete effects $Del(m)$. Hence macro operators can be added as new operators to an initial domain formulation. Figure 3.5 shows an example of a macro in Rovers. It collects a soil sample into a rover's store, and drops it back, with the overall effect of having analyzed that soil sample. Figure 3.6 shows complete STRIPS definitions for the sample macro and the operators that it contains.

Macro operators are obtained in a two-step process. First, an extended set of macros is built and, second, the macros are filtered in a quick training process. Since empirical analysis indicates that the extra information added to a domain definition should be quite small, the methods described next tend to minimize the number of macros and their size, measured by the number of variables, preconditions and effects. Static macro generation uses many constraints for pruning the space of macro operators, and discards large macros. Finally, dynamic filtering keeps only a few top macros for solving future problems.

```
(:action SAMPLE_SOIL__DROP
  :parameters
    (?r - rover ?s - store ?p - waypoint)
  :precondition
    (and (equipped_for_soil_analysis ?r) (empty ?s)
    (store_of ?s ?r) (at_soil_sample ?p) (at ?r ?p))
  :effect
    (and (not (at_soil_sample ?p)) (have_soil_analysis ?r ?p))
)
(:action SAMPLE_SOIL
  :parameters
    ( ?r - rover ?s - store ?p - waypoint)
  :precondition
    (and (equipped_for_soil_analysis ?r) (empty ?s)
    (store_of ?s ?r) (at_soil_sample ?p) (at ?r ?p))
  :effect
    (and (not (empty ?s)) (not (at_soil_sample ?p)
    (full ?s) (have_soil_analysis ?r ?p))
)
(:action DROP
  :parameters
    ( ?r - rover ?s - store)
  :precondition
    (and (full ?s) (store_of ?s ?r))
  :effect
    (and (not (full ?s)) (empty ?s))
)
```

Figure 3.6: STRIPS definitions of macro SAMPLE_SOIL__DROP and the operators that it contains.

3.2.1 Macro Generation

For each abstract type *at*, macros are generated that perform local processing within a component of type *at*, according to the locality rule detailed below. Macro generation is a forward search in the space of possible macro operators. The root state is an empty macro, with empty sets of operators, variables, preconditions, and effects. Each search step appends an operator to the current macro, and fixes the variable mapping between the operator and the macro. Adding a new operator o to a macro m modifies $Prec(m)$, $Add(m)$, and $Del(m)$

```
void addOperatorToMacro( operator $o$,
                         macro $m$,
                         variable-mapping $\sigma$) {
  for (each precondition $p \in Prec(o)$) {
    if ($p \notin Add(m) \cup Prec(m)$)
      $Prec(m) = Prec(m) \cup \{p\}$;
  }
  for (each delete effect $d \in Del(o)$) {
    if ($d \in Add(m)$)
      $Add(m) = Add(m) - \{d\}$;
    $Del(m) = Del(m) \cup \{d\}$;
  }
  for (each add effect $a \in Add(o)$) {
    if ($a \in Del(m)$)
      $Del(m) = Del(m) - \{a\}$;
    $Add(m) = Add(m) \cup \{a\}$;
  }
}
```

Figure 3.7: Adding operators to a macro.

as shown in Figure 3.7. The variable mapping σ in the procedure is used to check the identity between operator's predicates and macro's predicates (e.g., in $p \notin Add(m) \cup Prec(m)$). Two predicates are considered identical if they have the same name and the same set of parameters. The variable mapping σ shows what variables (parameters) are common in both the macro and the new operator.

The search is selective: it uses a set of rules for pruning the search tree and for validating a built macro operator. Validated macros are goal states in this search space. The following pruning rules are used for static filtering:

- The *negated precondition rule* prunes operators with a precondition that matches one of the current delete effects of the macro operator. This rule avoids building incorrect macros where a predicate should be both true and false.

- The *repetition rule* prunes operators that generate cycles. A macro containing a cycle is either useless, producing an empty effect set, or it can be written in a shorter form by eliminating the cycle. A cycle in a macro is detected when the effects of the first k_1 operators are the same as for the first k_2 operators, with $k_1 < k_2$. In particular, if $k_1 = 0$ then the first k_2 operators have no effect.

- The *chaining rule* requires that for consecutive operators o_1 and o_2 in a macro, o_2's preconditions must include at least one positive effect of o_1. This rule is motivated by the idea that the action sequence of a macro should have a coherent meaning.

- The size of a macro is limited by imposing a maximum length and a maximum number of preconditions. Similar constraints could be added for the number of variables or effects, but this was found unnecessary. Limiting the number of preconditions indirectly limits the number of variables and effects. Large macros are generally undesirable, as they can increase by a large margin the cost of evaluating a state with the relaxed graphplan algorithm.

- The *locality rule* is an important criterion that controls how component abstraction can be used to generate macro-operators. The following high-level discussion provides the intuitive idea and the motivation of this rule. Then a formal definition is given.

  Intuitively, macros generated component abstraction should perform local processing at the level of one component. Macros that change two or more abstract components at the same time are pruned. To motivate this, consider planning with component abstraction as a hierarchical planning framework, where each component defines a local problem. To limit the complexity of planning, it is desirable that local problems do not interact with each other directly i.e., each local problem interacts only with the next level in the hierarchy. This assumption is made in many hierarchical models described in the literature [3, 10, 12, 72].

  The formal definition of the locality rule is the following. Given an abstract type *at* and a macro m, let the *local static preconditions* be the static predicates that are part of both m's preconditions and at's edges. Local static preconditions and their parameters in m's definition define a graph structure (different variable bindings for the operators that compose m can create different graph structures). Locality requires that this graph is isomorphic with a subgraph of at. In other words, all local static preconditions are part of the same abstract component.

3.2.2 Macro Ranking and Filtering

The goal of ranking and filtering is to reduce the number of macros and use only the most efficient ones for solving problems. The overhead caused by poor macros can exceed their benefit. This is known as the *utility problem* [69].

A simple but efficient and practical approach to dynamic macro filtering can be effective at selecting a small set of useful macro operators. This method counts how often a macro operator is instantiated as an action in solution plans. The more often a macro has been used in the past, the greater the chance that the macro will be useful in the future.

For ranking, each macro operator is assigned a weight that estimates its efficiency. All weights are initialized to 0. Each time a macro is present in a plan, its weight is increased by the number of occurrences of the macro in the plan plus 10 bonus points. No effort was spent on tuning parameters such as the bonus. For *common macros* that are part of solutions of *all* training problems, any bonus value $v \geq 0$ will produce the same ranking among these common macros. No matter what the value v is, each common macro will receive $v \times T$ bonus points, where T is the number of training problems. Hence the occurrence points decide the relative ranking of common macros.

The simplest problems in a domain, which are usually the first ones in a collection, are used for training. For these simple problems, all macro operators are added to the domain, giving each macro a chance to participate in a solution plan and increase its weight. After the training phase, the best macro operators are selected to become part of the enhanced domain definition. In experiments, 2 macros, each containing two steps, were added as new operators to the initial sets of 9 operators in Rovers, and 5 operators in Depots and

Satellite. In these domains, such a small value was observed to be a good tradeoff between the benefits and the additional preprocessing and run-time costs. See the next section for an analysis of the benefits and the additional costs, and Chapter 5 for an empirical evaluation. In more domains with larger initial sets of operators, using more macros could probably be beneficial.

3.3 Analysis

3.3.1 How Macros Affect Planning in CA-ED

This section analyzes the impact of macro-operators in planning. The discussion focuses on how macros change the heuristic evaluation of states, the search space, the cost per node, and the preprocessing costs.

In experiments, enhanced domains and problem instances were solved using Hoffmann's planner FF [42]. When added as new operators to the initial domain formulation, macros affect FF's relaxed graphplan algorithm. When a state is evaluated with relaxed graphplan, a relaxed problem is solved that achieves a goal state starting from the current state. Relaxation is performed by ignoring all delete effects of actions. The length of the relaxed plan is used as a heuristic evaluation of the real distance from the current state to a goal state.

To illustrate the benefits of macros in relaxed graphplan, consider the example in Figures 3.5 and 3.6. Operator SAMPLE_SOIL has one add effect (FULL) and one delete effect (EMPTY) that update the status of a store. Similarly, operator DROP updates the store status with two such effects. However, when macro SAMPLE_SOIL__DROP is used, the status of the store does not change: it was empty before, and it will be empty after. No effects are necessary to express changes in the store status. Hence two delete effects (one for each operator) are safely eliminated from the real problem before relaxation is performed. The relaxed problem is more similar to the real problem and the information loss is less drastic.

A well-known property of macros is that they change the search space by adding new transitions between states. This is called an embedding abstraction [46]. A node that normally needs several steps to reach becomes a direct successor when the macro is applied. From the perspective of a search algorithm, embedding abstraction increases the branching factor but can reduce the distance between the initial state and a goal state. The trade-off is important for the overall performance of a search algorithm. See Section 5.1.1 for an empirical evaluation of the effects of macros on heuristic state evaluation and search depth.

Many planners expand operators into instantiated actions by replacing all parameters and quantifiers with constant symbols. This is done once for each problem, as a preprocessing step. Given an operator o, let $(v_1, v_2, ..., v_k)$, with $Type(v_i) = t_i$, be the set of all its parameters and quantifiers (the latter can be present in ADL domains). Assume that, in a problem, each type t has a number of n_t objects. An upper bound of the complexity of instantiating operator o is $O(n_{t_1} \times n_{t_2} \times ... \times n_{t_k})$. FF optimizes this by computing only a superset of the *reachable* actions. An action is reachable if its preconditions are true in a state reachable from the initial state. See [42] for details.

CA-ED macros increase the number of domain operators. Furthermore, since macros tend to have slightly more variables than regular operators, their instantiation cost can be higher.

Macros often increase the average cost per node in a search as well. Processing a node is usually dominated in cost by calling the evaluation function. Using macros makes relaxed graphplan more expensive, since an increased number of actions will be involved. See Section 5.1 for an empirical evaluation.

3.3.2 Limitations of CA-ED

The architecture of CA-ED has two main limitations. First, component abstraction is currently applied only to domains with static facts. Second, adding macros to the original domain definition is limited to simple subsets of the standard planning language PDDL such as STRIPS. The reason is that when a macro is added to a domain as a new operator, its complete definition is required, including precondition and effect formulas. This is easy to achieve in STRIPS, as illustrated in Figures 3.6 and 3.7

However, if more complex PDDL subsets such as ADL are used, adding macros to a domain file becomes impractical for two main reasons. First, the precondition and effect formulas of a macro are hard to infer from the formulas of contained operators. Second, even if the previous issue is solved and a macro with complete definition is added to a domain, the costs for preinstantiating it into ground macro-actions can be large.

Figure 3.8, which shows operator MOVE from the Airport domain used in IPC-4 [40], illustrates how challenging the formula inference is in ADL. The preconditions and the effects of this operator are quite complicated formulas that include quantifiers, implications and conditional effects. The formulas of a macro MOVE—MOVE with a given parameter mapping σ would have to be automatically composed from the preconditions and effects of the two contained operators.

Even if the above issue is solved and macros can be added as new domain operators, preinstantiating an ADL macro into ground actions can be costly. Action instantiation was discussed in Section 3.3.1. The cost of this preprocessing step can be much higher in ADL than in STRIPS because of the existence of quantifiers. ADL macros tend to have larger sets of parameters and quantifiers than regular ADL operators, and therefore their instantiation can significantly increase the total preprocessing costs. ADL Airport is a good illustration of how important this effect can be. As shown in Section 5.2, the preprocessing is so costly as compared to the main search that it dominates the total cost of solving a problem in this domain. Further increasing the preprocessing effort with new operators is not desirable in such domains.

3.4 Conclusion and Future Work

This section has presented CA-ED, a method that learns information about the structure of a planning domain and exploits it in new searches. Learning is performed in a training step that uses one or several sample problems from a domain. For each sample problem, abstract components are created that

```
(:action move
 :parameters
   (?a - airplane ?t - airplanetype ?d1 - direction
   ?s1 ?s2 - segment ?d2 - direction)
 :precondition
   (and    (has-type ?a ?t) (is-moving ?a)
           (not (= ?s1 ?s2))
           (facing ?a ?d1) (can-move ?s1 ?s2 ?d1)
           (move-dir ?s1 ?s2 ?d2) (at-segment ?a ?s1)
           (not
             (exists    (?a1 - airplane)
                        (and (not (= ?a1 ?a)) (blocked ?s2 ?a1))))
           (forall    (?s - segment)
                      (imply    (and    (is-blocked ?s ?t ?s2 ?d2)
                                        (not (= ?s ?s1)))
                                (not (occupied ?s))))
   )
 :effect
   (and    (occupied ?s2) (blocked ?s2 ?a)
           (not (occupied ?s1))
           (when    (not (is-blocked ?s1 ?t ?s2 ?d2))
                    (not (blocked ?s1 ?a)))
           (when    (not (= ?d1 ?d2))
                    (not (facing ?a ?d1)))
           (not (at-segment ?a ?s1))
           (forall    (?s - segment)
                      (when    (is-blocked ?s ?t ?s2 ?d2)
                               (blocked ?s ?a)))
           (forall    (?s - segment)
                      (when    (and    (is-blocked ?s ?t ?s1 ?d1)
                                       (not (= ?s ?s2))
                                       (not (is-blocked ?s ?t ?s2 ?d2)))
                               (not (blocked ?s ?a))))
           (at-segment ?a ?s2)
           (when (not (= ?d1 ?d2))
           (facing ?a ?d2))
   )
)
```

Figure 3.8: Operator MOVE in ADL Airport.

group together related low-level objects. Analysis of the components is used to generate macro-operators that perform local processing at the level of one component. Macros are filtered down to a few top candidates that are added as new operators to the initial domain.

The applicability of CA-ED is limited to domains that are expressed in STRIPS and contain static facts in their definition. The next chapter shows how CA-ED can be extended around these limitations.

A challenging long-term goal of component abstraction would be automatic reformulation of planning domains and problems. When a real-world problem is abstracted into a planning model, the corresponding formulation is expressed at an abstraction level that a human designer considers appropriate. However, choosing a good abstraction level could be a difficult problem for humans. Planning domains and problems may be generated automatically as a translation from other areas of computing science. For example, the Promela domains in IPC-4 [40] have been obtained from the area of model checking.

As shown in Section 3.3.1, a macro added to an original domain formulation as a regular operator influences the results of the heuristic function. This is convenient (no changes are necessary in the planning engine), but limited only to STRIPS domains. For other subsets of PDDL, the relaxed graphplan algorithm can be extended with special capabilities to handle macros when no enhanced domain definition is provided.

To explain the behavior of the relaxed graphplan heuristic, Hoffmann analyzes topological features of planning domains both empirically and theoretically [38, 39]. This work could be extended to explore how macro-operators affect the topology of planning benchmarks.

Chapter 4

Solution Abstraction in AI Planning

This chapter presents SOL-EP (Solution Abstraction – Enhanced Planner), an approach similar to CA-ED but more general. SOL-EP was designed with the goal of eliminating the limitations of CA-ED. First, the applicability is extended from STRIPS domains to ADL domains. Second, CA-ED generates macros only from component abstraction, which is limited to domains with static predicates. The new method generates macros from solutions of sample problems, increasing its generality. Third, the size of macros increases from 2 moves to arbitrary values. Fourth, the definition of macros is generalized, allowing partially ordered sequences.

As in CA-ED, the four main steps of SOL-EP are the ones shown in Figure 3.1: analysis, generation, filtering and planning. However, each step is performed differently than in CA-ED. At step 1, domain knowledge is acquired with *solution abstraction*, rather than component abstraction as in CA-ED. Solution abstraction builds a structure called a *solution graph* from the linear action sequence that a planner produces for a problem. A solution graph contains one node for each step in the plan. Edges model the causal effects that an action has on subsequent actions of the linear plan. At step 2, *partial-order* macro-operators (i.e., macros with partial ordering of their operators) are extracted from a solution graph. Steps 1 and 2 can be repeated for several problem instances as part of a training process. In step 3, the set of generated macros is filtered such that only the most promising macros are kept for future

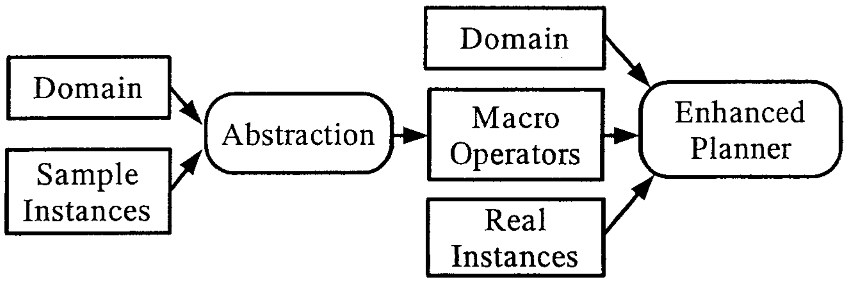


Figure 4.1: The general architecture of SOL-EP. Enhanced Planner means a planner with capabilities to handle macros.

use. Finally, in step 4, the selected macros are used to improve planning in new problems.

The general architecture of this approach is shown in Figure 4.1. For comparison with classical planning and CA-ED, see Figure 3.2. As before, the module Abstraction implements steps 1–3. Macros produced with abstraction are distinct input data for the planner rather than being added to the original domain formulation. This allows the generalization from STRIPS to ADL domains. As discussed in Section 3.3.2, inferring the precondition and effect formulas of a macro from the formulas of contained operators is hard in ADL. Hence the precondition and effect formulas of a SOL-EP macro are not explicitly stated. For this reason, macros cannot be added to the original domain formulation anymore. For step 4, the planner is enhanced with code to handle macro operators.

Using macro-actions at run-time can potentially introduce the utility problem [69], which appears when the savings are dominated by the extra costs of macros. The potential savings come from the ability to generate a useful action sequence with no search. On the other hand, macros can increase the branching factor. Many instantiations of a selected macro-operator could be applicable to a state, but only a few would actually be shortcuts towards a goal state. If all these instantiations are considered, the induced overhead can be larger than the savings achieved by the useful instantiations. To select what macro instantiations to use for state expansion, heuristic techniques such as *helpful macro pruning* and *goal macro pruning* are introduced. See Section

4.2.3 for details.

The rest of this chapter is structured as follows: The next two sections provide details about building a solution graph, and how to extract and use macro-operators. Macro-FF, a planner that implements SOL-EP and CA-ED on top of FF, participated in the fourth international planning competition IPC-4. The results are highlighted in Section 4.3. The last section contains conclusions and future work ideas.

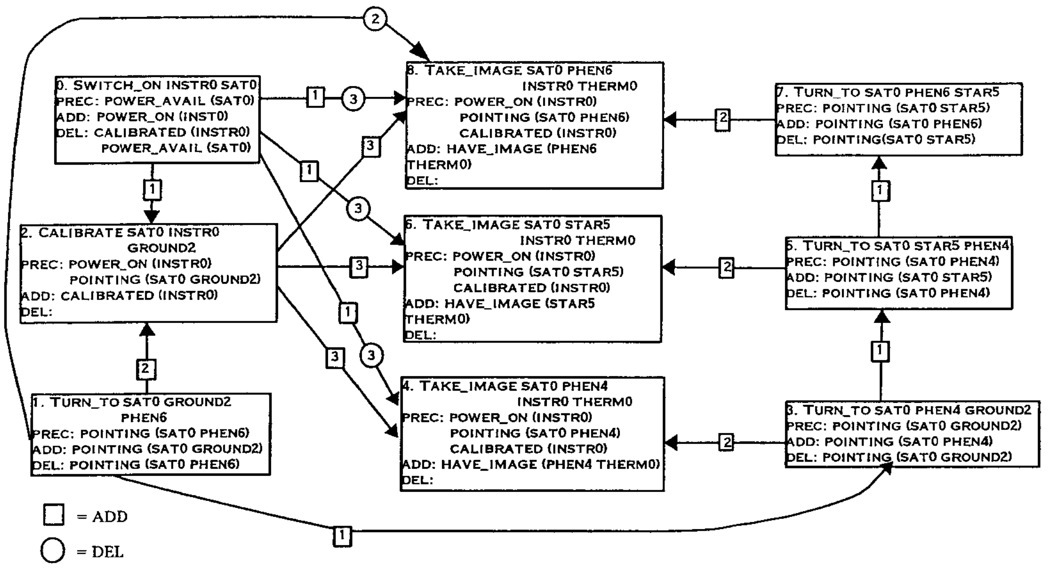
4.1 Solution Graph

This section describes how to build the solution graph for a problem, starting from the solution plan and exploiting the effects that an action has on the following plan sequence. First the discussion framework is set with some preliminary comments and definitions. Next a high-level description of the method, an example, and the algorithm in pseudo-code are presented.

In the general case, the solution of a planning problem consists of a partially ordered sequence of steps. When actions have conditional effects, a step in the plan should be a pair (state, action) rather than only an action. This allows for a precise determination of what effects a given action has in the local context. The implementation of this method handles domains with conditional effects in their actions and can be extended to partial-order plans. However, for simplicity, the following discussion assumes that the initial solution is a totally-ordered sequence of actions. When an action occurs several times in a solution, each occurrence is a distinct solution step.

To introduce the solution graph, the causal links in a solution have to be defined. Let $< a_1, ..., a_n >$ be a solution. A positive causal link between a_i and a_j by p, written as $a_i \xrightarrow{+p} a_j$, exists in the solution if: (1) $i < j$, (2) p is a precondition of a_j and a positive (add) effect of a_i, and (3) no a_k, $i < k < j$ adds p. A positive causal link is the same as a causal link in partial-order planning [73].

A negative causal link between a_i and a_j by p, written as $a_i \xrightarrow{-p} a_j$, exists if: (1) $i < j$, (2) p is a precondition of a_j and a negative (delete) effect of a_i,



0. SWITCH_ON INSTR0 SAT0
PREC: POWER_AVAIL (SAT0)
ADD: POWER_ON (INST0)
DEL: CALIBRATED (INSTR0)
POWER_AVAIL (SAT0)
2. CALIBRATE SAT0 INSTR0 GROUND2
PREC: POWER_ON (INSTR0)
POINTING (SAT0 GROUND2)
ADD: CALIBRATED (INSTR0)
DEL:
1. TURN_TO SAT0 GROUND2 PHEN6
PREC: POINTING (SAT0 PHEN6)
ADD: POINTING (SAT0 GROUND2)
DEL: POINTING (SAT0 PHEN6)
8. TAKE_IMAGE SAT0 PHEN6 INSTR0 THERM0
PREC: POWER_ON (INSTR0)
POINTING (SAT0 PHEN6)
CALIBRATED (INSTR0)
ADD: HAVE_IMAGE (PHEN6 THERM0)
DEL:
6. TAKE_IMAGE SAT0 STAR5 INSTR0 THERM0
PREC: POWER_ON (INSTR0)
POINTING (SAT0 STAR5)
CALIBRATED (INSTR0)
ADD: HAVE_IMAGE (STAR5 THERM0)
DEL:
4. TAKE_IMAGE SAT0 PHEN4 INSTR0 THERM0
PREC: POWER_ON (INSTR0)
POINTING (SAT0 PHEN4)
CALIBRATED (INSTR0)
ADD: HAVE_IMAGE (PHEN4 THERM0)
DEL:
7. TURN_TO SAT0 PHEN6 STAR5
PREC: POINTING (SAT0 STAR5)
ADD: POINTING (SAT0 PHEN6)
DEL: POINTING(SAT0 STAR5)
5. TURN_TO SAT0 STAR5 PHEN4
PREC: POINTING (SAT0 PHEN4)
ADD: POINTING (SAT0 STAR5)
DEL: POINTING (SAT0 PHEN4)
3. TURN_TO SAT0 PHEN4 GROUND2
PREC: POINTING (SAT0 GROUND2)
ADD: POINTING (SAT0 PHEN4)
DEL: POINTING (SAT0 GROUND2)
1
2
3
□ = ADD
○ = DEL

Figure 4.2: The solution graph for problem 1 in the Satellite benchmark.

and (3) no a_k, $i < k < j$ deletes p. $a_i \rightarrow a_j$ denotes that there exists a causal link (either positive or negative) from a_i to a_j.

For each step in the linear solution, a node in the solution graph is created. The graph edges model causal links between the solution actions. An edge between two nodes a_1 and a_2 is created if $a_1 \rightarrow a_2$. An edge has two labels: The ADD label is the (possibly empty) list of all facts p such that $a_1 \xrightarrow{+p} a_2$. Similarly, the DEL label is the list of negative causal links.

Figure 4.2 shows the solution graph for problem 1 in the Satellite benchmark. See Appendix B for details on Satellite. The graph has 9 nodes, one for each step in the linear solution. Each node contains a numerical label showing the step in the linear solution, the action name and arguments, the preconditions and the effects. Static preconditions are safely ignored: no causal link can be generated by a static fact, since such a fact is never part of an action's effects. Graph edges have their ADD labels shown as square boxes, and DEL labels as circles. Consider the edge from node 0 to node 8. Step 0 adds the first precondition of step 8, and deletes the third. Therefore, the ADD label of this edge is 1 (the index of the first precondition), and the DEL label is 3.

The pseudo-code for building a solution graph is given in Figure 4.3. A time upper bound is $O(L^2 \times m)$, where L is the length of the solution at hand, and m is the maximum number of preconditions in a step. The methods are in general self explanatory, and follow the high-level description provided before. The method *findAddActionId*(p, id, s) returns the most recent action before the current step *id* that adds precondition p. The method *addEdgeInfo*(n_1, n_2, t, f, g) creates a new edge between nodes n_1 and n_2 (if one didn't exist) and concatenates the fact f to the label of type $t \in \{ADD, DEL\}$. An integer *nodes*(a), used in method *buildNodes*, provides information extracted from the search tree generated while looking for a solution. A search tree has states as nodes and actions as transitions. For each action a in the tree, *nodes*(a) is the number of nodes expanded in the search right before exploring action a. The *node heuristic* (NH) associated with an instantiated macro sequence $m = a_1 ... a_k$ is defined as follows:

$$NH(m) = \mathit{nodes}(a_k) - \mathit{nodes}(a_1).$$

NH measures the effort needed to dynamically discover the given sequence at run-time. As shown in the next section, the node heuristic is used to rank macro-operators in a list.

In this work, preconditions of solution steps are given in a STRIPS-like fashion: they are conjunctions of positive facts. Hence it is important to show why this method works for ADL domains, where preconditions can be more complicated formulas (e.g., have disjunctions). The explanation is that an ADL operator can be compiled down into a set of STRIPS[1] operators. This compilation is part of the preprocessing that FF and other planners perform [42].

A brief analysis of the graph in Figure 4.2 reveals interesting insights about the problem and the domain structure. The action sequence TURN_TO TAKE_IMAGE occurs three times (between steps 3–4, 5–6, and 7–8), which takes 6 out of a total of 9 actions. For each occurrence of this sequence, there is a graph edge that shows the causal connection between the actions: applying op-

[1]In practice, the compiled operators are a little more general than STRIPS, since they can have conditional effects too.

```
void buildSolutionGraph(Solution $s$, Graph & $g$) {
   buildNodes($s$, $g$);
   buildEdges($s$, $g$);
}
void buildNodes(Solution $s$, Graph & $g$) {
   for (int $id = 0$; $id <$ length($s$); ++$id$) {
      Action $a =$ getSolutionStep($id$, $s$);
      addNode($id$, $a$, nodes($a$), $g$);
   }
}
void buildEdges(Solution $s$, Graph & $g$) {
   for (int $id = 0$; $id <$ length($s$); ++$id$) {
      Action $a =$ getSolutionStep($id$, $s$);
      for (each precondition $p \in$ Precs($a$)) {
         $id_{add}$ = findAddActionId($p$, $id$, $s$);
         if ($id_{add}$ != NO_ACTION_ID)
            addEdgeInfo($id_{add}$, $id$, ADD, $p$, $g$);
         $id_{del}$ = findDeleteActionId($s$, $id$, $p$);
         if ($id_{del}$ != NO_ACTION_ID)
            addEdgeInfo($id_{del}$, $id$, DEL, $p$, $g$);
      }
   }
}
```

Figure 4.3: Pseudo-code for building the solution graph.

erator TURN_TO satisfies precondition 2 of operator TAKE_IMAGE. In addition, the sequence SWITCH_ON TURN_TO CALIBRATE (steps 0–2) is important for repeatedly applying macro TURN_TO TAKE_IMAGE. This sequence establishes two of the preconditions of operator TAKE_IMAGE. The graph also shows that operator CALIBRATE should be applied between SWITCH_ON and TAKE_IMAGE. It restores the fact (CALIBRATED INSTR0), deleted by SWITCH_ON but needed by TAKE_IMAGE. Finally, there is no ordering constraint between SWITCH_ON and TURN_TO, so the ordering of the actions of this sequence is partial. SOL-EP performs this type of analysis to learn useful information about a domain.

4.2 Macro-Operators

A SOL-EP macro-operator is a structure $m = (O, \prec, \sigma)$, where O is a bag of domain operators, $\prec$ a partial ordering of the elements in O, and σ a mapping that defines the macro's variables from the operators' variables. In particular, if $\prec_t$ is a total ordering, then $m = (O, \prec_t, \sigma)$ is a sequential macro-operator. A domain operator can occur several times in O. A macro's preconditions and effects are not explicitly given. They are determined at run-time, when a macro is dynamically instantiated by applying its actions in sequence.

This section focuses on how SOL-EP learns and uses macro-operators. Generation, filtering, and run-time instantiation are discussed. A global set of candidate macros is generated from the solution graphs of several training problems. This set is reduced to a small number of selected macros, completing the learning phase. Finally, the selected macros are used to speed up planning in new problems.

4.2.1 Generating Macro-Operators

Macros are extracted from the solution graphs of one or more training problems. SOL-EP enumerates and selects subgraphs from the solution graph(s) and builds one macro for each selected subgraph. Two distinct subgraphs can produce the same macro. All generated macros are inserted into a global list that will later be filtered and sorted. The list contains no duplicates. When an extracted macro is already part of the global list, relevant information associated with that element is updated. The algorithm increments the number of occurrences, and adds the node heuristic $NH(mi)$ of the extracted instantiation mi.

Figure 4.4 presents the procedure for extracting macros from the solution graph. Parameters MIN_LENGTH and MAX_LENGTH limit the length l of a macro. The minimal size is trivial: each macro should have at least two actions. The upper bound is set to speed up macro generation. In the most general setup, the upper bound could be the plan length of the problem at hand, provided that the whole solution might be an useful macro. This is

```
void generateAllMacros(Graph $g$, MacroList & $macros$) {
  for (int $l$ = MIN_LENGTH; $l \leq$ MAX_LENGTH; ++$l$ )
    generateMacros($g$, $l$, $macros$);
}
void generateMacros(Graph $g$, int $l$, MacroList & $macros$) {
  selectSubgraphs($l$, $g$, $subgraphList$);
  for (each subgraph $s \in subgraphList$) {
    buildMacro($s$, $m$);
    int $pos$ = findMacroInList($m$, $macros$);
    if ($pos$ != NO_POSITION)
      updateInfo($pos$, $m$, $macros$);
    else
      addMacroToList($m$, $macros$);
  }
}
```

Figure 4.4: Pseudo-code for generating macros.

usually not the case in practice. This thesis focuses on identifying a few local patterns that are generally useful, rather than caching many complete solutions of solved problems.

In Figure 4.4, method *selectSubgraphs*$(l, g, subgraphList)$ finds *valid* subgraphs of size l of the original solution graph. It is implemented as a backtracking procedure that produces all the valid node combinations and prunes incorrect partial solutions early.

To describe the validation rules, consider a subgraph sg with l arbitrary distinct nodes $a_{m_1}, a_{m_2}, ..., a_{m_l}$. Node a_{m_i} is the m_i-th step in the linear solution $< a_1, ..., a_n >$. Assume that the nodes are ordered according to their position in the linear solution: $(\forall i < j) : m_i < m_j$. A subgraph sg is valid if:

- $m_l - m_1 + 1 \leq l + k$. The nodes of sg are obtained from a sequence of consecutive steps in the linear solution by skipping at most k steps, where k is a parameter. Skipping actions allows irrelevant actions to be ignored for the macro at hand. The upper bound k captures the heuristic that good macros are likely to have their steps "close" together in a solution. In the Satellite example, consider the subgraph with nodes

$\{0, 1, 2, 6\}$. For this subgraph, $l = 4$, $m_l = 6$, and $m_1 = 0$. The subgraph is invalid for $k = 2$, since $m_l - m_1 + 1 = 7 > 6 = l + k$, but it would be valid for $k \geq 3$.

- A valid subgraph must be connected, since two separated connected components are assumed to correspond to two independent macros. Consider the example in Figure 4.2. Nodes 2 and 3 do not form a valid subgraph, since there is no direct link between them, and therefore this subgraph is not connected. However, nodes 3 and 4 are connected through a causal link, so this subgraph will be validated.

- When selecting a subgraph, a solution step a_r $(m_1 < r < m_l)$ can be skipped only if a_r is not connected to the current subgraph: $(\forall i \in \{1, .., l\}) : \neg(a_{m_i} \rightarrow a_r \vee a_r \rightarrow a_{m_i})$.

Method *buildMacro*(s, m) in Figure 4.4 builds a partially ordered macro m based on the subgraph s. For each node of the subgraph, the corresponding action is added to the macro. At this step, actions are still instantiated: they have constant arguments rather than generic variables. After all actions have been added, all constant arguments are replaced with generic variables, obtaining a variable identity map σ. The partial order between the operators of the macro is computed using the positive causal links of the subgraph. If a positive causal link exists between two nodes a_i and a_j, then a precondition of action a_j was made true by action a_i. Therefore, action a_i should come before a_j in the macro sequence. The ordering has no cycles, since the ordering constraints are determined using a subgraph of the solution graph, and the solution graph is acyclic. A graph edge can exist from a_i to a_j in the solution graph only if $i < j$.

As an example, from the solution graph in Figure 4.2, 24 distinct macros are extracted. The largest contains all nodes in the solution graph. One macro occurs 3 times (TURN_TO TAKE_IMAGE), another twice (TURN_TO TAKE_IMAGE TURN_TO), and all remaining macros occur once.

The upper bound on complexity of generating macros of length l from a solution graph with L actions is

$$O\left(l^2 \times \binom{l+k}{l} \times L \times I\right).$$

The first factor is the cost to build one macro, where the number of ordering constraints can be quadratic on the number of steps. The second factor is the cost to enumerate all macros of length l within a window of size $l+k$ (i.e., a subgraph with $l+k$ nodes that are consecutive in the initial sequential solution – see the first validation rule). The window slides along the solution plan, obtaining the third factor (assume $l+k < L$). I is the cost to find/insert a macro into the global set of macros.

The value of k controls the trade-off between the speed of the algorithm and the number of enumerated subgraphs. A small k speeds up the computation but misses macros that are too widely spread over the solution sequence. In contrast, a large k increases the processing time exponentially, and allows enumerating subgraphs with nodes far away from each other. In experiments, a small value of k turned out to be a good trade-off: processing is fast, and most useful macro-occurrences are caught, since the steps of useful macros usually form a local sequence in the plan. However, for domains where useful macro-sequences are often spread out across a solution, increasing k can result in a much better set of macros.

4.2.2 Filtering and Ranking

Filtering addresses the utility problem. After all training problems have been processed, the global list of macros is *statically* filtered and sorted, so that only the most promising macros will be used to solve new problems. When the selected macros are used in future searches, they are further filtered *dynamically* by evaluating their run-time performance.

Static filtering uses an *overlap rule.* A macro is removed from the list when two occurrences of it overlap in a given solution. Consider the following sequence in a solution:

$$...a_1a_2...a_la_1a_2...a_la_1a_2...a_l...$$

Assume both $m_1 = a_1a_2...a_l$ and $m_2 = a_1a_2...a_la_1$ are macros. When m_1 is used in the search, applying this macro three times could be enough to discover the given sequence with little effort. Consider now using m_2 in the search. This macro cannot be applied twice in a row, since the first occurrence ends beyond the beginning of the next occurrence. The sequence $a_2...a_l$ in the middle has to be discovered with low-level search.

An important property of the overlap rule is the capacity to automatically limit the length of a macro. For example, $a_1a_2...a_l$ is kept in the final list, while larger macros such as $a_1a_2...a_la_1$ or $a_1a_2...a_la_1a_2$ are rejected. As an exception to the overlap rule, a macro that is a double occurrence of a small (i.e., of length 1 or 2) sequence will not be rejected. In Satellite, a macro such as (TURN_TO TAKE_IMAGE TURN_TO) is removed because of the overlap, but the macro (TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE), a double occurrence of a short sequence, is kept.

Macros are ranked according to the *total node heuristic* $TNH(m)$ associated with each macro m, with ties broken based on the occurrence frequency F. For a generic macro m in the list, $TNH(m)$ is the sum of the node heuristic values (NH) for all instantiations of that macro in the solutions of all training problems. The *average* node heuristic ANH and estimates the average search effort needed to discover an instantiation of this macro at run-time:

$$TNH(m) = ANH(m) \times F(m).$$

The total node heuristic is a robust ranking method, which combines two factors that influence the performance of a macro. Since TNH is proportional to F, it favors macros that occur frequently in the training set, and may be more likely to be applicable in the future. TNH directly depends on ANH, which evaluates the search effort that one application of the macro could save.

TNH depends on the search strategy. For instance, changing the order in which moves are considered in the search can potentially change the ranking in the macro list. How much the search strategy affects the ranking, and how a set of macros selected based on one search algorithm would perform in a different search algorithm are still open questions.

After ranking and filtering the list, only a few elements from the top of the list are kept for future searches. The precise number is not crucial, since the dynamic filtering process defined below further tunes its value. In the Satellite example, the selected macros are (SWITCH_ON TURN_TO CALIBRATE TURN_TO TAKE_IMAGE) and (TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE).

For dynamic filtering, the following values are accumulated for each macro m. $IN(m)$ is the number of search nodes in which at least one instantiation of m is applicable and is not rejected by the helpful macro pruning test. $IS(m)$ is the number of times when an instantiation of m occurs in a solution. The efficiency rate is

$$ER(m) = \frac{IS(m)}{IN(m)}.$$

Dynamic filtering evaluates each macro after solving a number of problems NP given as a parameter. If $IN(m) = 0$ or $ER(m) < T$ for a threshold T, m is removed from the list.

T's value was set based on the empirical observation that there is a gap between the efficiency rate of successful macros and the efficiency rate of macros that should be filtered out. The efficiency rate of successful macros has been observed to range roughly from more than 0.05 to almost 1.00. For inefficient macros, $ER < 0.01$. T is set to 0.03.

4.2.3 Instantiating Macros at Run-Time

A classical search algorithm expands a node by considering low-level actions that can be applied to the current state. In addition, SOL-EP adds to the successor list states that can be reached by applying a sequence of actions that compose a macro. This enhancement affects neither the completeness nor the correctness of the original algorithm. Completeness is preserved since no regular successors are deleted. Since macros have no explicit preconditions and effects, a run-time function verifies their correctness. Given a state s_0 and a sequence of actions $m = a_1a_2...a_k$, m can correctly be applied to s_0 if a_i can be applied to s_{i-1}, $i \in 0, ..., k$, where $s_i = \gamma(s_{i-1}, a_i)$ and $\gamma(s, a)$ is the state obtained by applying a to s.

In principle, the definition of a SOL-EP macro at the beginning of Section 4.2 allows that: (1) only part of the possible step orderings of a macro instantiation could be applicable in a state and (2) different orderings of the steps in a macro instantiation could result in different destination states. For the sake of efficiency, only one ordering of steps is considered when a macro is instantiated at run-time.[2] Searching for an ordering that corresponds to an instantiation applicable to a state is significantly more expensive, but it can succeed more often. How to best balance this trade-off is an open question.

Two heuristic methods, *helpful macro pruning* and *goal macro pruning*, are introduced with the goal of pruning macro instantiations that guide the search in a wrong direction. The next paragraphs discuss these heuristics in detail.

Helpful Macro Pruning

Helpful macro pruning uses the relaxed plan $RP(s)$ that FF performs for each evaluated state s, and hence is available at no additional cost. See Section 2.1.2 for details on RP. The relaxed plan is used to decide what macro-instantiations to select in a given state. Since actions from the relaxed plan are often useful in the real world, a selected macro and the relaxed plan should match partially or totally (i.e., have common actions). To formally define the matching, consider a macro $m(v_1, ..., v_n)$, where $v_1, ..., v_n$ are variables, and an instantiation $m(c_1, ..., c_n)$, with $c_1, ..., c_n$ constant symbols, applicable in s. $Match(m(c_1, ..., c_n), RP(s))$ is the number of actions present in both $m(c_1, ..., c_n)$ and $RP(s)$.

If total matching is required (i.e., each action of the macro is mapped to an action in the relaxed plan) then it will often happen that no instantiation can be selected, since the relaxed plan can be too optimistic and miss actions needed in the real solution. On the other hand, a loose matching can significantly increase the number of selected instantiations, with negative effects on the overall performance of the search algorithm. The solution is to select only those instantiations which have the best matching seen so far for the given

[2]No particular ordering is preferred. In the implemented system, this is the ordering of the first instantiation discovered in the solutions of the training problems, but this is just an implementation detail.

macro in the given domain. A macro instantiation is selected only if

$$Match(m(c_1, ..., c_n), RP(s)) \geq MaxMatch(m(v_1, ..., v_n)). \qquad (4.1)$$

$MaxMatch(m(v_1, ..., v_n))$ is the largest value of $Match(m(c'_1, ..., c'_n), RP(s'))$, with $m(c'_1, ..., c'_n)$ applicable in s', that has been encountered so far in that domain.

Experiments show that $MaxMatch(m(v_1, ..., v_n))$ quickly converges to a stable value. In the Satellite example, MaxMatch(SWITCH_ON TURN_TO CALIBRATE TURN_TO TAKE_IMAGE) converges to 4, and MaxMatch(TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE) converges to 3.

If necessary, helpful macro pruning could be further refined to allow different values of *Match* for different instantiations of the same macro-operator. Assume a single large match sets *MaxMatch* to such a high value that no further matches can be made. A possible solution would be to replace the condition 4.1 by the following two-phase rule: Given a state s and a macro m, first build the set M of all instantiations $m(c_1, ..., c_n)$ applicable in s for which

$$Match(m(c_1, ..., c_n), RP(s)) \geq T_m,$$

where T_m is a threshold. Then, keep only instantiations

$$mi \in \text{argmax}_{mi' \in M} Match(mi', RP(s)).$$

Goal Macro Pruning

As shown in Section 2.1.2, FF implements two search algorithms. Planning starts with Enforced Hill Climbing (EHC), a fast but incomplete algorithm. When EHC fails to find a solution, the search restarts with a complete Best First Search (BFS).

In experiments it was observed that, in domains where macros are very successful, EHC is robust enough to solve problems and BFS does not need to be called. As shown in Section 5, such domains include Promela Dining Philosophers, Promela Optical Telegraph and Satellite. In this scenario, helpful macro pruning is powerful enough to filter instantiations of macros at runtime.

In domains such as Power Supply Restoration (PSR), where the benefits of macros are more limited, problem instances can be hard for both EHC and BFS. See Appendix B for a brief description of PSR. Often in this domain, EHC quickly fails and most of the search effort is spent in BFS. For such hard problems, macros can have either positive or negative effects on the overall planning effort.

Goal macro pruning is a very selective heuristic used in BFS in order to reduce the fluctuations in the performance of macros. Goal macro pruning keeps a macro instantiation only if the number of satisfied goal conditions is greater in the destination state as compared to state where the macro is applied. In BFS, an instantiated macro has to pass both the helpful macro pruning and the goal macro pruning tests.

4.2.4 Discussion

Desirable properties of macros and trade-offs involved in combining them into a filtering method are discussed in [64]. The authors identify five factors that can be used to predict the performance of a macro set. The next paragraphs briefly introduce these factors and discuss how SOL-EP deals with each of them.

TNH includes the first two factors ("the likelihood of some macro being usable at any step in solving any given planning problem", and "the amount of processing (search) a macro cuts down"). Factor 3 ("the cost of searching for an applicable macro during planning") mainly refers to the additional cost per node in the search algorithm. At each node, and for each macro, it must be tested if instantiations of the macro are applicable to the current state, and satisfy the macro pruning tests. The costs are greatly reduced by keeping only a small list of macros, but there often is an overhead as compared to searching with no macros. No special care is taken of factor 4 ("the cost (in terms of solution non-optimality) of using a macro"). Chapter 5 contains an empirical analysis of factors 3 and 4.

Factor 5 refers to "the cost of generating and maintaining the macro set". In SOL-EP, the costs to generate macros include, for each training problem, solving the problem instance, building the solution graph, extracting macros from the solution graph, and inserting the macros into the global list. The only maintenance operations that SOL-EP performs are to dynamically filter the list of macros and to update *MaxMatch* for each macro, which need no significant cost.

4.3 Participating in the International Planning Competition

FF was enhanced with macros and implementation enhancements to reduce memory and CPU time requirements. The resulting program, Macro-FF [6, 9], competed in the fourth international planning competition IPC-4 [40]. An overview of FF was presented in Section 2.1.2. This section describes the features of Macro-FF and summarizes its participation in the competition.

4.3.1 Macro-FF

Macros in the Competition System

Macro capabilities are based on a version of the SOL-EP model, which has broader applicability than CA-ED. Since, at the competition time, SOL-EP was not fully developed as described in the previous sections, a preliminary version was implemented in the competition system. The differences between the preliminary version and the final version of SOL-EP are summarized next.

Macros in the competition system were limited to only 2 actions. This choice removed many challenges that have to be solved for arbitrary-length macros. Operations such as building and processing a solution graph, and run-time macro instantiation are considerably simplified. Another difference is in the ranking method (see discussion below). The ranking method based on a node heuristic had not been developed at the competition time.

The competition system used a first implementation of the solution graph, where causal edges can be defined only between two consecutive actions of a plan. Hence only sequences of two consecutive actions can be considered as possible macros. Local chaining is enforced by the rule that the actions a_1 and

a_2 of a macro should have at least one common variable, unless a_1 and/or a_2 have no parameters. Macros with null effects are discarded.

Run-time macro instantiation uses helpful macro pruning. A macro instantiation is used only if both its actions are part of the relaxed plan computed for the current state. The competition system implemented neither goal macro pruning nor dynamic macro filtering based on efficiency rate.

For ranking, macro-operators are stored in a global list ordered by their *weight*, with smaller being better. Weights are initialized to 1.0 and updated using a gradient-descent method.

For each macro-operator m extracted from the solution of a training problem, the problem is re-solved with m in use. Let L be the solution length when no macros are used, N the number of nodes expanded to solve the problem with no macros, and N_m the number of expanded nodes when macro m is used. The difference $N - N_m$ is used to update w_m, the weight of macro m. Since $N - N_m$ can take arbitrarily large values, it is mapped to a new value in the interval $(-1, 1)$ by

$$\delta_m = \sigma\left(\frac{N - N_m}{N}\right)$$

where

$$\sigma(x) = \frac{2}{1 + e^{-x}} - 1$$

is the sigmoid function shown in Figure 4.5, centered in $(0, 0)$ and bounded to $(-1, 1)$. In particular, the symmetry property ensures that, if $N_m = N$, the weight update of m at the current training step is 0. The size of the boundary interval has no effect on the ranking procedure, it only scales all weight updates by a constant multiplicative factor. The interval $(-1, 1)$ is used as a canonical representation, which limits the absolute value of δ_m between 0 and 1.

The update formula also contains a factor that measures the difficulty of the training instance. The harder the problem, the larger the weight update should be. The reason is that macros that appear to be effective in very simple training instances could in fact be useless for larger problems. The difficulty factor is estimated by the solution length L rather than N, since the former

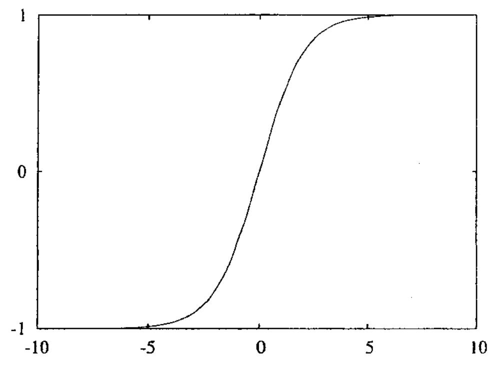


Figure 4.5: Sigmoid function.

has a smaller variance over a training problem set. The formula for updating w_m is

$$w_m = w_m - \alpha\delta_m L$$

where $\alpha = 0.001$. The value of α does not affect the ranking of macros. It is used only to keep macro weights within the vicinity of 1.

Implementation Enhancements

The enhancements described below have the goal of reducing the space and CPU requirements of the planner, and affect neither the number of expanded nodes nor the quality of found plans. However, when the memory or CPU time necessary to solve a problem are larger than the available resources, these improvements can make the difference between failure and success in solving a problem. The three enhancements described below affect the open queue in best-first search, the memory requirements in the preprocessing phase, and state hashing. The first two were implemented by Enzenberger [9].

The open queue in BFS was originally implemented as a single linked list. This was changed into a linked list of buckets, one bucket for each heuristic value. The time to find/insert an element reduces from linear in the total number of states in the list to linear in the number of different heuristic values of the states in the list. The buckets are implemented as linked lists and need constant time for insertion, since they no longer have to be sorted.

As part of preprocessing, FF builds a lookup table with all facts of the initial state. In the original implementation, this table is sparsely populated but the allocated memory is equal to the number of constants to the power of the arity of each predicate summed over all predicates in the domain. The lookup table was replaced by a balanced binary tree with minimal memory requirements and a lookup time proportional to the logarithm of the number of facts in the initial state.

In the original implementation of FF's state hashing, each fact of a planning problem is assigned a unique 32-bit random number. The hash code of a problem state is the sum of all random numbers associated with the facts that characterize the given state. When two states have the same hash code, a full fact-by-fact comparison checks whether the states are identical. The original implementation was replaced by a 64-bit Zobrist hashing [63]. Facts are assigned 64-bit random numbers, and the hash key of a state is obtained by computing the XOR of all random numbers corresponding to the facts true in the state. The larger size of the hash key and the better randomization make hash conflicts so improbable that full state comparison is no longer necessary.

4.3.2 Competition Results

The fourth international planning competition IPC-4 had a classical part and a probabilistic part. For detailed information about the classical part, including domain description and results in each domain, see [22, 40]. As shown in [40], 19 competitors participated in the classical part (21 if all system versions are counted).

The organizers performed the ranking of systems by hand, since it is hard to obtain a meaningful ranking using strict formal rules. A distinct ranking was performed for each *version* (i.e., temporal, non-temporal, numeric, etc.) of each domain. Moreover, optimal and satisficing (suboptimal) planners were evaluated separately. In each domain version, the ranking was based on a visual analysis of the data charts. Apparently, the CPU time mattered the most. For each planner, a performance record (N_1, N_2) was computed. N_1 counts how many times that planner took first place, and N_2 is the number of second-place rankings.

Macro-FF entered the classical part and competed in the following seven domains: Promela Dining Philosophers – ADL (containing a total of 48 problems), Promela Optical Telegraph – ADL (48 problems), Satellite – STRIPS (36 problems), PSR Middle Compiled – ADL (50 problems), Pipesworld Notankage Nontemporal – STRIPS (50 problems), Pipesworld Tankage Nontemporal – STRIPS (50 problems), and Airport – ADL (50 problems). See Appendix B for details on these benchmarks. The performance record of Macro-FF was $(3, 0)$. It took the first place in Promela Optical Telegraph, Satellite (tied with YAHSP [88]), and PSR Middle Compiled.

4.4 Conclusions and Future Work

This chapter presented SOL-EP, a technique that automatically learns a small set of macro-operators from previous experience in a domain, and uses them to speed up the search in future problems.

Exploring this method more deeply and improving the performance in more classes of problems are major directions for future work. Also, the learning method could possibly be generalized from macro-operators to more complex structures such as hierarchical task networks. Little research focusing on learning such structures has been conducted, even though the problem is of great importance.

Another interesting topic is to use macros in the graphplan algorithm, rather than the current framework of planning as heuristic search. The motivation is that a solution graph can be seen as a subset of the graphplan associated to the initial state of a problem. Since SOL-EP learns common patterns that occur in solution graphs, it seems natural to try to use these patterns in a framework that is similar to solution graphs.

Chapter 5

Experiments in AI Planning

This chapter presents experiments that evaluate the CA-ED and SOL-EP abstraction techniques. All experiments were run on a AMD Athlon 2 GHz machine. The experiments were designed at the standards of the international planning competition. As in IPC-4, the time and memory resources are limited to 30 minutes and 1 GB of memory for each problem. All domains used as testbeds were used in either IPC-3 or IPC-4 or both. For most domains, the problem sets are the same as in the competition. The exceptions are Satellite and Rovers, where additional problems were generated on top of the competition problem sets. More details are provided for each experiment in its corresponding section.

Section 5.1 analyzes and compares CA-ED and SOL-EP in a common experimental framework. Section 5.1.1 provides an empirical evaluation of how CA-ED macros can affect the heuristic evaluation of states and the depth of goal states. Section 5.2 focuses on the performance of the competition system, which implements the preliminary version of SOL-EP described in Section 4.3. Section 5.3 includes an analysis of the full-scale SOL-EP and a comparison between the preliminary and the full-scale versions of SOL-EP. Section 5.4 concludes Chapters 3, 4 and 5, dedicated to AI planning research.

5.1 Evaluating CA-ED vs. SOL-EP

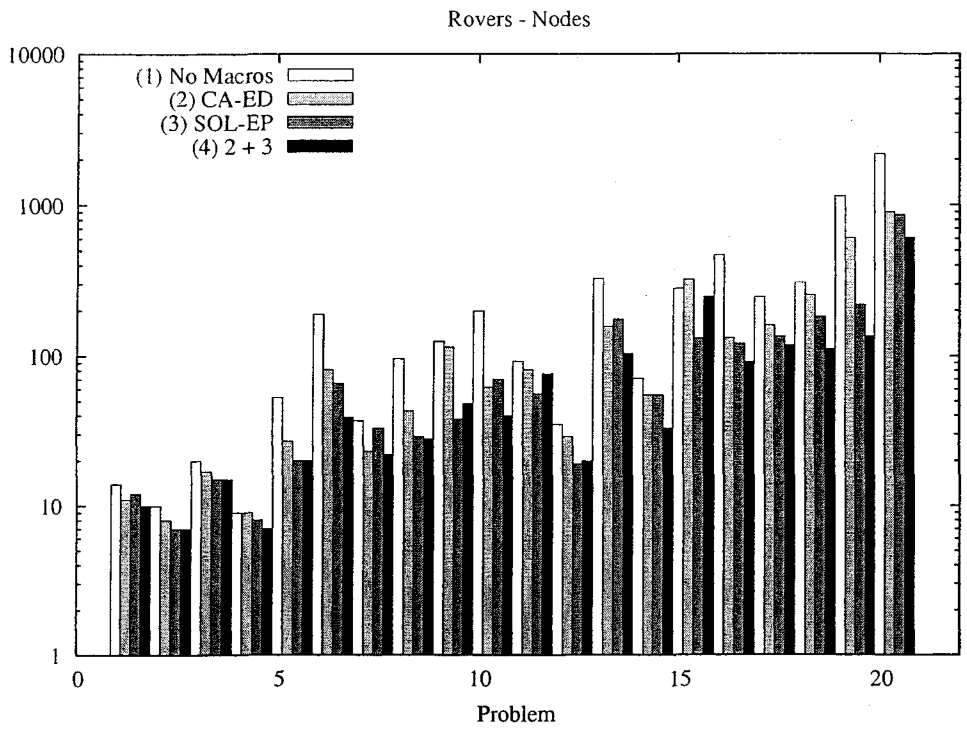
This section compares classical planning, CA-ED, and SOL-EP. The analysis is restricted to match CA-ED's constraints: STRIPS domains with static facts are used, and the size of macros is limited to only 2 actions. Therefore, this section is not a detailed analysis of SOL-EP, which was included mainly for comparison reasons.

Four program setups are compared in this experimental evaluation. Setup 1 is the planner FF with no macros. Setup 2 is FF + CA-ED, the method described in Chapter 3. Setup 3 is FF + SOL-EP, the method described in Chapter 4. The preliminary version of SOL-EP is used in this experiment, so that both CA-ED and SOL-EP limit the size of macros to 2 actions. Setup 4 is a combination of 2 and 3. Since both methods have benefits and limitations, it is interesting to analyze how they perform when applied together. In setup 4, first CA-ED is applied, obtaining an enhanced domain. Next this is treated as a regular domain, and SOL-EP is applied to generate a list of macros with incomplete definitions. Finally, the enhanced planner uses as input the enhanced domain, the list of macros, and regular problem instances.

Since CA-ED can be applied only to STRIPS domains with static facts in their formulation, Rovers, Depots and Satellite, which satisfy these constraints, were used as testbeds. All three domains were used in the third international planning competition. Satellite was re-used in the fourth edition with a larger problem set.

The set of 22 Depots problems used in the third competition contains both easy and hard instances, so no problems were added for this experiment. The Rovers and Satellite problem sets used in the third competition can easily be solved by FF. Hence, for the experiments reported below, they were extended from 20 to 40 problems each [13]. The additional instances were created with the same problem generator as for the competition. The generator takes as parameters the number of objects of each type, the number of goals, and the value of the random seed. In addition, the fourth competition included 16 new Satellite problems. These were added, obtaining a final set of 56 Satellite problems. Problems 1–36 are from the fourth competition, and problems 37–56 were additionally generated.

Figures 5.1–5.5 and Table 5.1 summarize the results. Each figure shows the number of expanded nodes and the CPU time for one domain on a logarithmic



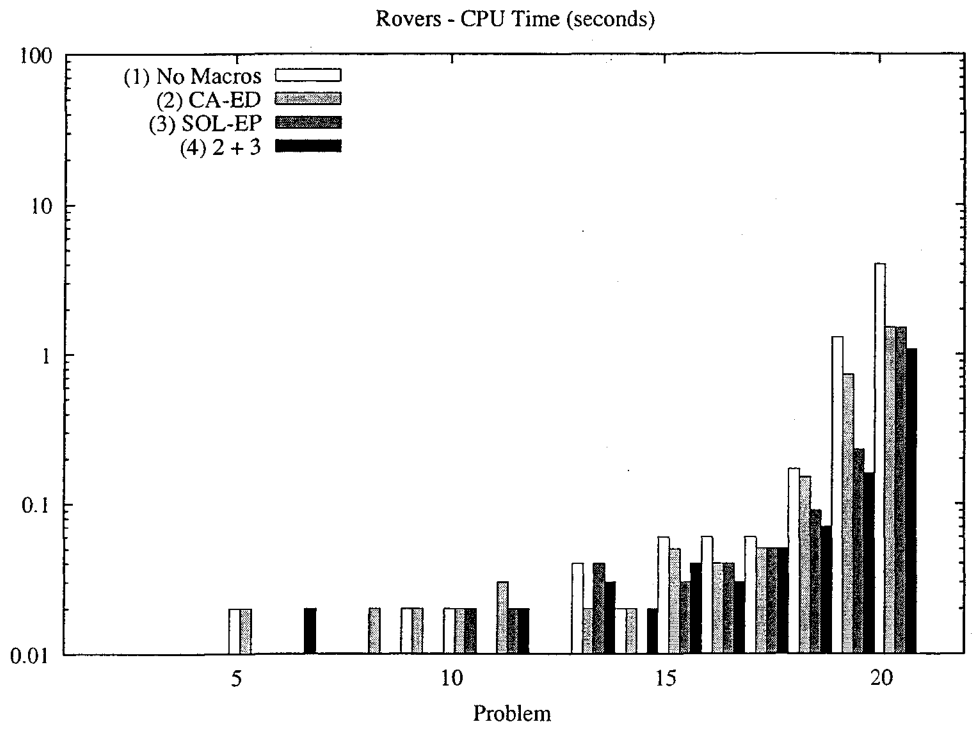
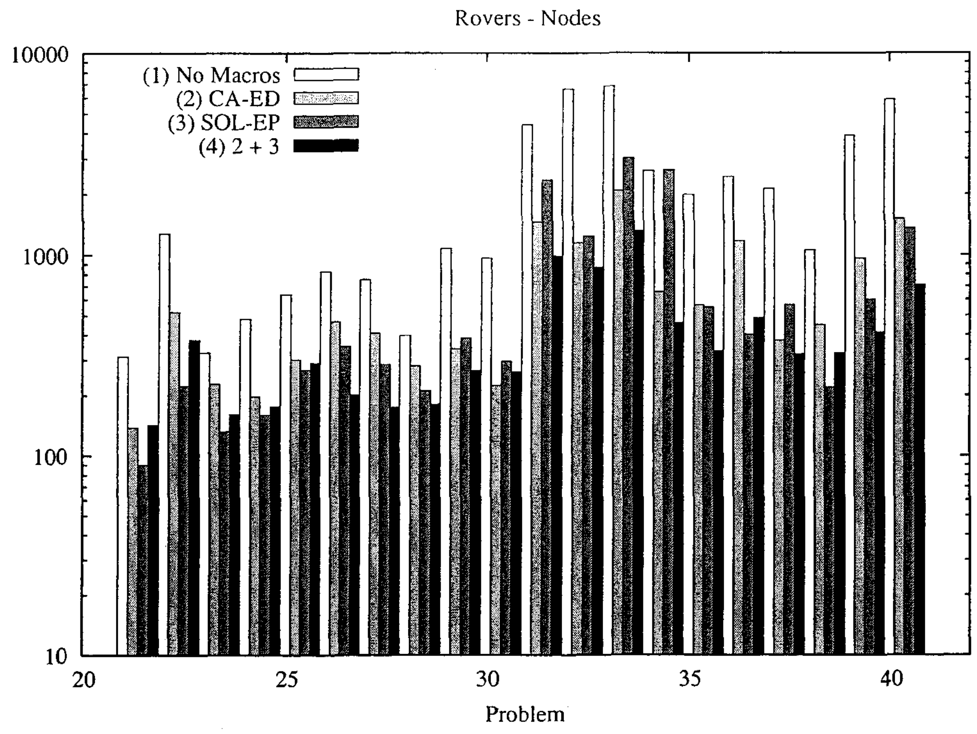


Figure 5.1: Evaluating abstraction techniques in Rovers. Problems 1–20 are shown.



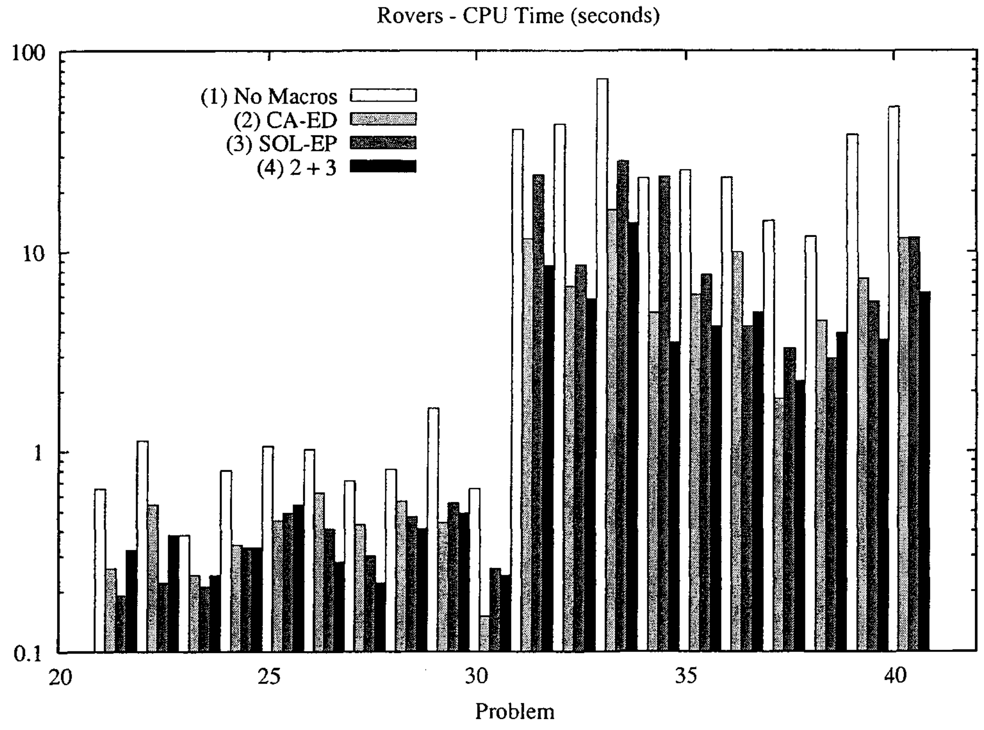
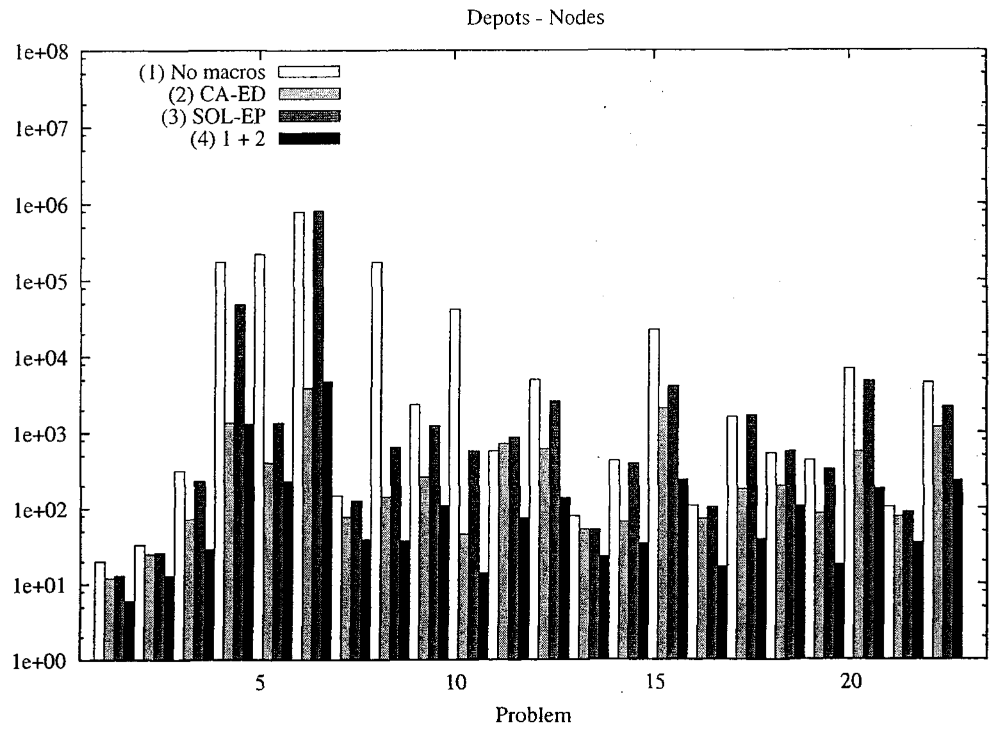


Figure 5.2: Evaluating abstraction techniques in Rovers. Problems 21–40 are shown.



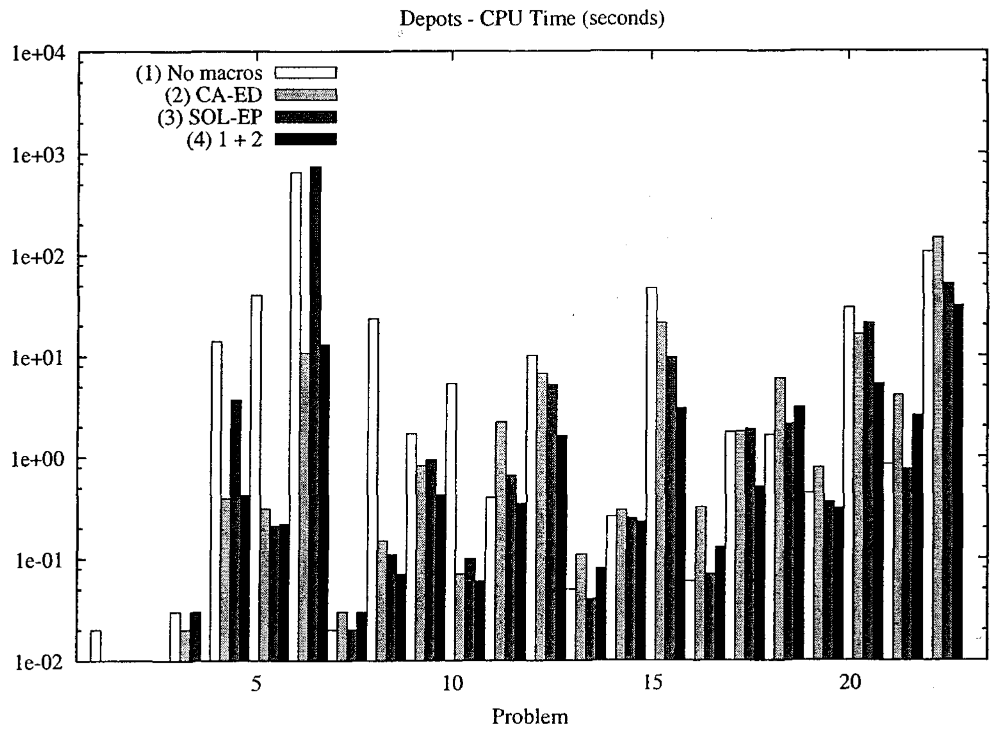
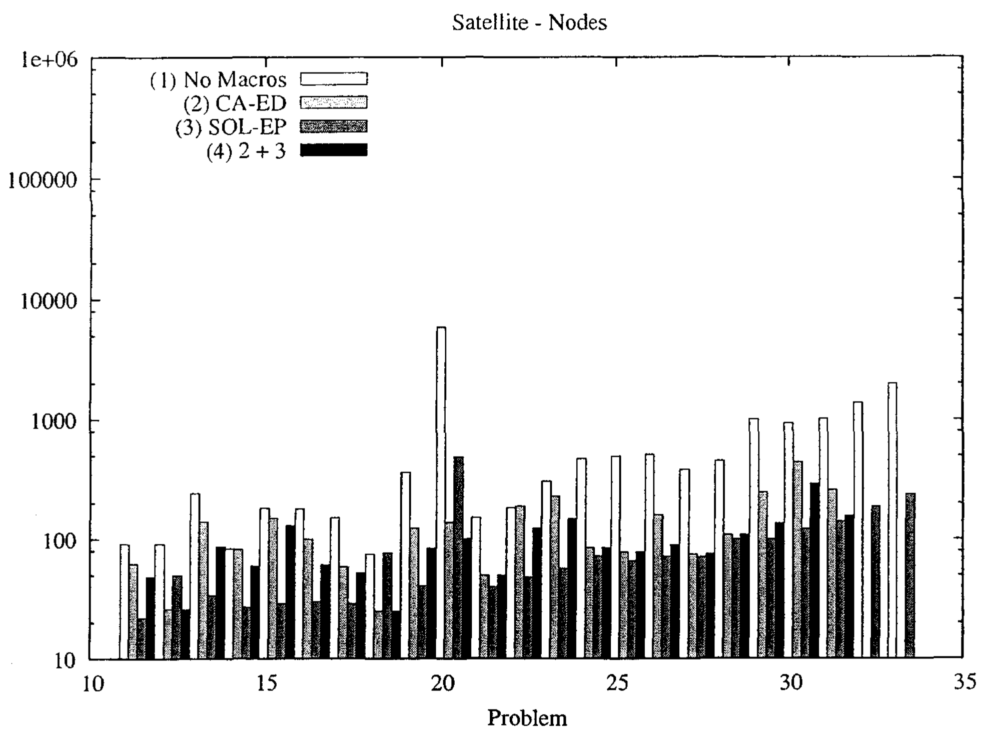


Figure 5.3: Evaluating abstraction techniques in Depots.



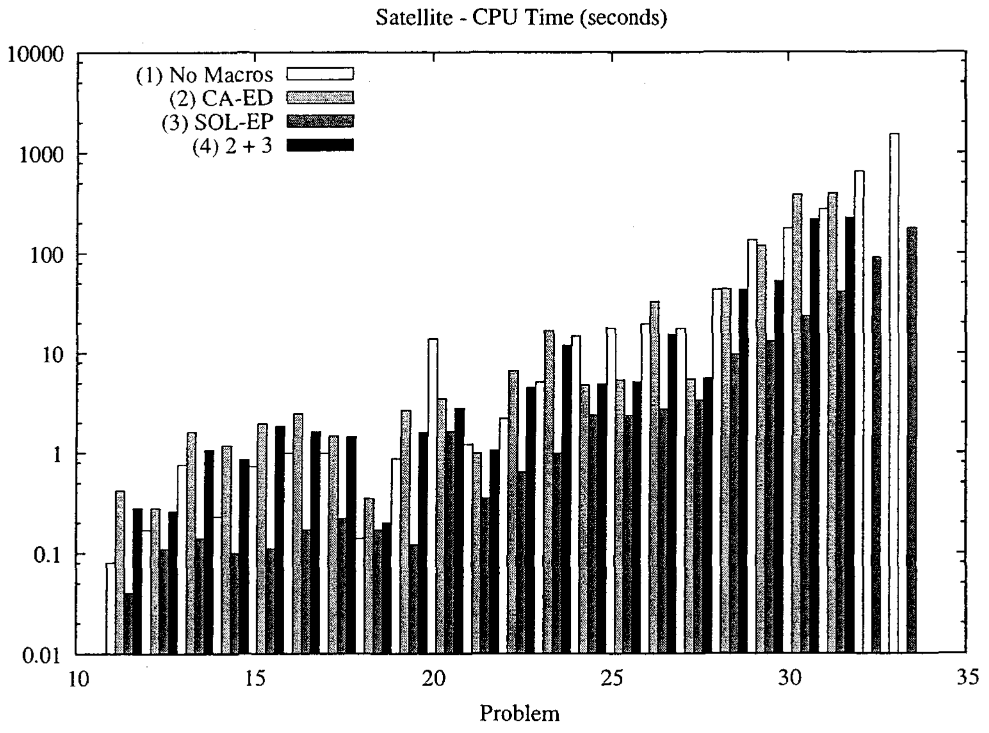
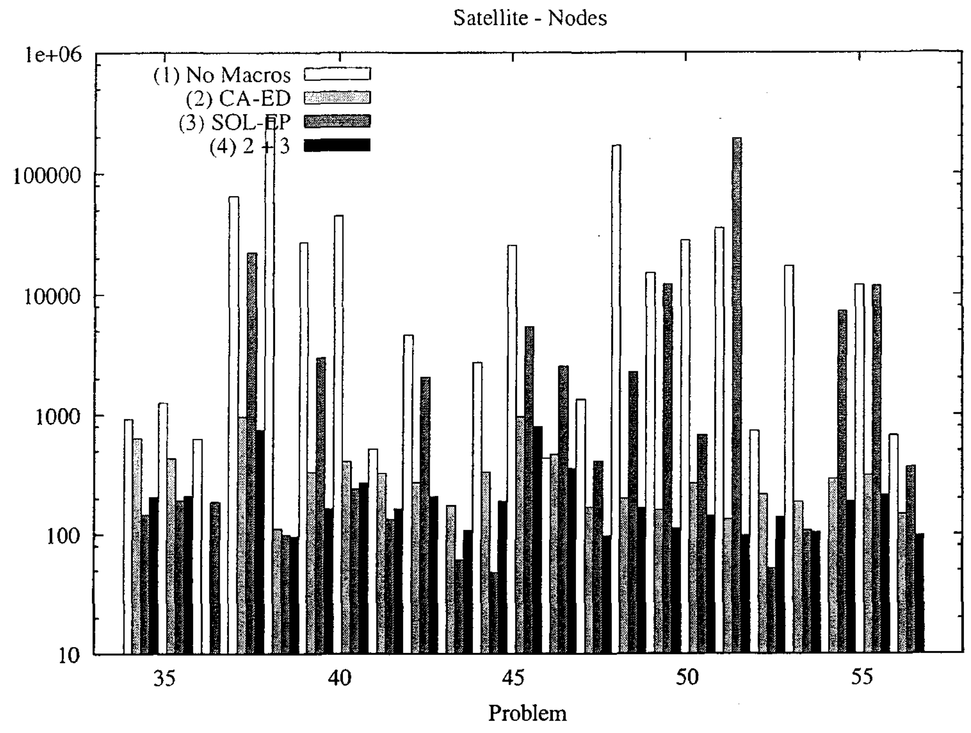


Figure 5.4: Problems 11–33 are shown.



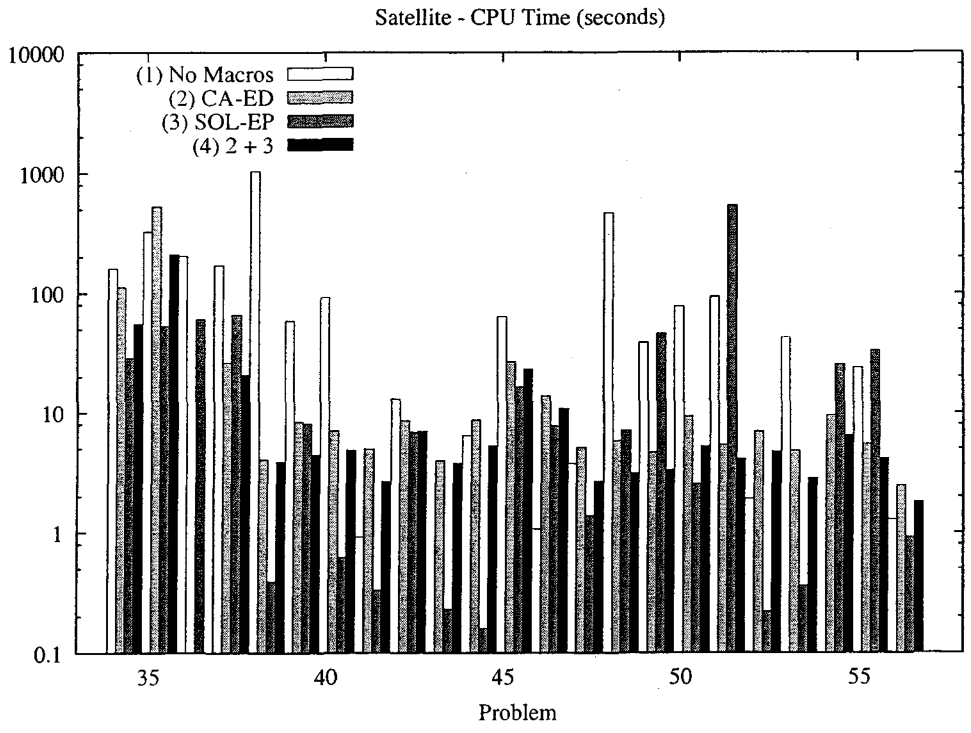


Figure 5.5: Evaluating abstraction techniques in Satellite. Problems 34–56 are shown.

scale. The results show consistent performance improvement when macros are used. Interestingly, combining CA-ED and SOL-EP often leads to better performance than each abstraction method taken separately. In Rovers, all three abstraction setups produce quite similar results, with a slight plus for the combined setup. In Depots, CA-ED is more effective than SOL-EP in terms of expanded nodes. The differences in CPU time become smaller, since adding new operators to the original domain significantly increases the cost per node in Depots (see the discussion below). Again, the overall winner in this domain is the combined setup. In Satellite, adding macros to the domain reduces the number of expanded nodes, but has significant impact on cost per node (see Table 5.1) and memory requirements. Note that setups 2 and 4, which add macros to the original domain, fail to solve three problems (32, 33, and 36) because of large memory requirements in the preprocessing phase. The classical system fails on two problems (43 and 54), so SOL-EP is the only system that solves all Satellite problems.

Domain	CA-ED vs. No Macros			SOL-EP vs. No Macros		
	Min	Max	Avg	Min	Max	Avg
Depots	3.27	8.56	6.06	0.93	1.14	1.04
Rovers	0.70	0.90	0.83	0.85	1.14	1.00
Satellite	0.98	14.38	7.70	0.92	1.48	1.11

Table 5.1: Rate of cost per node.

The average cost per node in search for a problem p is defined as

$$CPN(p) = \frac{Time(p)}{Nodes(p)}, \tag{5.1}$$

where $Nodes(p)$ is the number of expanded nodes, and $Time(p)$ is the search time. CPN is mostly determined by the relaxed graphplan algorithm that computes the heuristic value of states. Note that using a domain-independent heuristic comes with a high price in terms of cost per node. In Figures 5.1–5.5, a typical value for planning speed is in the range of 1,000 nodes per second. In other AI applications, where much faster heuristics can be used, algorithms such as A* could expand 1,000,000 nodes per second.

The total cost of the relaxed graphplan algorithm for a state is a combination of three main factors: (1) the cost to build one graphplan level, (2) the number of levels, and (3) the cost to extract a relaxed plan after all levels have been built. Factor 1 depends upon the number of actions that have been instantiated during preprocessing for a given problem, which in turn depends on both the number of operators and the number of objects defined for a problem. See Section 3.3.1 for details.

Table 5.1 evaluates how macros can affect the cost per node in search. A value in the table is the cost per node in the corresponding setup (i.e., CA-ED or SOL-EP) divided by the cost per node in the setup with no macros. For each of the two methods the minimum, the maximum, and the average value are shown.

The right part of the table shows relatively small values and variation for the cost rate in SOL-EP. In contrast, the cost rate in CA-ED shows both small and large values, varying both inside the same domain (e.g., Satellite) and across domains (e.g., from Rovers to Satellite).

In CA-ED, using macros as new operators often reduces the number of levels in the relaxed graphplan of a state s, but never increases it. Also, relaxed plans with macros are usually shorter, so extracting a relaxed plan after the relaxed graphplan has been built can be faster.

In contrast, building one graphplan level can be more expensive. Adding new operators to a domain increases the number of preinstantiated actions. Since macros tend to have more variables than a regular operator, the corresponding number of instantiations can be large. Let the *action instantiation rate* be the number of actions instantiated for a problem when macros are added to a domain (CA-ED) divided by the number of actions instantiated in the original domain formulation. Experiments show that the average action instantiation rate is 6.03 in Satellite, 3.20 in Depots, and 1.04 in Rovers.

The way these effects of macros combine determines how large the cost rate is. In Satellite and Depots, it can reach relatively high values, showing that the overhead of factor 1 (building one level) is higher than the savings of factors 2 and 3. This is not surprising, given the relatively high instantiation rates of these domains. On the other hand, the cost rate is less than 1 in CA-ED Rovers, showing that, in this domain, factors 2 and 3 are dominant.

Figure 5.2 shows a big jump in the CPU time starting with problem 31, but a smaller increase in terms of expanded nodes. The explanation is that the cost per node is significantly higher for the last 10 problems of this dataset, as shown next. Since the last 10 problems were generated with a larger number of objects than the previous ones, the number of instantiated actions is larger. As explained before, building graphplan levels gets more expensive as more actions are present, so the cost of processing a node is higher.

The experiments show no significant impact of macro-operators on the solution quality. When macros are used, the length of a plan slightly varies in both directions, with an average close to the value of the classical system.

5.1.1 Effects of CA-ED Macros on Search

As shown in Section 3.3.1, macros added to a domain as new operators affect both the structure of the search space (the embedding effect) and the heuristic evaluation of states with relaxed graphplan (the evaluation effect). This section presents an empirical analysis of these.

Figure 5.6 shows results for Depots, Rovers and Satellite. For each domain, the chart on the left shows data for the original domain formulation, and the chart on the right shows data for the macro-enhanced domain formulation. For each domain formulation, the data points are extracted from solution plans as follows. Each state along a solution plan generates one data point. The coordinates of the data point are the state's heuristic evaluation on the vertical axis, and the number of steps left until the goal state is reached on the horizontal axis. Note that the number of steps to a goal state may be larger than the distance (i.e., length of shortest path) to a goal state. The reason why states along solution plans were used to generate data is that for such states, both the heuristic evaluation, and the number of steps to a goal state are available after solving a problem.

The first conclusion from Figure 5.6 is that macros added to a domain improve the accuracy of heuristic state evaluation of relaxed graphplan. The closer a data point is to the diagonal, the more accurate the heuristic evaluation of the corresponding state. In each of the three domains, the data cloud on the right, obtained with macro-operators, is clearly closer to the diagonal than its correspondant on the left.

As a second conclusion, the projection of data clouds on the horizontal axis is shorter in macro-enhanced domains. This is a direct result of the embedding effect: since each macro counts as one step, the distance from a state to a goal state becomes shorter. The largest reduction is in Satellite, where the length of the projection (i.e., the number of steps in the longest solution) drops by a factor of two. A similar situation is observed in Depots, where the projection reduces by about 40%. For the data points (states) corresponding to the Rovers domain, a typical reduction is within the range of 10–20%.

5.2 Evaluating the Competition System

This section evaluates the preliminary version of SOL-EP that was used in the planning competition IPC-4. The seven domains that Macro-FF competed in as part of IPC-4, shown in Section 4.3.2, are used as testbeds.

Planning with macros (Macro-FF) is compared against classical planning (FF). Both planners contain the implementation enhancements reported in Section 4.3.1 that deal with open queue management, state hashing, and pre-processing. The new 64-bit state hashing is especially effective in the PSR and Promela Dining Philosophers domains. Figure 5.7 shows a speedup of up to a factor of 2.5. As a result, 3 more problems were solved in PSR, contributing to Macro-FF's success in this domain.

In Figures 5.8–5.10, the number of expanded nodes and the total CPU time are shown for each of the seven domains on logarithmic scales. A CPU time chart shows no distinction between a problem solved very quickly (within a time close to 0) and a problem that could not be solved. To determine what the case is, check the corresponding node chart, where the absence of a data point always means no solution.

Figure 5.8 summarizes the results in Satellite, Promela Optical Telegraph,

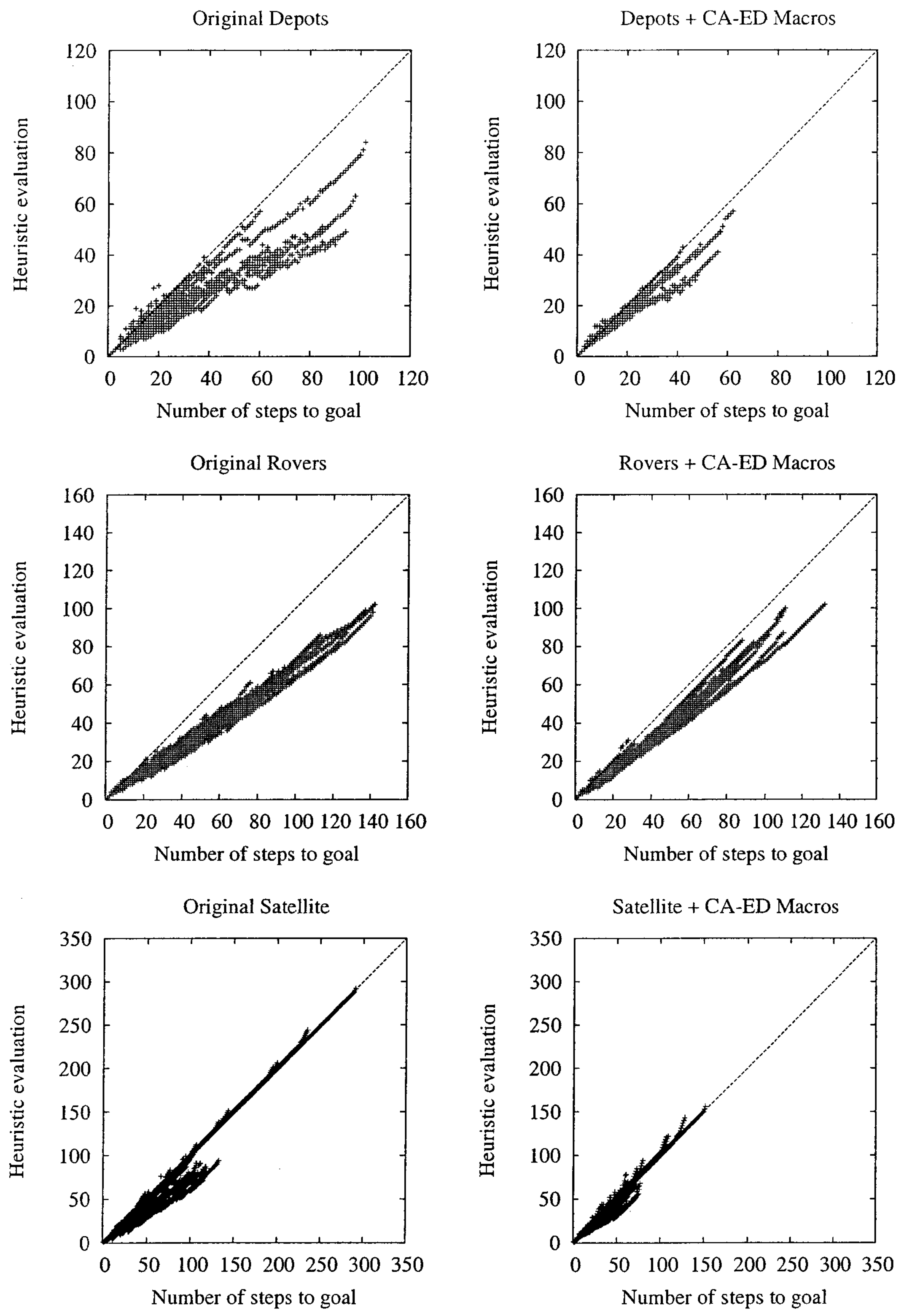
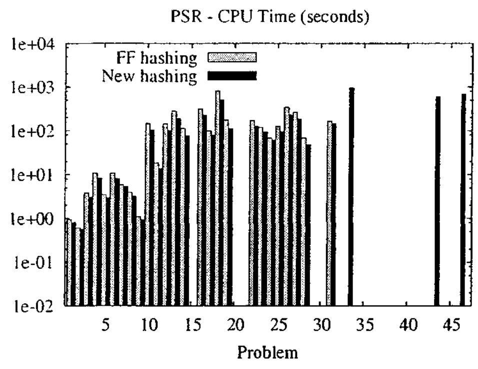


Figure 5.6: Effects of CA-ED macros on heuristic state evaluation and depth of goal states.



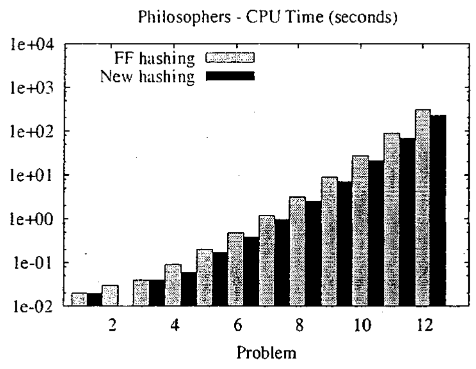


Figure 5.7: Comparison of the two implementations of state hashing in PSR (left) and Promela Dining Philosophers (right).

and Promela Dining Philosophers. In Satellite and Promela Optical Telegraph, macros greatly improve performance over the whole problem set, allowing Macro-FF to win these domain formulations in the competition. In Promela Optical Telegraph macros led to solving 12 additional problems. The savings in Promela Dining Philosophers are limited, resulting in one more problem being solved.

In Satellite and Promela Optical Telegraph, the CPU time grows faster than the number of expanded nodes as problem instances become larger, showing that state evaluation with relaxed graphplan is more expensive in large problems. The explanation is the following: Larger problems typically are characterized by more objects and/or longer solutions. As shown in Section 5.1, more objects result in more instantiated actions, which in turn increase the cost to build one graphplan level. Long solutions suggest an increased average number of steps from a state s in the search space to a goal state s_G. Unless the heuristic is of very poor quality, this often results in a longer relaxed plan of s, which is more expensive to compute.

Figure 5.9 shows the results in the ADL version of Airport. The savings in terms of expanded nodes are significant, but they have little effect on the total running time. In this domain, the preprocessing costs dominate the total running time. The preprocessing also limits the number of solved problems to 21. The planner can solve more problems when the STRIPS version of

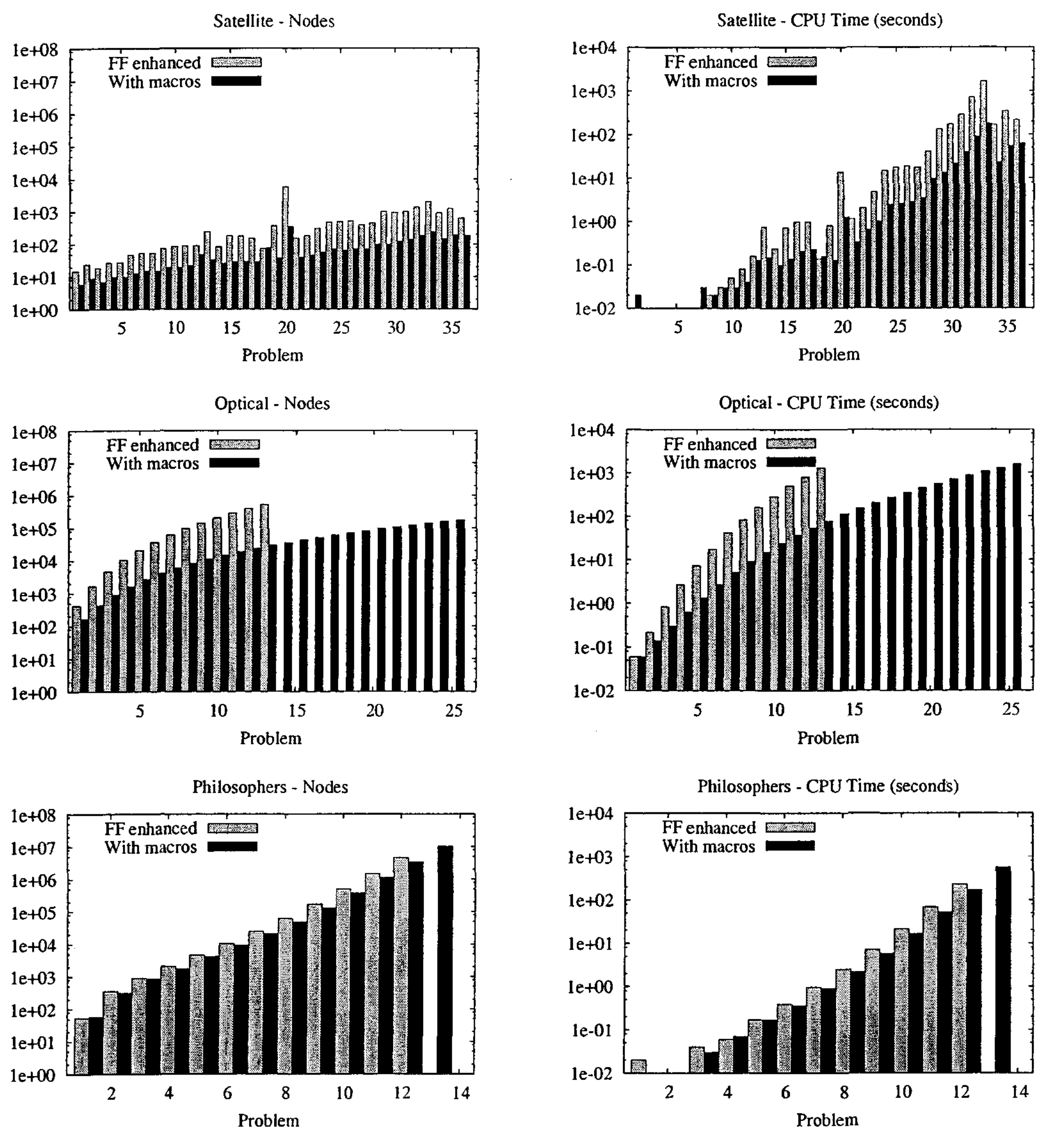


Figure 5.8: Comparison of FF with and without competition macros in Satellite, Promela Optical Telegraph and Promela Dining Philosophers.

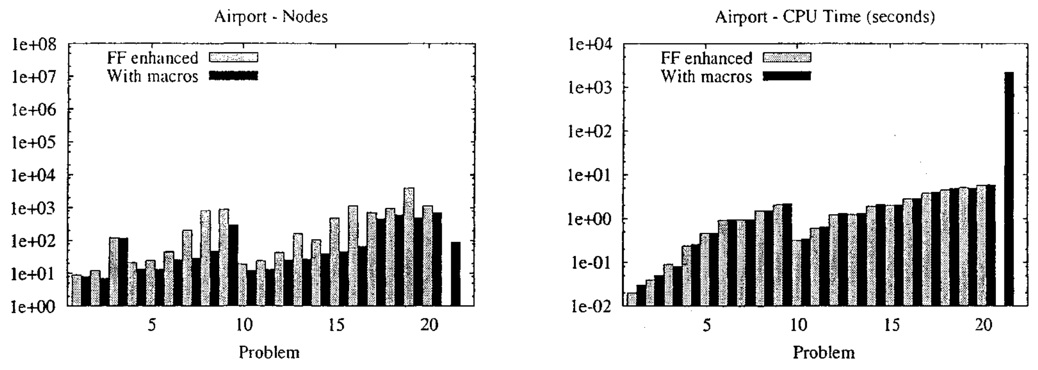


Figure 5.9: Comparison of FF with and without competition macros in Airport.

Airport is used, but no macros could be generated for this domain version. STRIPS Airport uses separate domain definitions for each problem instance, whereas the learning process requires several training problems for one domain definition.

Figure 5.10 contains the results in Pipesworld Non-Temporal No-Tankage, Pipesworld Non-Temporal Tankage, and PSR. In Pipesworld Non-Temporal No-Tankage, macros often lead to significant speed-up. As a result, the system solves four new problems. On the other hand, the system with macros fails in three previously solved problems. The contribution of macros is less significant in Pipesworld Non-Temporal Tankage. The system with macros solves two new problems and fails in one previously solved instance. Out of all seven benchmarks, PSR is the domain where macros have the smallest impact. Both systems solve 29 problems using similar amounts of resources. In the official run on the competition machine, Macro-FF solved 32 problems in this domain.

Table 5.2 shows the number of training problems and the training time in each domain. The training phase used 10 problems for each of Airport, Satellite, Pipesworld Non-Temporal No-Tankage, and PSR. The training sets were reduced to 5 problems for Promela Optical Telegraph, 6 problems for Promela Dining Philosophers, and 5 problems for Pipesworld Non-Temporal Tankage. In Promela Optical Telegraph, the planner with no macros solves 13 problems, and using most of them for training would leave little room for

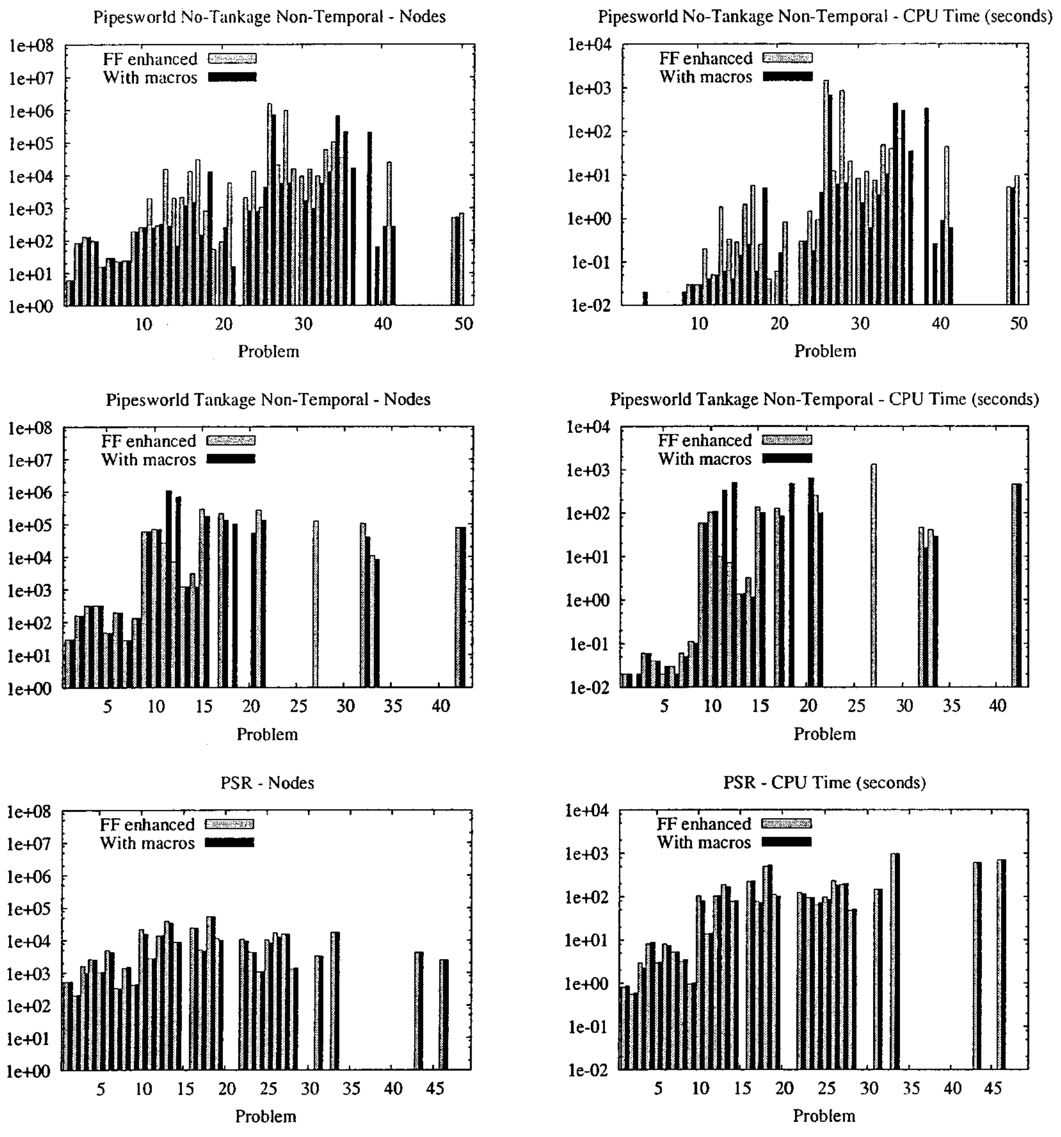


Figure 5.10: Comparison of FF with and without competition macros in Pipesworld No-Tankage Non-Temporal, Pipesworld Tankage Non-Temporal and PSR.

Domain	Number of training problems	Training time (seconds)
Airport	10	365
Promela Optical Telegraph	5	70
Promela Dining Philosophers	6	10
Satellite	10	8
Pipesworld Non-Temporal No-Tankage	10	250
Pipesworld Non-Temporal Tankage	5	4,206
PSR	10	1,592

Table 5.2: Summary of training in each domain.

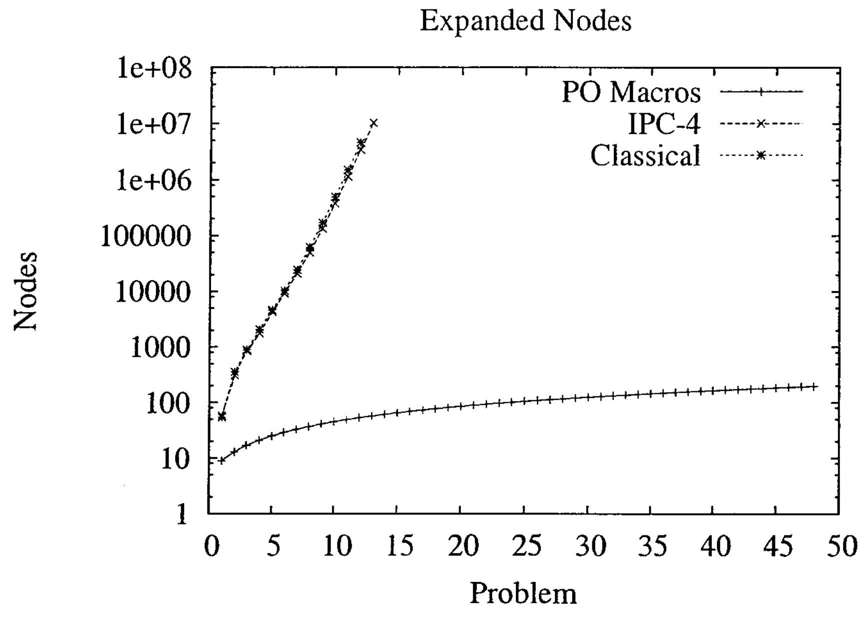
evaluating the learned macros. The situation is similar in Promela Dining Philosophers; the planner with no macros solves 12 problems. In Pipesworld Non-Temporal Tankage, the smaller number of training problems is caused by both the long training time and the structure of the competition problem set. The first 10 problems use only a part of the domain operators, so these were not included into the training set. Out of the remaining problems, the planner with no macros solves 11 instances.

5.3 Evaluating SOL-EP

The main goal of this section is to evaluate the full-scale version of SOL-EP, which extends the version used in the IPC-4 competition. Full-scale SOL-EP is compared with the preliminary SOL-EP and with planning with no macros.

The same seven domains as in the previous section are used. Detailed performance analysis is shown for Promela Dining Philosophers, Promela Optical Telegraph, Satellite, and PSR. Then brief comments are made on Pipesworld Notankage, Pipesworld Tankage and Airport.

Figures 5.11–5.14 show three data curves each. The curves are not cumulative: each data point shows a value corresponding to one problem in the given domain. The horizontal axis preserves the problem ordering as in the compe-



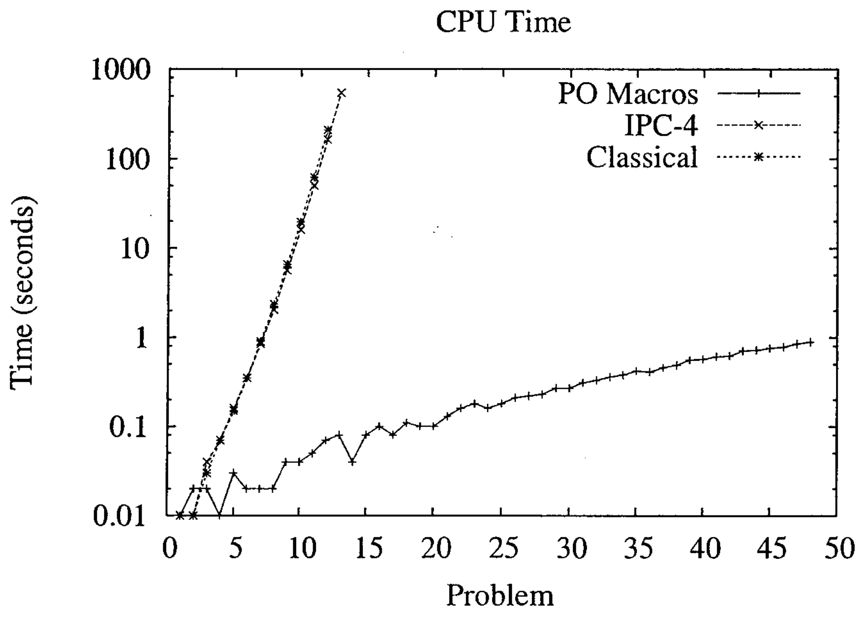
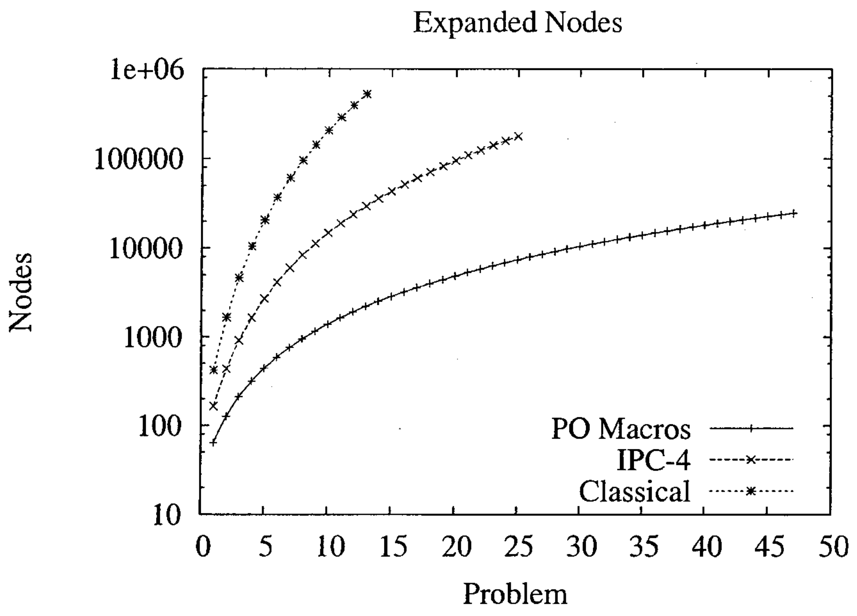


Figure 5.11: Experimental results in Promela Dining Philosophers.



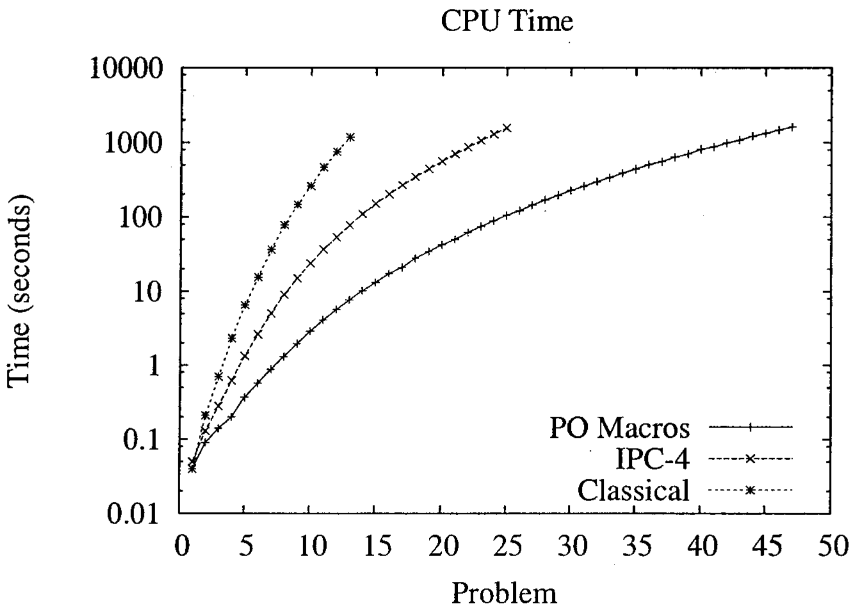
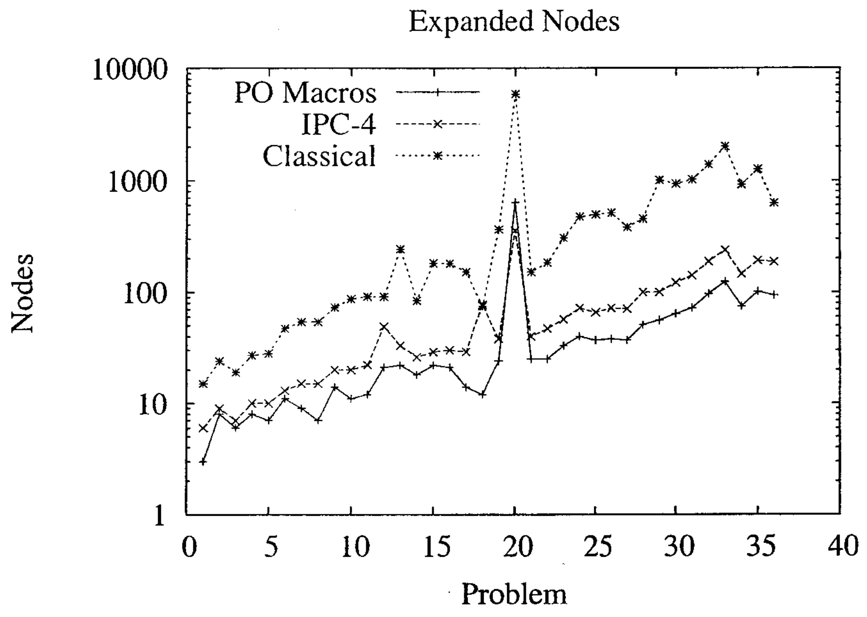


Figure 5.12: Experimental results in Promela Optical Telegraph.



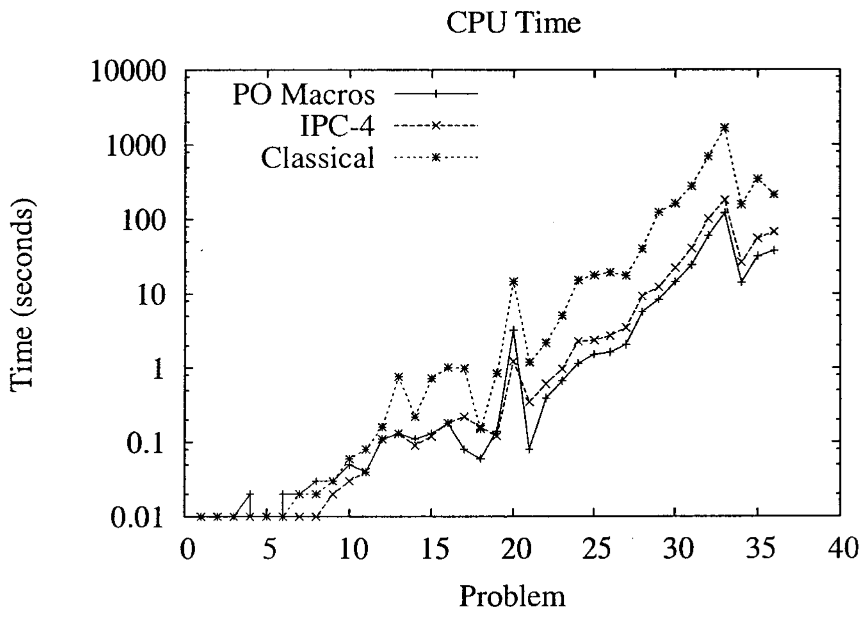


Figure 5.13: Experimental results in Satellite.

tition domains. The data labeled with "Classical" are obtained with FF plus the implementation enhancements, but no macro-operators. "PO Macros" (partial-order macros) corresponds to a planner that implements full-scale SOL-EP on top of "Classical". "IPC-4" shows results with the preliminary SOL-EP.

Figures 5.11 and 5.12 show the results for Promela Dining Philosophers and Optical Telegraph. The new extended model leads to a massive improvement. In Dining Philosophers, each problem is solved in less than 1 second, while expanding less than 200 nodes. In both domains, the new system outperforms by far the top performers in the competition for the same domain versions. If the new version of Macro-FF had been used in the competition, it would have taken the first place in one more domain (Dining Philosophers). No difference was observed in terms of average solution quality between "Classical" and "PO Macros".

Given a problem p, let $CPN_{PO}(p)$ be the cost per node $CPN(p)$ when partial-order macros are used, and $CPN_{Cl}(p)$ be $CPN(p)$ in the classical setting, with no macros. See formula 5.1 for the definition of $CPN(p)$. Let the *cost rate* of problem p be

$$CR(p) = \frac{CPN_{PO}(p)}{CPN_{Cl}(p)}.$$

Statistics were collected about the cost rate from problems solved by *both* planners. In Optical Telegraph, the cost rate varies between 1.40 and 1.47, with an average of 1.43. Since problems in Dining Philosophers are solved very easily (e.g., 33 nodes in 0.01 seconds) in the "PO Macros" setup, it is hard to obtain accurate statistics about the cost rate. The reason is that the reported CPU time always has a small amount of noise partly caused by truncation to two decimal places. When the total time is small too, the noise significantly affects the statistic's accuracy.

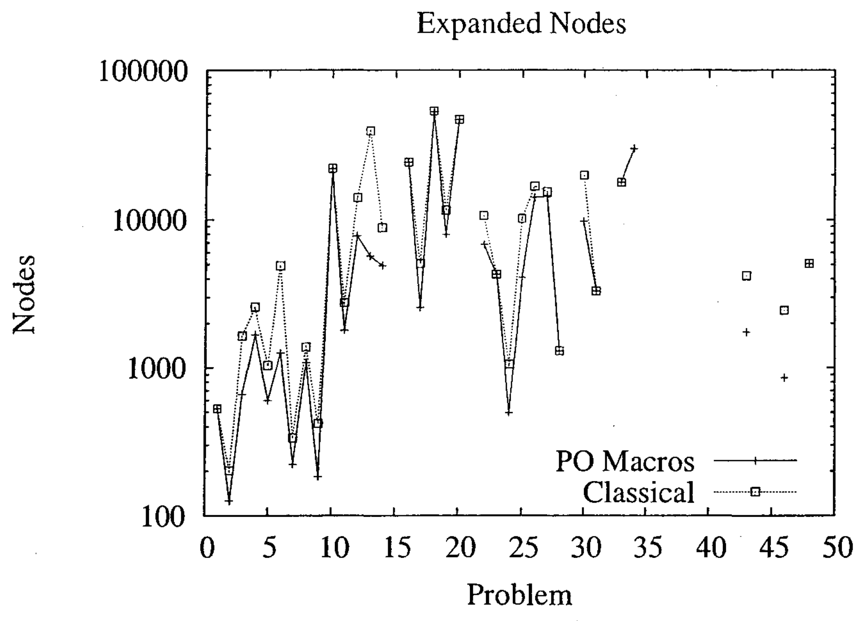
Figure 5.13 summarizes the experiments in Satellite. In the competition results for this domain, Macro-FF and YAHSP tied for first place. The new model further improves Macro-FF's result, gaining up to about one order of magnitude speedup compared to classical search. In Satellite, the heuristic evaluation of a state becomes more expensive as problems grow in size, with interesting effects for the system performance. The rate of the extra cost per node that macros induce is greater for small problems, and gradually decreases for larger problems since, in large problems, the cost of heuristic evaluation dominates. The cost rate varies from 0.83 to 2.04 and averages 1.14. The solution quality slightly varies in both directions, with no significant impact for the system performance.

Figure 5.14 shows experiments in PSR Middle Compiled. Partial-order macros solve 33 problems, as compared to 32 problems in "Classical" (i.e., planning with no macros). In Section 5.2, 29 problems were reported solved by both the classical system and the competition system. The difference, which comes from a small modification in the memory management module of the planner, has little relevance for these experiments.

For this problem set, partial-order macros often achieve significant savings, but never result in more expanded nodes. This is mainly due to the goal macro pruning rule, which turned out to be very selective in PSR. There are problems where the number of expanded nodes is exactly the same in both setups, suggesting that no macro was instantiated at run-time. The cost rate averages 1.39, varying between 1.01 and 1.87.

Compared to the previous three testbeds, the performance improvement in PSR is rather limited. A probable explanation is that the definition of macro equivalence is too relaxed and misses useful structural information in PSR. When checking if two action sequences are equivalent, the current algorithm considers the set of operators, their partial ordering, and the variable binding. The algorithm ignores whether conditional effects are activated correspondingly in the two compared sequence instantiations. However, in PSR, conditional effects encode a significant part of the local structure of a solution. There are operators with zero parameters but rich lists of conditional effects (e.g., operator AXIOM). Further exploration of this insight is left as future work.

In Pipesworld, the generated macros have a very small efficiency rate ER, and the dynamic filtering drops all of them, reducing the search to the classical



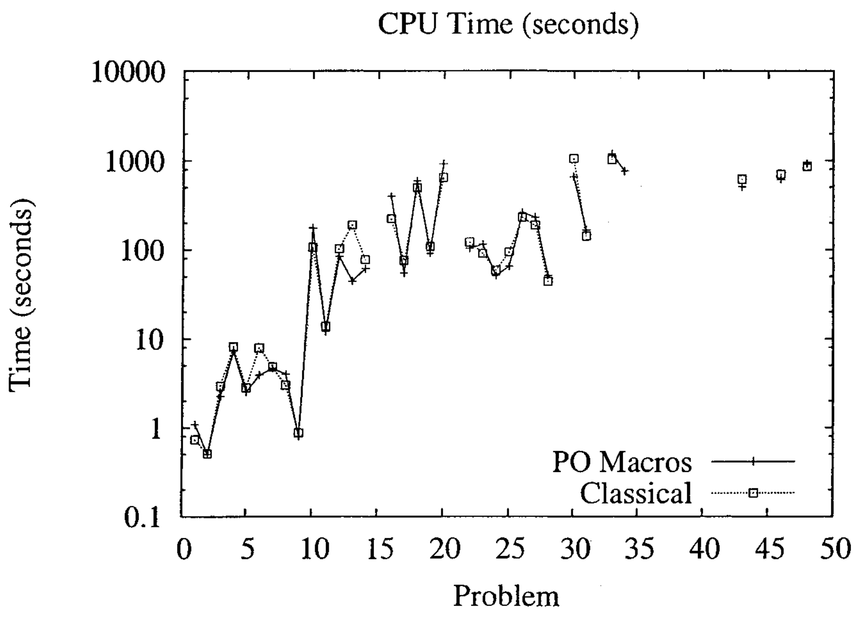


Figure 5.14: Experimental results in PSR Middle Compiled.

algorithm. No experiments were run in Airport. In the ADL version of this domain, the classical algorithm quickly solves the first 20 problems, leaving little room for further improvement. The preprocessing phase of the remaining problems is so hard that only one more instance can be solved within 30 minutes.

An important problem is to evaluate in which domains SOL-EP works well, and in which classes of problems this approach is less effective. Several factors affect the method's performance. The first factor is the efficiency of the macro pruning rules, which control the set of macro instantiations at run-time and influence the planner performance. Efficient pruning keeps only a few instantiations that are shortcuts to a goal state (one such instantiation in a state will do). The performance drops when more instantiations are selected, and many of them lead to subtrees that contain no goal states. The efficiency of helpful macro pruning directly depends on the quality of both the relaxed plan associated with a state, and the macro-schema that is being instantiated. Since the relaxed plan is more informative in Promela and Satellite than in PSR or Pipesworld, the performance of SOL-EP is significantly better in the former applications.

As a second factor, experience suggests that SOL-EP performs better in "structured" domains rather than in "flat" benchmarks. Intuitively, a domain is more structured when more local details of the domain in the real world are preserved in the PDDL formulation. In such domains, local move sequences occur over and over again, and SOL-EP can catch these as potential macros. In contrast, in a "flat" domain, such a local sequence is often replaced with one single action by the designer of the PDDL formulation.

5.4 Conclusions

Chapters 3, 4 and 5 have presented contributions to domain-independent planning. CA-ED and SOL-EP, two abstraction methods that learn macro-operators based on automatic domain analysis, were described and evaluated in detail.

Future work ideas directly related to CA-ED and SOL-EP were previously expressed. At a higher level, planning research should faster expand from the narrow area of pure research towards solving more classes of real-life problems. Planning has many applications, from an automated personal agenda to complex industrial or research projects. Any problem that exhibits frequent multiple-choice decisions generates a search space and can potentially be modeled with planning. Hence excellent opportunities for integrating planning solutions into multi-disciplinary projects exist.

An example from commercial games illustrates this. The behavior of game characters is often encoded with scripts, which are in fact *rigid* plans. The rigidity means that in some circumstances a script might be used innapropriately. As an example, consider a character that enters a tavern, steps to the counter and says "One drink, please!". This is totally unreasonable if nobody is at the counter to answer the request. Scripts could be replaced by a planning engine that dynamically generates plans according to the given context. There is no need to spend useful resources to generate long and complicated plans. Even quickly-found short plans that make sense to users would make a big difference in the quality of a game.

Chapter 6

Hierarchical Path-Finding with Topological Abstraction

The problem of path-finding in commercial computer games has to be solved in real time, often under constraints of limited memory and CPU resources. The industry standard is to use A* [83] or iterative-deepening A*, IDA* [56]. A* is generally faster, but IDA* uses less memory. There are numerous enhancements to these algorithms to make them run faster or explore a smaller search tree. For many applications, especially those with multiple moving units (such as in real-time strategy games), these time and/or space requirements are limiting factors.

Hierarchical search is acknowledged as an effective approach to reduce the computational effort needed to find path-finding solutions. Recently, variations of hierarchical search appear to be in use in several games. However, no detailed study of hierarchical path-finding in commercial games had been published before [12]. Part of the explanation is that game companies usually do not make their ideas and source code available.

This chapter describes Hierarchical Path-Finding A* (HPA*), a new method for hierarchical search on grid-based maps. HPA* decomposes a map into a collection of local clusters. Each cluster has a small set of entrances. Within each cluster, distances between all pairs of entrances are precomputed and cached. Search is done at an abstract level, where a cluster can be crossed in one single step. Several such abstractions can hierarchically be applied, making this approach scalable for large problem spaces. A cluster at a new abstraction level groups several adjacent clusters at the previous level, so that a higher level abstracts the map into fewer clusters of larger size.

After clustering, path planning starts with an abstract search at the highest level. An abstract path can gradually be refined until a complete low-level path is obtained.

Compared to low-level A*, problem decomposition has two main benefits. Finding a solution at a high level of abstraction is usually solved much faster than the original problem while producing only slightly sub-optimal results. Second, increased execution flexibility is possible, allowing for parts of a problem to be solved only if and when it is necessary. This is useful in several circumstances. First, when memory is available, results of popular local searches can be cached for future reuse. This can be the case when many searches have the same origin and/or destination, or more generally when many paths share a common portion such as a bridge. Second, for many real-time path-finding applications, the complete path between two points is not needed beforehand. Quickly obtaining the first few steps of a valid path often suffices, allowing a mobile unit to start moving in the right direction. Subsequent path refinement can be solved as needed, providing additional moves. If a unit has to change its plan, for example because of collisions with other mobile units, then no effort has been wasted on computing a detailed path to a goal node that was never used.

In contrast, A* must complete its search and generate the entire path from start to destination before it can determine the first steps of a *correct* path. Using a partial solution that A* can provide in a limited amount of time is not guaranteed to work. Since partial solutions do not reach the target location, the direction of a partial solution may be wrong.

The hierarchical framework is suitable for both static and dynamically changing environments. In the latter case, assume that local topology changes can occur (e.g., a bomb destroys a bridge). HPA* will recompute the information extracted from the modified cluster locally and keep the rest of the framework unchanged.

HPA* is simple, easy to implement, and independent of the map properties.

No implementation changes are required to handle variable cost terrains and various topology types such as forests, open areas with obstacles of any shape, or building interiors.

Section 6.1 presents this new approach to hierarchical A*. Performance tests are presented in Section 6.2, showing up to 10 times speed-up and 1% solution degradation as compared to A*. Section 6.3 presents conclusions and topics for further research. Appendix C provides algorithmic details of HPA*, including pseudo-code.

6.1 Hierarchical Path-Finding A*

HPA* starts with a *preprocessing* phase, which uses map abstraction to build a hierarchical search space called an *abstract graph.* Then a path can be computed with the so-called *on-line search* phase. An abstract graph can be re-used for many online searches, amortizing its computational cost. This section discusses in more detail how the framework for hierarchical search is built (preprocessing) and how it is used for path finding (on-line search). The initial focus is on building a two-level hierarchy. Adding more hierarchical levels is discussed at the end of this section. The 40×40 map shown in Figure 6.1 (a) serves as an illustrative example.

A few assumptions are made with respect to the grids and the search algorithms used by HPA*, as shown below. HPA* is by no means limited to these settings, they are used only for a simpler and more clear presentation. A grid uses *octiles*, which are tiles that define the adjacency relationship in 4 straight and 4 diagonal directions. The cost of vertical and horizontal transitions is 1. Diagonal transitions have the cost set to 1.42.[1] Diagonal moves between two blocked tiles are not allowed.

All searches that HPA* performs (e.g., during preprocessing, abstract search, refinement, etc.) are assumed to use A* with the following heuristic. Given two grid locations $l_1(x_1, y_1)$ and $l_2(x_2, y_2)$, consider $M = \max(|x_1 - x_2|, |y_1 - y_2|)$

[1]The path-finding library used to implement HPA* utilizes this value for approximating $\sqrt{2}$. A slightly more appropriate approximation would probably be 1.41.

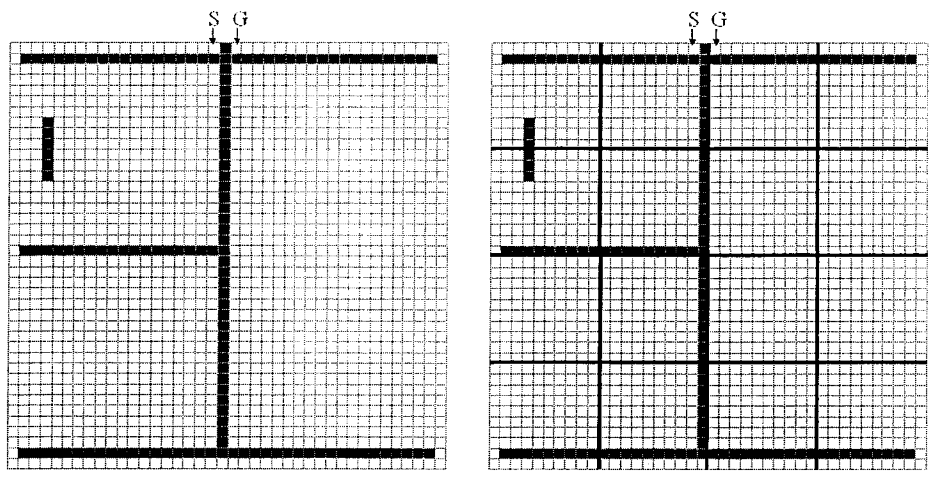


Figure 6.1: (a) The 40×40 grid g used as an example. The obstacles $obs(g)$ are painted in black. S and G are the start and the goal nodes. (b) The bold lines show the boundaries between 10×10 clusters.

and $m = \min(|x_1 - x_2|, |y_1 - y_2|)$. The heuristic distance between l_1 and l_2 is

$$d(l_1, l_2) = 1.42 \times m + (M - m).$$

This is in fact the length of a shortest path between l_1 and l_2 on an octile grid with no obstacles.

6.1.1 Preprocessing a Grid

Preprocessing is performed in two steps: (1) apply topological abstraction to the map and (2) build the abstract graph. Topological abstraction partitions the space into a set of disjunct rectangular areas called *clusters*. In this example, the 40×40 grid is partitioned into 16 clusters of size 10×10, as shown in Figure 6.1 (b). No domain knowledge is used to do this abstraction, other than the presence of a map and, perhaps, tuning the size of the clusters.

For each border line between two adjacent clusters, a (possibly empty) set of entrances connecting the clusters is identified. An entrance is a maximal obstacle-free segment along the common border of two adjacent clusters c_1 and c_2, formally defined below. Consider the two adjacent rows (or columns) of tiles l_1 and l_2, one in each cluster, that are separated by the border edge

b between c_1 and c_2. For a tile $t \in l_1 \cup l_2$, let $sym(t)$ be the symmetrical tile of t with respect to b. Tiles t and $sym(t)$ are adjacent and never belong to the same cluster. An entrance e is a set of tiles that respects the following conditions:

1. e is connected.
2. The border limitation condition: $e \subset l_1 \cup l_2$. This condition states that an entrance is defined along and cannot exceed the border between two adjacent clusters.
3. The symmetry condition: $\forall t \in l_1 \cup l_2 : t \in e \Leftrightarrow sym(t) \in e$.
4. An entrance contains no obstacle tiles: $e \cap obs(g) = \emptyset$.
5. Maximality: no superset of e satisfies conditions 1 – 4.

Figure 6.2 shows a zoomed-in picture of the upper-left quarter of the sample map. The picture shows details on how entrances are identified and used to build the abstract problem graph. In this example, the two clusters on the left side are connected by two entrances of width 3 and of width 6 respectively. For each entrance e, one or two *transitions* are defined, depending on the entrance width. A transition is a pair of symmetrical tiles $(t, sym(t)) \in e \times e$ that allows communication between clusters. Let T_e be the set of all tiles that belong to transitions of e. Only tiles $t \in T_e$ are used for communication between clusters. If the width of the entrance is less than a predefined constant (6 in the example), then one transition is defined at the middle of the entrance. Otherwise, two transitions are created, one on each end of the entrance. Defining such a small number of transitions preserves the completeness and the correctness of the algorithm. However, solutions can be suboptimal, since there can be an entrance e and a node pair (S, G) such that all optimal paths between S and G must intersect $e \setminus T_e$.

Transitions are used to build the abstract problem graph. Each transition generates two nodes in the abstract graph, and an edge that links them. Since

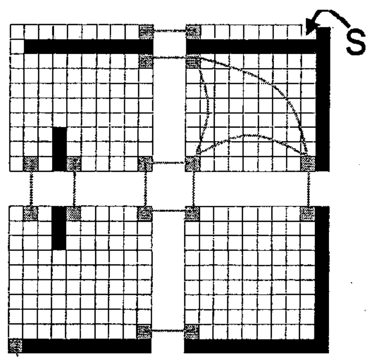
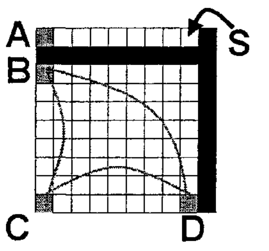


Figure 6.2: Abstracting the top-left corner of g. All abstract nodes and inter-edges are shown in light grey. For simplicity, intra-edges are shown only for the top-right cluster.

such an edge represents a transition between two clusters, it is called an *inter-edge.* Inter-edges always have length 1. For each pair of nodes inside a cluster, an edge linking them, called an *intra-edge*, is defined. The length of an intra-edge is obtained by searching for an *optimal* path inside the cluster area. In this work, optimality is defined with respect to path length. The length of a path is the sum of the weights of its steps (edges). In particular, when all edges on a path have weight 1, its length is the number of steps.

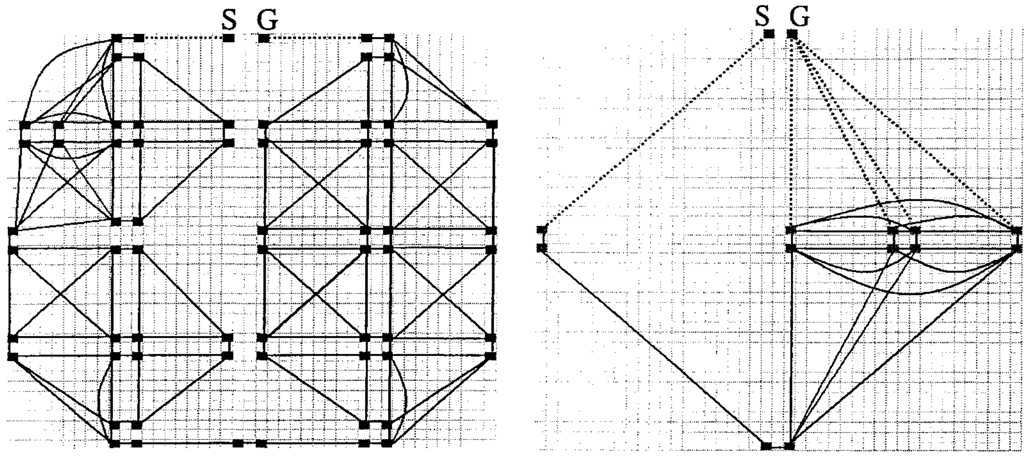
Figure 6.2 shows all nodes (light grey squares), all inter-edges (light grey lines), and part of the intra-edges (for the top-right cluster). Figure 6.3 shows the details of the abstracted internal topology of the cluster in the top-right corner of Figure 6.2. The data structure contains a set of nodes as well as distances between them. For example, going from B to D has a minimal cost of 10.94, the result of 7 diagonal moves and one move to the right. This method currently caches distances between nodes and discards the actual optimal paths corresponding to these distances. If desired, the paths can also be stored, for the price of more memory usage.

Figure 6.4 (a) shows the abstract graph for the running example. The picture includes the result of inserting the start and goal nodes S and G into the graph (the dotted lines), which is described in the next subsection. The graph has 68 nodes, including S and G, which can change for each search. At this level of abstraction, there are 16 clusters with 43 interconnections and 88



	A	B	C	D
A	■	∞	∞	∞
B	■	■	7.00	10.94
C	■	■	■	8.00
D	■	■	■	■

Figure 6.3: Cluster-internal path information.



(a) (b)

Figure 6.4: (a) The abstract problem graph in a hierarchy with one low level and one abstract level. (b) Level 2 of the abstract graph in the 3-Level hierarchy.

intraconnections. There are 2 additional edges that link S and G to the rest of the graph. For comparison, the low-level (non-abstracted) graph contains 1,463 nodes, one for each unblocked tile, and 2,714 edges.

Once the abstract graph has been constructed and the intra-edge distances computed, the grid is ready to use in a hierarchical search. This information can be precomputed (before a game ships), stored on disk, and loaded into memory at run-time. This is sufficient for static (non-changing) grids. For dynamically changing grids, the precomputed data has to be modified at run-time. When the grid topology changes (e.g., a bridge blows up), only the intra- and inter-edges of the affected local clusters need to be re-computed.

6.1.2 On-line Search

HPA* first searches for a path in the abstract graph. The abstract path can subsequently be refined, as well as improved in quality (e.g., aesthetics and length) as needed.

Searching for an Abstract Path

Searching for an abstract solution in the hierarchical framework is a three-step process based on the following strategy: First, discover how to travel from the start to each node on the border of its neighborhood. Second, discover how to travel from each node on the border of the goal neighborhood to the goal position. Third, search for a path from the border of the start neighborhood to the border of the goal neighborhood. This is done at an abstract level, where search is simpler and faster. A single action traverses a relatively large area.

At step 1, S is temporarily inserted into the abstract graph by adding edges to all reachable nodes on the border of the cluster containing S. Local searches are run for each pair (S, n), where n is a graph node on the border of S's cluster. Such a local search is restricted to the area of the cluster. An intra-edge between S and n is added if a local path exists between them. Each edge is weighted by the length of an optimal path between the two nodes. In Figure 6.4 these edges are represented with dotted lines. The paths can also be cached and reused in the refinement phase, allowing a mobile unit to start moving as soon as the on-line search finishes. Step 2, which connects G to its cluster border, is similar to step 1.

In experiments, S and G are assumed to change for each new search. Therefore, the cost of inserting S and G is added to the total cost of finding a solution. After a path is found, S and G are removed from the graph. However, in practice this computation can be done more efficiently. Consider a game where many units have to find a path to the same goal. In this case, G can be inserted once and reused in several searches, amortizing the insertion cost. In general, a cache can be used to store connection information for popular start and goal nodes.

At step 3, a search in the abstract graph computes a path between S and G. The last two steps of the on-line search are optional:

- Path-refinement can be used to convert an abstract path into a sequence of moves on the original grid. Each abstract edge is mapped to a shortest low-level path between its two end nodes. Note that global optimality is not ensured, because of the small number of nodes defined for each entrance.

- Path-smoothing can be used to improve the quality of the path-refinement solution.

Path Refinement

Path refinement translates an abstract path back into a low-level path. Each intra-edge in the abstract path is replaced by an equivalent sequence of low-level moves. If the move sequence attached to an abstract step has been cached, then its refinement is simply a table look-up. Otherwise, a small search is performed inside the corresponding cluster to rediscover the low-level move sequence.

There are two factors that keep the refinement search simple. First, abstract solutions are guaranteed to be correct, provided that the environment does not change after finding an abstract path. This means that neither backtrack nor re-planning for correcting an abstract solution are necessary. Second, path-refinement is a sum of small searches, one for each intra-edge on an abstract path. The total effort to solve all subproblems is often smaller than the effort to solve the original problem.

Path Smoothing

Topological abstraction defines only one or two transition points per entrance. While efficient, this gives up the optimality of the computed solutions. Solutions are optimal in the abstract graph but not necessarily in the initial problem graph. Path smoothing improves the solution quality (i.e., cost and

Search Technique	*SG*	Main	**Total Abstract**	Refinement (optional)
L-0	0	1,462	**1,462**	0
L-1	17	67	**84**	145
L-2	44	7	**51**	161

Table 6.1: Number of expanded nodes in the running example.

aesthetics). The technique for path smoothing is simple, but produces good results. Assume $n_1 n_2 ... n_l$ are the nodes of a path. Path-smoothing detects pairs $(n_i, n_j), i < j$ such that n_i and n_j can be connected by a straight line, and $n_i n_{i+1} ... n_j$ is sub-optimal. Then $n_i n_{i+1} ... n_j$ is replaced by a straight line.

When all pairs $(n_i, n_j), i < j$ are considered, an upper bound on the complexity of path-smoothing is $O(l^2)$, where l is the number of low-level path nodes. In practice, better performance is achieved based on a few simple ideas. First, no check is necessary between two nodes inside the same cluster, since all interior local paths are optimal. Second, an effective heuristic is that, after a local sequence $n_i ... n_j$ has been corrected, the process continues from n_j rather than n_{i+1}. Third, if a path is optimal beforehand, or becomes optimal after one or several smoothing steps, no further smoothing is necessary. However, a challenge is how to quickly determine whether a given path is optimal. A simple but partial solution is to check whether the cost of a path is the same as the heuristic distance between S and G. If so, the path is proven to be optimal, as the heuristic is admissible. Otherwise, no conclusion can be drawn on this matter.

6.1.3 Experimental Results for the Running Example

The experimental results for the running example are summarized in the first two rows of Table 6.1. *SG* is the effort for inserting S and G into the graph. Main represents searching for an abstract path. *Total Abstract* is the sum of the previous two columns. This measures the effort for finding an abstract solution. *Refinement* shows the effort for complete path-refinement. L-0 represents running A* on the low-level graph (called level 0). L-1 uses two hierarchy levels (level 0 and level 1), and L-2 uses three hierarchy levels. For now the focus is only on L-0 and L-1. L-2 will be described in Section 6.1.5.

Low-level (original grid) search using A* has poor performance. The example has been chosen to show a worst-case scenario. Without abstraction, A* will visit all the unblocked positions in the map. The search expands 1,462 nodes. The only factor that limits the search is the map size. A larger map with a similar topology represents a hard problem for A*.

The performance is greatly improved by using hierarchical search. When inserting S into the abstract graph, it can be linked to only one node on the border of the starting cluster. Therefore one node (corresponding to S) and one edge that links S to the only accessible node in the cluster are added. Finding the edge cost uses a search that expands 8 nodes. Inserting G into the graph is almost identical (9 nodes expanded).

A* is used on the abstract graph to search for a path between S and G. Searching at level 1 expands all the nodes of the abstract graph. The problem is also a worst-case scenario for searching at level 1. However, the search effort is much smaller: The main search expands 67 nodes. Inserting S and G expands 17 nodes. In total, finding an abstract path requires 84 node expansions. If desired, this abstract path can be refined, partially or completely, for additional cost. The cost is higher when the path has to be refined completely and no actual paths for intra-edges were cached. For each intra-edge in the path, a search computes a corresponding low-level action sequence. In the example, there are 12 such small searches, which expand a total of 145 nodes.

6.1.4 Adding Levels of Hierarchy

The hierarchy can be extended to several levels, transforming the abstract graph into a *multi-level* graph. In a multi-level graph, nodes and edges have labels showing their level in the abstraction hierarchy. HPA* performs path-finding as a combination of small searches in the graph at various abstraction levels. Additional levels in the hierarchy can reduce the search effort, especially for large maps. See Appendix C.2.2 for details on efficient searching in a multi-level graph. To build a multi-level graph, map abstraction is structured on several levels. The higher the level, the larger the clusters in the map decomposition. Clusters at level l are called l-clusters. Each new level is built on top of the existing structure. Building the 1-clusters has been presented in Section 6.1.1. For $l \geq 2$, an l-cluster is obtained by grouping together $n \times n$ adjacent $(l-1)$-clusters, where n is a parameter.

If two nodes and an inter-edge at level $l-1$ make a transition between two newly created l-clusters, all three elements update their level to l. (Nodes at level l are called l-nodes, and edges at level l are called l-edges.) Note that l-nodes and l-inter-edges, $l \geq 2$, are inherited from the previous level. Not introducing new nodes with a new graph level is beneficial for both building new intra-edges at level l, and refining an abstract solution, as detailed below.

Intra-edges with level l (i.e., l-intra-edges) are added for pairs of communicating l-nodes placed on the border of the same l-cluster. Since both ends of a new intra-edge were present at level $l-1$ too, such an edge is quickly computed with a search at level $l-1$ inside the current l-cluster. More details are provided in Appendix C.2.2.

Inserting S into the graph iteratively connects S to the nodes on the border of the l-cluster that contains it, with l increasing from 1 to the maximal abstraction level. Searching for a path between S and a l-node is restricted to level $l-1$ and to the area of the current l-cluster that contains S. An identical processing is performed for G too.

The number of abstract levels can affect the computation speed, but not the solution itself. In particular, adding a new level $l \geq 2$ to the graph does not diminish the solution quality. The intuition behind this is the following: All nodes and inter-edges at level l are obtained from nodes at level $l-1$. A new intra-edge added at level l corresponds to an existing shortest path at level $l-1$. The weight of the new edge is set to the cost of the corresponding path. Searching at level l finds faster the same solution as searching at level $l-1$ (only more abstracted), since the added edges do not change shortest distances.

In the example, adding an extra level with $n = 2$ creates 4 large clusters, one for each quarter of the map. Figure 6.2 is an example of a single 2-cluster. This cluster contains 2×2 1-clusters of size 10×10. Besides S, the only other level-2 node of this cluster is the one in the bottom-left corner. Compared to level 1, the total number of nodes at the second abstraction level is reduced even more. Level 2, where the main search is performed, has 14 nodes (including S and G). Figure 6.4 (b) shows level 2 of the abstract graph. The edges pictured as dotted lines connect S and G to the graph at level 2.

Abstraction level 2 is a good illustration of how the preprocessing solves local constraints and reduces the search complexity in the abstract graph. The 2-cluster shown in Figure 6.2 is large enough to contain the large dead end "room" that exists in the local topology. At level 2, the algorithm avoids any useless search in this "room" and goes directly from S to the cluster exit in the bottom-left corner.

After inserting S and G, the graph can be searched for a path between these two nodes. Search is performed at the highest abstraction level. If desired, the abstract path can repeatedly be refined to the previous level until the low-level solution is obtained. A solution refined from level l to level $k, 1 \leq k < l$ is identical to the solution computed in a hierarchy with only k levels. As shown before, refinement from level 1 to the original grid is not guaranteed to produce an optimal solution.

6.1.5 Experimental Results for Example with 3-Level Hierarchy

The third row of Table 6.1 shows numerical data for the running example with a 3-Level hierarchy. As shown in Section 6.1.3, connecting S and G to the border of their 1-clusters expands 17 nodes in total. Similarly, S and G are connected to the border of their 2-clusters. These searches at level 1 expand 5 nodes for S and 22 nodes for G.

The main search at level 2 expands only 7 nodes. No nodes other than the ones in the abstract path are expanded. This is an important improvement, considering that search in the level 1 graph expanded all nodes in the graph. In total, finding an abstract solution in the extended hierarchy requires 51 nodes.

After adding a new abstraction level, the cost for inserting S and G dominates the main search cost. This illustrates the general characteristic of the method that the cost for inserting S and G increases with the number of levels, whereas the main search becomes simpler. Finding a good trade-off between these searches is important for optimizing performance.

Table 6.1 also shows the costs for a complete solution refinement. Refining the solution from level 2 to level 1 expands 16 nodes and refining from level 1 to level 0 expands 145 nodes, for a total of 161 nodes.

6.1.6 Storage Analysis

Besides the computational speed, the amount of storage is another important performance indicator for path-finding. This section analyzes the size of the problem graph and the size of the open list used by A*.

Graph Storage Requirements

Table 6.2 shows the average size of a problem graph for a set of maps extracted from the BALDUR'S GATE game. See Section 6.2.1 for details on this dataset. The original low-level graph is compared to the abstract graphs in hierarchies with one, two, and three abstract levels (not counting level 0). The table shows the number of nodes N, the number of inter-edges E_1, and the number of intra-edges E_2. For the multi-level graphs, both the total numbers and the numbers for each level L_i, $i \in \{1, 2, 3\}$ are presented.

The data shows that the abstract graph is small compared to the size of the original problem graph. Adding a new graph level does not create new nodes and inter-edges. The only overhead consists of the new intra-edges. In the data set, at most $1,846$ intra-edges (when three abstract levels are defined) are added to an initial graph having $4,469$ nodes and $16,420$ edges. Assuming that a node and an edge occupy about the same amount of memory, the overhead is less than 10%.

The way that the abstract graph translates into bytes is highly dependant on factors such as implementation, compiler optimizations, or size of the prob-

	Graph 0	Graph 1		Graph 2			Graph 3			
		L_1	Total	L_1	L_2	Total	L_1	L_2	L_3	Total
N	**4,469**	367	**367**	186	181	**367**	186	92	89	**367**
E_1	**16,420**	198	**198**	100	98	**198**	100	50	48	**198**
E_2	**0**	722	**722**	722	662	**1,384**	722	622	462	**1,846**

Table 6.2: The average size of the problem graph in the BALDUR'S GATE test set. N is the number of nodes, E_1 is the number of inter-edges, and E_2 is the number of intra-edges. Graph 0 is the initial low-level graph. Graph 1 represents a graph with one abstract level (L_1), Graph 2 has two abstract levels (L_1, L_2) , and Graph 3 has three abstract levels (L_1, L_2, L_3).

lem map. For instance, if the map size is at most 256×256, then storing the coordinates of a node takes two bytes. More memory is necessary for larger maps.

Since abstract nodes and edges are labeled by their level, the memory necessary to store an element might be larger in the abstract graph than in the initial graph. This additional requirement can be as little as 2 bits per element, corresponding to a largest possible number of levels of 4. Since most compilers round up the bit-size of objects to a multiple of 8, this overhead might not exist in practice.

The storage utilization can be optimized by keeping in memory (e.g., the cache) only those parts of the graph that are necessary for the current search. In the hierarchical framework, only the sub-graph corresponding to the level and the area of the current search is required. For example, when the main abstract search is performed, the low-level problem graph can be dropped, greatly reducing the memory requirements for this search.

The worst case scenario for a cluster is when blocked tiles and free tiles alternate on the border, and any two border nodes can be connected to each other. Assume the size of the problem map is $m \times m$, the map is decomposed into $c \times c$ clusters, and the size of a cluster is $n \times n$. In the worst case, a number of $4n/2 = 2n$ nodes per cluster is obtained. Since each pair of nodes defines an intra-edge, the number of intra-edges for a cluster is $2n(2n-1)/2 = n(2n-1)$. This analysis holds for clusters in the middle of the map. No abstract nodes

	Low level	Abstract		
		Main	*SG*	Refinement
Open list size	51.24	17.23	4.50	5.48

Table 6.3: Average size of the open list in A*. For hierarchical search, the average size for the main search, the *SG* search (i.e., search for inserting S and G into the abstract graph), and the refinement search are shown.

are defined on the map edges, so marginal clusters have a smaller number of abstract nodes. For the cluster in a map corner, the number of nodes is n and the number of intra-edges is $n(n-1)/2$. For a cluster on a map edge, the number of nodes is $1.5n$ and the number of intra-edges is $1.5n(1.5n-1)/2$. There are 4 corner clusters, $4c-8$ edge clusters, and $(c-2)^2$ middle clusters. Therefore, the total number of abstract nodes is $2m(c-1)$. The total number of intra-edges is $n(c-2)^2(2n-1)+2n(n-1)+3n(c-2)(1.5n-1) \approx 2n^2c^2 = 2m^2$, having the same order as the number of original nodes and edges. The number of inter-edges is $m(c-1)$.

Storage for the A* Open List

Since hierarchical path-finding decomposes a problem into a sum of small searches, the open list in A* usually is smaller in hierarchical search than in low-level search. Table 6.3 illustrates this for searches run on the BALDUR'S GATE testset described in Section 6.2.1. The data shows a three-fold reduction of the list size between the low-level search and the main search in the abstracted framework.

6.2 Experimental Results

6.2.1 Experimental Setup

Experiments were performed on a set of 120 maps extracted from BioWare's game BALDUR'S GATE, varying in size from 50×50 to 320×320. For each map, 100 searches were run using randomly generated S and G pairs for which a valid path between the two locations existed. The atomic map decomposition uses octiles. Entrances with width less than 6 have one transition. For larger entrances two transitions are generated.

The code was implemented using the University of Alberta Path-finding Code Library available at [25]. This library is used as a research tool for quickly implementing different search algorithms using different grid representations. Because of its generic nature, there is some overhead associated with using the library. All times reported in this section should be viewed as generous upper bounds on a custom implementation.

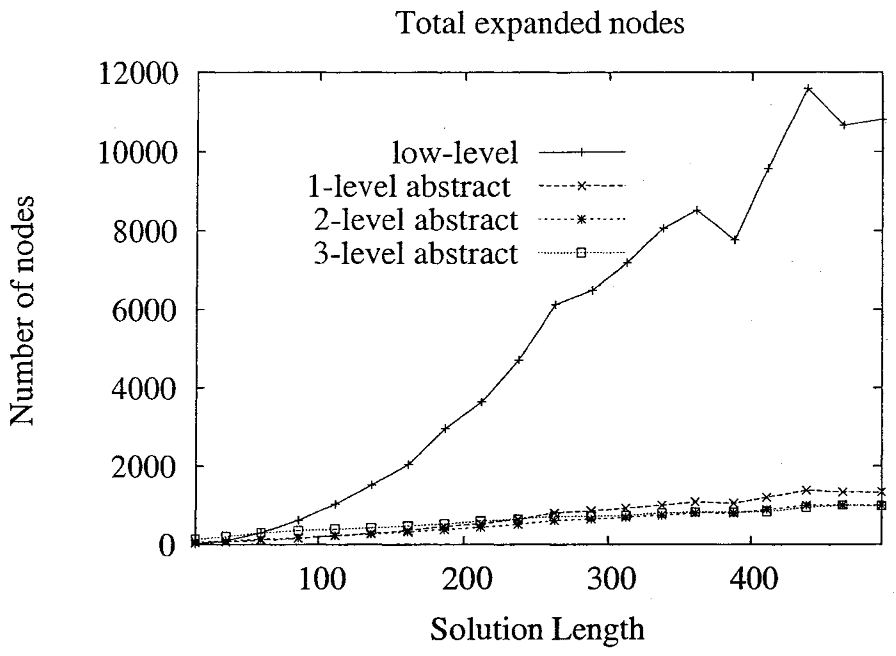
6.2.2 Analysis

Figure 6.5 compares low-level A* to abstract search on hierarchies with the maximal level set to 1, 2, and 3. The top graph shows the number of expanded nodes and the bottom graph shows the time. For hierarchical search, the figures display the total effort, which includes inserting S and G into the graph (the *SG effort*), searching at the highest level (the *main effort*), and refining the path (the *refinement effort*). The real effort can be smaller since the SG effort can be amortized for many searches, and path refinement is not always necessary. The graphs show that, when complete processing is performed, the first abstraction level is good enough for the map sizes used in this experiment. For larger maps, the benefits of more levels could be more significant.

Even though the reported times are for a generic implementation, it is important to note that for any solution length the appropriate level of abstraction was able to provide answers in less than 10 milliseconds on average. Through length 400, the average time per search was less than 5 milliseconds on a 800 MHz machine.

A* is slightly better than HPA* for easy search problems, when the solution length is very small. The overhead of HPA* (e.g., the SG cost) in such cases is larger than the potential savings that the algorithm could achieve. A* is also better when S and G can be connected through a "straight" line on the grid. In this case, the heuristic provides perfect information, and A* expands no nodes other than those that belong to the solution.

The CPU times reported for A* are under 0.1 seconds, and hence one could



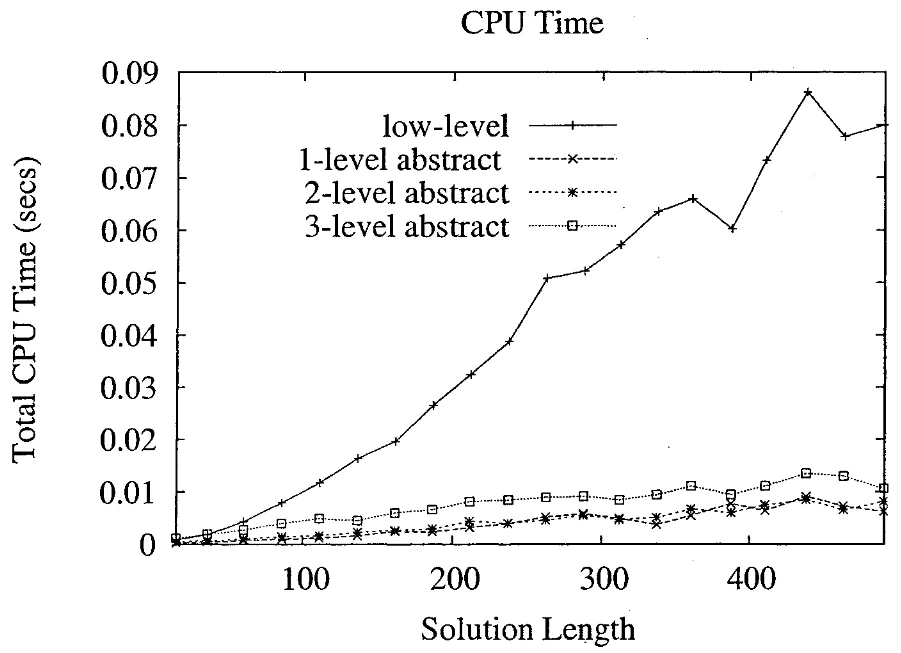


Figure 6.5: Low-level A* vs. hierarchical path-finding.

wonder why bother to improve a result that already looks good. There are several reasons that motivate this. First off, the performance of A* is expected to decrease as the size of a problem map increases. Second, in a game with mobile units, *many* path-finding problems (say, one for each unit) have to be solved on a map within a limited time interval. The difference in performance between A* and HPA* multiplies with the number of problems being solved. Third, in a game, many CPU cycles are taken by other game modules (e.g., the graphics engine), and waiting until the AI module gets a share of 0.1 seconds of CPU for each mobile unit could be impractical.

Figure 6.6 shows how the total effort for hierarchical search is composed of the main effort, the SG effort, and the refinement effort. The charts show that more levels are useful when path refinement is not necessary and S or G can be used for several searches.

Figure 6.7 shows the solution quality. Solutions obtained with hierarchical path-finding are compared to optimal solutions computed by low-level A*. The difference from the minimal cost solution before and after path-smoothing is plotted. The difference is independent of the number of hierarchical levels. The only factor that generates sub-optimality is not considering all the possible transitions for an entrance.

The cluster size is a parameter that can be tuned. The experiments were run using 1-clusters with size 10×10. This choice is supported by the data presented in Figure 6.8. This graph shows how the average number of expanded nodes for an abstract search changes with varying cluster size. While the main search reduces with increasing cluster size, the cost for inserting S and G increases faster. The expanded node count reaches a minimum around cluster size 10.

For higher levels, an l-cluster contains 2×2 $(l-1)$-clusters. When larger values are used, the cost for inserting S and G increases faster than the reduction of the main search. This tendency is especially true on relatively small maps, where smaller clusters achieve good performance and the increased costs for using larger clusters might not be justified. The overhead of inserting S and G results from having to connect S and G to many nodes placed on the

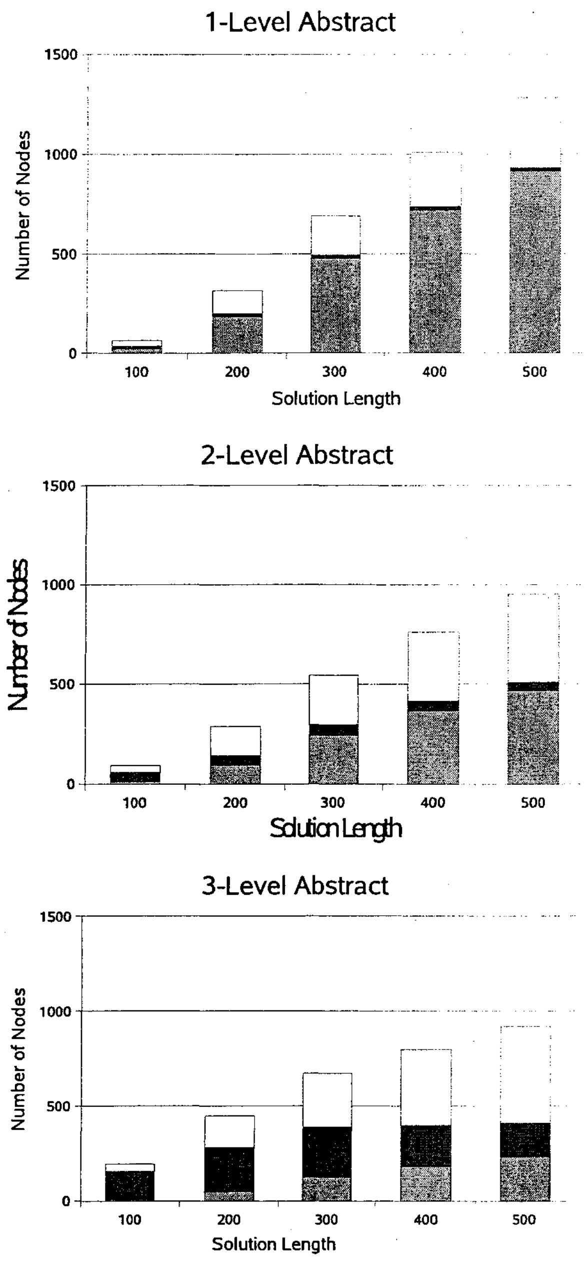


Figure 6.6: The effort for hierarchical search in hierarchies with one, two, and three abstract levels. The total effort is split into the main effort (gray), the *SG* effort (black), and the refinement effort (white).

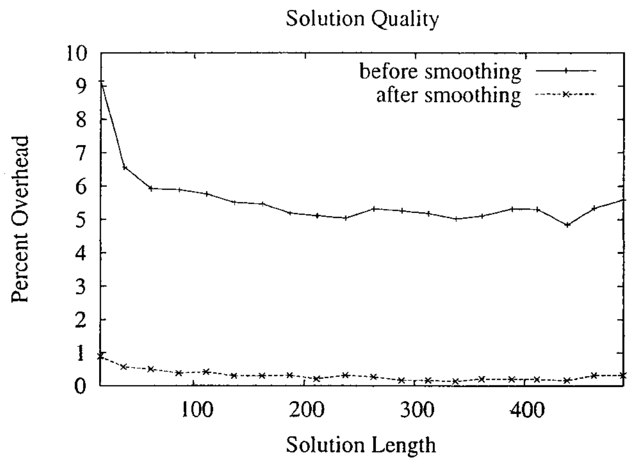


Figure 6.7: Solution quality.

border of a large cluster. The longer the cluster border, the more nodes to connect to.

6.3 Conclusions and Future Work

This chapter presented a hierarchical technique for efficient near-optimal pathfinding. This approach is easy to apply and works well for different kinds of map topologies. The method adapts to dynamically changing environments. The hierarchy can be extended to several abstraction levels, making it scalable for large problem spaces. As seen in planning, adding some simple abstraction allows for significant performance improvement. On maps extracted from a real game, HPA* produces near-optimal solutions much faster than low-level A*.

This work can be extended in several directions. Inserting S and G into the abstract graph can be optimized. As Figure 6.6 shows, these costs increase significantly with adding a new abstraction layer. One strategy for improving the performance is to connect S only to a sparse subset of the nodes on the border, maintaining the completeness of the abstract graph. For instance, if each "unconnected" node (i.e., a node on the border to which no connection from S is attempted) is reachable in the abstract graph from a "connected" node (i.e., a node on the border already connected to S), then completeness is

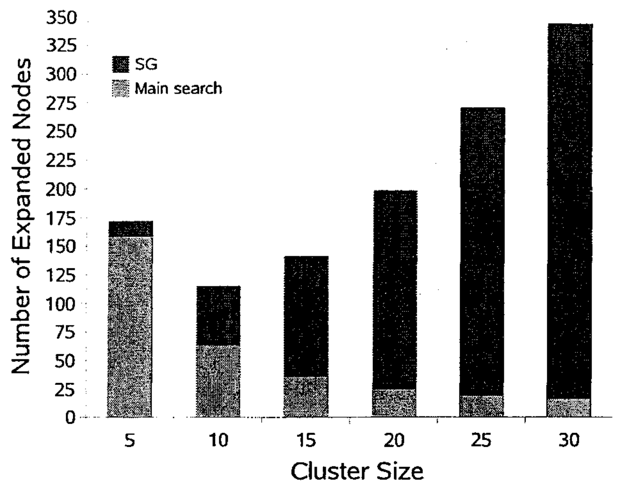


Figure 6.8: The search effort for finding an abstract solution.

preserved. Another idea is to consider for connection only border nodes that are in the direction of G. However, this does not guarantee completeness. If the search fails because of incompleteness, it should be restarted with a larger subset of border nodes.

The currently-used clustering method is simple and produces good results. However, more sophisticated strategies can be explored. For example, automatically minimize measures such as number of abstract clusters, cluster interactions, and cluster complexity (e.g., the percentage of internal obstacles).

An interesting topic is to extend HPA* to non-grid maps. Finally, experiments can be run on classes of problems characterized by either multiple agents, apriori unknown domains, or mobile targets. All these require replanning, and the ability of HPA* to save resources by postponing unneeded refinements could be very beneficial.

Chapter 7

Using Abstraction for Planning in Sokoban

Heuristic search has led to impressive performance in games such as Chess and Checkers. However, for some two-player games like Go, or puzzles like Sokoban, approaches based on low-level heuristic search are limited. Alternative approaches are needed to deal with such hard domains, where humans still perform much better than the best existing programs.

The Sokoban domain was described in Section 2.3. The problem is difficult for several reasons including deadlocks (positions from which no goal state can be reached), the large branching factor (can be over 100), long optimal solutions (can be over 600 moves), and an expensive lower-bound heuristic estimator which limits search speed. Sokoban problems are especially challenging since the domain is PSPACE-complete. Many problems are combinations of wonderful and subtle ideas, and finding their solution may require substantial resources – for humans and especially for computers.

Sokoban is so hard for computers that a standard algorithm such as A* would fail even on problems that humans can easily solve. Humans plan their moves at a high strategic level, rather than performing exhaustive search at the level of atomic actions. Based on this example, abstraction might be the answer to improve an automated solver. This chapter introduces *abstract Sokoban*, an approach that combines planning and abstraction. Ideas such as topological abstraction, hierarchical problem decomposition, and macro-moves, which are part of the overall theme of this thesis, are explored in an application-specific context.

Similarly to map decomposition in HPA*, a Sokoban maze is decomposed into *rooms* connected by *tunnels*, resulting in a two-level hierarchical representation of the problem. At the higher level of the hierarchy, a maze is seen as a small graph where nodes are rooms and edges are tunnels. The solving strategy is planned at this level, using abstract actions such as transferring a stone between two connected rooms, and rearranging the stones inside a room so that the man can cross it. Planning the solving strategy, also called the *global problem*, uses TLPlan [2], a standard planner. Details of abstract actions are solved at the low level of the hierarchy. Each room is assigned a *local problem* that deals with issues such as the stone configuration of that room, moves inside the room, and local deadlocks. Sokoban-specific functionality is implemented on top of Rolling Stone [49].

In Sokoban, the solution length can be defined in two ways: either man movements or stone pushes can be counted. Solutions in abstract Sokoban are not guaranteed to be optimal by either criterion. Giving up optimality allows for the definition of equivalence relationships between configurations of a given room or tunnel. Elements of an equivalence class are merged into one abstract local state, reducing the search space. If desired, non-optimal solutions can be improved in a post-processing phase.

The rest of this chapter is structured as follows: Section 7.1 contains a discussion of planning in Sokoban. Section 7.2 provides details about hierarchical problem abstraction in Sokoban. Section 7.3 presents experimental results, and Section 7.4 contains conclusions and ideas for further work.

7.1 Planning in Sokoban

The first part of this section focuses on how to formulate Sokoban as a standard planning problem. The impact of using domain-specific knowledge and abstraction is discussed. Three domain representations, each at a different level of abstraction, are considered. The conclusion of this analysis is that planning in Sokoban greatly improves as application-specific information and abstraction are used.

The second part explains why a planner such as TLPlan was chosen to address the global problem in abstract Sokoban.

7.1.1 Representing Sokoban as a Planning Problem

Several formulations of Sokoban as a planning domain are possible, depending upon factors such as the abstraction level and the application-specific information used. A first, naive approach is to use neither abstraction nor domain-specific knowledge. All properties of the domain are translated into a standard planning language such as STRIPS. For instance, a regular low-level move in Sokoban becomes an action in the planning domain. Previous planning experiments based on such a naive Sokoban representation showed poor performance even for very small problems [51, 66].

Planning in Sokoban significantly improves when domain-specific knowledge and an abstracted problem formulation are employed. The main Sokoban-specific functions implemented in this work deal with:

- Deadlock: Since deadlocks affect the search efficiency, a quick test to detect local deadlock patterns is used. Deadlocks are detected using Rolling Stone's database, which contains all local deadlock patterns that can occur in a 5x4 area [49]. Although this enhancement is an important gain, the problem of deadlocks is far from being solved.

- Heuristic evaluation function: Since the heuristic function has a big impact on the quality of a search algorithm, a custom heuristic, called *Minmatching*, was used. This is also reused from Rolling Stone [49].

- State equivalence with respect to the man's position: Suppose that two states have identical stone configurations but different man positions, and that the man can walk from one position to the other. The two states are equivalent, unless optimal solutions that minimize man movements are sought.

To illustrate how performance improves as more abstraction is used, two domain formulations, each with a different level of abstraction, are introduced in addition to naive Sokoban. *Tunnel* Sokoban is a partially abstracted representation, where all tunnels present in a maze are treated as atomic entities. All possible configurations of a tunnel are reduced to a few abstract states and planning actions such as parking a stone inside a tunnel or pushing a stone across a tunnel are defined as atomic actions. See Section 7.2.1 for details.

Tunnel Sokoban with the domain-specific functionality presented above is difficult for the planner. The system could not solve even moderately complex puzzles. Only one from the standard test suite of 90 problems [48] can be solved by this approach. Tunnel abstraction reduces the search space, but the reduction is not big enough to achieve reasonable performance. Moreover, although small deadlocks are detected, there are many larger deadlock patterns that still have to be dealt with. Hence further reducing the search space and dealing with deadlocks more efficiently are desired. For this reason, *abstract* Sokoban, which abstracts not only tunnels but also the rest of a maze, is introduced. Abstract Sokoban is described in detail in Section 7.2.

7.1.2 Using a Standard Planner in Sokoban

To solve the global problem in the two-level hierarchy of abstract Sokoban, the TLPlan [2] planner was chosen, primarily since it allows users to plug-in libraries that contain custom functions tailored for the application at hand. In addition, TLPlan supports utilizing domain-specific knowledge encoded with temporal logic formulas, as mentioned in Section 2.1.3. However, the ability of TLPlan to reason with temporal logic was not exploited in this Sokoban project. TLPlan is a forward chaining planner and implements several search strategies such as best-first search, depth-first search, and breadth-first search. In experiments, a best-first search algorithm with nodes ordered according to their heuristic (pure heuristic search) was used. The heuristic is Minmatching.

Custom code is necessary in two important parts of the planning model described in this chapter. First, domain-specific knowledge such as deadlock detection, heuristic state evaluation and state equivalence can efficiently be

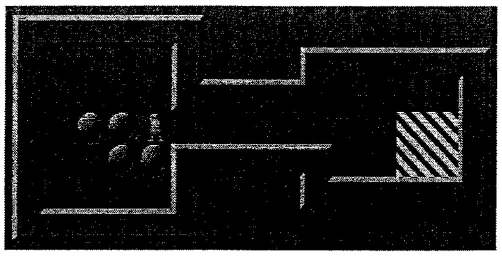


Figure 7.1: Toy Sokoban problem used as an example.

implemented. Second, custom functions can be used to model a hierarchical planning framework. In principle, hierarchical planning can be modeled in STRIPS, but this would result in a tremendous performance decay. When an abstract action such as transferring a stone from one room to another is applied, a Sokoban-specific function is called that verifies that the action is possible given the current state (i.e., check the action preconditions), maps the action to a sequence of low-level moves, and computes the changes on the maze (i.e., the action effects). This mechanism simulates hierarchical planning. Following standard terminology of hierarchical task networks [32], an abstract action is similar to a *nonprimitive task* and the associated custom function implements a *method* that tells how the task can be decomposed into a finer granularity level.

7.2 Abstraction in Sokoban

This section focuses on abstract Sokoban. Section 7.2.1 provides details on puzzle decomposition and abstract states of tunnels. Two-level hierarchical problem representation is discussed in Section 7.2.2. The following sections focus on one hierarchical level each: Section 7.2.3 describes room processing performed at the local level. Finally, Section 7.2.4 presents the global planning architecture. The toy problem shown in Figure 7.1 is used as a running example.

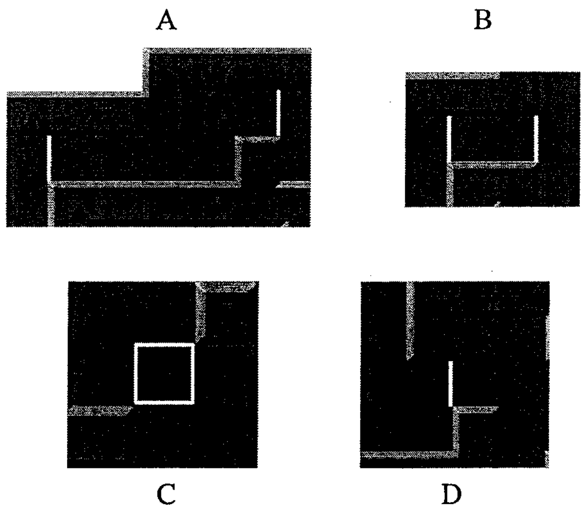


Figure 7.2: Various types of tunnels.

7.2.1 Puzzle Decomposition

Before decomposition, a simple preprocessing detects two types of "dead" squares. This is also performed in Rolling Stone. First, useless parts of a maze such as tunnels with one end closed are safely removed from the problem. Second, *stone-dead* squares, where the man can go but stones cannot be pushed because of deadlock, are marked.

A puzzle is decomposed in two steps. The first step is to identify its tunnels. Any contiguous sequence of interior (i.e., unblocked) tiles such that each tile has exactly two interior neighbours is a tunnel. Patterns A and B in Figure 7.2 are examples of such tunnels. The white lines that separate an end of a tunnel from the rest of the maze are called separation lines. In addition to the previous tunnel definition, the patterns C and D are considered tunnels too. Tunnel C contains one tile and four separation lines. It is the central tile of a 3×3 area where only two opposite corners are blocked. Tunnel D contains no tiles – only one separation line. It is created by a 3×2 area where only two opposite corners are blocked. These two patterns are useful when rooms are identified, since they act as room separators (see details below).

As a second step of puzzle decomposition, rooms are detected as areas separated by tunnels. All separation lines and interior tiles that already belong

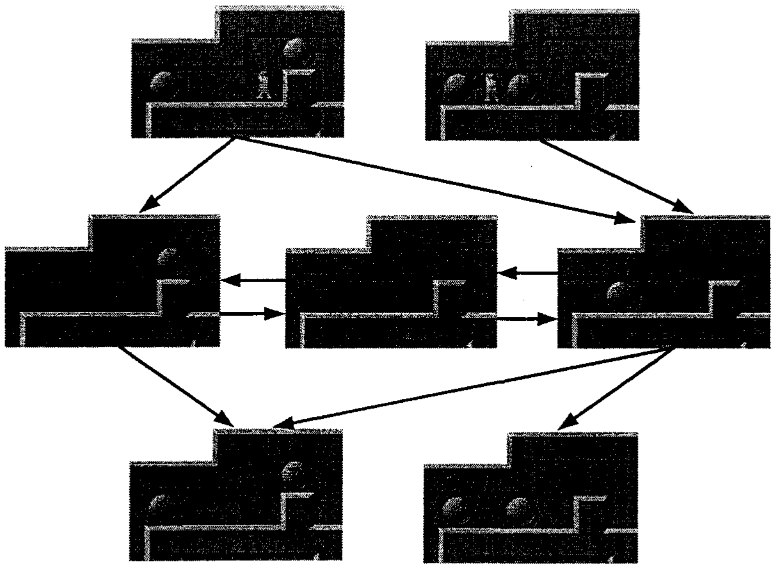


Figure 7.3: Abstract states of a tunnel.

to tunnels are considered walls. Rooms are maximal contiguous collections of interior tiles. The maze in Figure 7.1 decomposes into two rooms linked by a tunnel.

Tunnels are simple objects whose properties can be obtained with little computational effort. All stone configurations of a tunnel can be mapped to a few abstract states while preserving completeness. As an example, consider the top-left tunnel configuration in Figure 7.3. The left stone can be on any of the three squares at the left of the man, without changing abstract state of the tunnel.

Figure 7.3 shows the graph of abstract states and transitions for a tunnel. Each transition has preconditions that may depend on the rest of the maze. For instance, pushing a stone out of the tunnel is possible only if the configuration of the destination room allows it. The two states at the top can exist only in the initial state of a problem. The two states at the bottom are deadlock configurations (assume the man is outside the tunnel). The three states in the middle ignore the man position. Correctness is preserved by considering the man position in the preconditions of the transitions that initiate from these states.

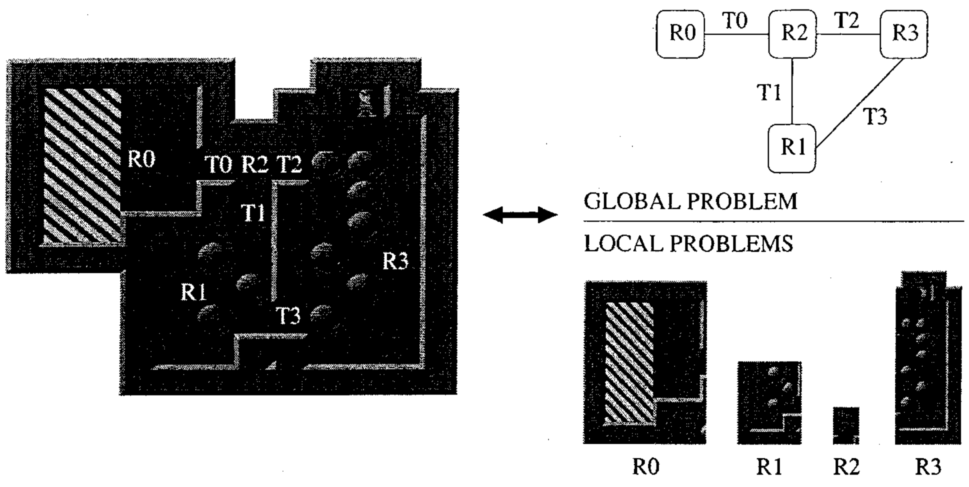


Figure 7.4: Hierarchical representation of a problem as a global component and a local component.

7.2.2 Hierarchical Problem Representation

Once the maze is split into rooms and tunnels, the initial problem can be decomposed into several smaller ones, as shown in Figure 7.4. At the global level, a maze is mapped into a graph (R_i, T_j), where the nodes R_i represent rooms and the edges T_j represent tunnels. A global planning problem focuses on how to transfer all stones to goal rooms through the graph. In addition, several local search problems, one for each room, are defined. The complexity of a local problem depends on both the size and the shape of a room. The local problem attached to the one-square room $R2$ is much simpler than the one attached to the largest room $R3$. While the complexity of the initial problem increases exponentially with the size of the maze, the complexity of the local problems increases exponentially with the size of the rooms only. Moreover, the results of local computation can be reused many times during the global-level search.

7.2.3 Local Problems

Local problems provide information about the preconditions, effects, and low-level refinements of global planning actions. In addition, they detect local deadlocks that can occur inside a room. The following paragraphs provide

1. Build local move graph;
2. Mark deadlock configurations with retrograde analysis;
3. Find strongly connected components;
4. Compute properties of each strongly connected component.

Figure 7.5: Local processing of a small room.

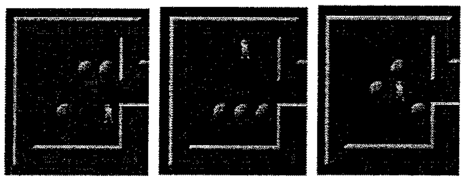


Figure 7.6: A few equivalent configurations of a room.

more details on local processing. *Small* rooms with no goal squares, *large* rooms with no goal squares, and goal rooms are separately discussed.

Small Rooms

For small rooms, with up to 15 non-dead squares, complete preprocessing is possible. Figure 7.5 summarizes the steps of preprocessing At step 1, the *local move graph* is computed. This includes all configurations, regardless of the number of stones, that can be reached from the initial configuration of the room.

In this work, a room configuration is called *deadlocked* if no path exists to the *goal* state in the local move graph. In general, a goal state of a room has one stone on each goal square and no stones on other squares. For rooms with no goal squares, a configuration is deadlocked if the room cannot be cleared of stones from that configuration. Otherwise, the configuration is *legal*. According to this definition, a legal room configuration does not exclude the existence of a larger-scale deadlock, involving a larger maze area.

At step 2, local deadlock configurations are detected in the local move graph. Positions are labeled as legal or deadlocked using retrograde analysis, starting from the empty position, which is marked as legal.

At step 3, a graph of abstract states and transitions is computed for each room, as in the case of tunnels. An abstract state of a room represents a collection of *equivalent* configurations. Two or more configurations are equivalent if they can be obtained from one another in such a way that neither the man nor any stone leaves the room. Merging several equivalent configurations into one abstract state greatly reduces the state space of a local problem. The abstract states of a room are computed as strongly connected components of the local move graph. In this computation, graph edges that involve interactions with the rest of the maze (i.e., the man or a stone leaving the room) are ignored. Figure 7.6 illustrates how one abstract state of the left room in the toy problem represents several equivalent configurations. This abstract state contains 45 equivalent configurations, but only three are shown in the picture.

At step 4, for each abstract state, predicates used to check action preconditions are also computed (e.g., "can push one more stone inside the room through entrance X"). When the value of such a predicate is true, several ways to accomplish the corresponding action can exist, each with a different resulting abstract state. However, in the prototype implementation used in experiments, only one such state is stored and used to update the problem state after performing an action. This speeds up search for the price of losing completeness. How to best balance this trade-off is an important open problem, whose more thorough study is left as future work.

Large Rooms

Local computation for large rooms is performed dynamically, as the planner requests new information, and consists of two main types of searches. *Action search* computes the preconditions, effects, and refinements of planning actions that involve a large room (e.g., transfer a stone from a large room to a tunnel). *Deadlock search* detects deadlocks that can occur in a room.

Action search implements a breadth-first strategy and includes the following enhancements. When the goal is to take out a stone, *pull macros*, which eliminate a stone without touching any other stone, are added as regular moves. A local transposition table, which is re-initialized for each local search, tells whether a given state has already been visited. During an action search, no stones leave or enter the room, except perhaps for a goal state of that search. Hence, only configurations with the same number of stones as the initial state have to be hashed. When the number of such configurations allows, a perfect hashing uniquely maps each configuration to one bit in the table. Otherwise, a classical hash table is used. At the global level, transposition tables store the results of precondition searches, so that they can be reused by the planner during the global search.

A deadlock search tries to take out all stones of a room configuration. As in action search, a local transposition table and pull macros are used as enhancements. When a pull macro can be applied, all the other moves generated from that position are safely ignored.

Two tables, L with legal configurations and D with *minimal* deadlock patterns, are used for prunning. A deadlock pattern d is minimal if

$$\forall s \prec d : s \text{ is legal.}$$

Relation $c_1 \prec c_2$ exists between two room configurations c_1 and c_2 if the first can be obtained from the latter by ignoring one or more stones. At the beginning, D is empty and L contains the configuration of the room in the initial problem state.

Assume a room configuration c is encountered in a deadlock search. If

$$\exists d \in D : d \prec c,$$

then c is deadlocked and no further expansion of this state is necessary. If desired, minimal deadlock patterns $d \prec c$ are detected and added to D. If

$$\exists l \in L : c \prec l,$$

then c is legal and hence the root state is legal. Legal configurations c so that $(\forall l \in L) : \neg(c \prec l)$ are added to L.

Goal Rooms

For a goal room with only one stone-traversable entrance (i.e., an entrance through which a stone can be pushed in and out), a reduced set of macro-actions that fill the goal squares is precomputed. Each time a new stone is pushed into the room, it is automatically placed on its designated position. A similar approach is implemented in Rolling Stone too. For a small goal room with multiple stone-traversable entrances, a complete preprocessing is performed, as in the case of small regular rooms. Many puzzles in the standard testset [48] have more general goal rooms than the ones described above. Coping with more types of goal rooms would be a major step in the effort of scaling the application to more complex puzzles.

7.2.4 Global Problem

The global problem is formulated as a planning problem. In the example in Figure 7.1, the objects are declared as follows:

```
(room 1000)
(room 1001)
(linear_tunnel 0).
```

Room 1000 is the leftmost one, and room 1001 is the goal room in the right.

The global state space S is a cross-product of the local state spaces of all rooms and tunnels:

$$S = S_1 \times S_2 \times ... \times S_k.$$

As this equation suggests, the local space reduction achieved with abstract states for tunnels and small rooms results in a global space simplification. In the example, global states are triples that describe the local states of the two rooms and the tunnel. The initial state is

```
(= (state 1000) 27)
(= (state 1001) 33)
(empty_tunnel 0).
```

The number that describes the state of a room (i.e., 27 and 33 in the example) is an index into an array of abstract states. At that index, a complete description of the state can be found, including the resulting abstract state of the room after an action has been applied. The linear tunnel in this problem can have only two legal abstract states: it either is empty or has a stone parked inside.

To express the goal state, one condition, which states the final state of the goal room, is enough: since all stones have to be on goal squares, it is obvious that the left room and the tunnel should be empty after the puzzle has been solved. The goal state is

```
(= (state 1001) 37).
```

Four types of actions are defined in this model:

- *Man-walk* takes as arguments one room and two entrances. This action is only considered when the man can reach the first entrance but cannot reach the other one. The result is to re-arrange the stones so that the man can cross the room from one entrance to another. This action is also defined for the case when the man is already inside a room and needs to leave via a particular entrance.
- *Room-to-room* transfers a stone from one room to another via a specified tunnel that links the two rooms.
- *Room-to-tunnel* takes a stone from a room and parks it in an adjacent tunnel.
- *Tunnel-to-room* takes a stone from a tunnel and pushes it to an adjacent room.

Rooms and tunnels involved in a stone movement change their abstract states after the corresponding action is completed. To be able to move one stone from one room to another, stones in both rooms may have to be re-arranged. The exact way to do this is computed at the local level.

In the example, the abstract solution is a sequence of 4 macro operators. In this case, each macro transfers one stone from room 1000 onto a free goal square in the goal room 1001, via the linear tunnel 0:

```
room-to-room 1000 0 1001
room-to-room 1000 0 1001
room-to-room 1000 0 1001
```

```
room-to-room 1000 0 1001.
```

Each abstract action has a corresponding sequence of atomic moves. For instance, the first abstract move consists of 12 stone pushes.

Compared to plain and tunnel Sokoban, the abstract representation shows greater promise for addressing the game as a planning problem. As will be shown in Section 7.3, some problems that cannot be solved by the first two approaches are easily handled in the abstract one. The improvement is explained by the hierarchical formulation, search space reduction, and deadlock detection.

A well-known property of hierarchical task networks [72] is that higher abstraction levels guide the planning at lower levels. A similar effect is present in abstract Sokoban: low-level searches have precise goals such as moving a stone from a room to another, or changing the local configuration so that the man can cross the room.

In abstract Sokoban the global search space is much smaller than in plain and tunnel Sokoban. Both branching factor and distance to a goal state are greatly reduced as a result of abstraction. Planning in abstract Sokoban is also simpler because there are fewer deadlocks to deal with. Deadlocks inside a room are detected by the local analysis. Still, large deadlocks that involve interactions between several rooms and tunnels remain undetected.

7.3 Experimental Results

This section describes experiments designed to empirically evaluate planning and abstraction in Sokoban. First, abstract Sokoban is compared to the state-of-the-art application-specific solver Rolling Stone. Second, to evaluate how planning in Sokoban improves as more abstraction is used, abstract Sokoban is compared against tunnel Sokoban. Experiments were run on 10 problems from the standard test suite [48]. These problems, shown in Appendix D, are the ones that can be solved by the abstract Sokoban system used in the experiments.

As in the case of abstract Sokoban, Rolling Stone also uses two types of search and, to perform a measurement, a one-to-one correspondence is considered between the search spaces in the two approaches. At the global level, Rolling Stone performs the so-called top-level search, whose purpose is to find a goal state. This is compared with the global planning in abstract Sokoban. There is also the pattern search in Rolling Stone, whose main goal is to determine deadlock patterns and find better bounds for the heuristic function [50]. Pattern search in Rolling Stone is compared with local preprocessing in abstract Sokoban, as they both are means to simplify the main search.

Figure 7.7 illustrates how abstraction reduces the depth of the global search in both abstract Sokoban and Rolling Stone. SP represents the number of stone pushes in the solutions found by Rolling Stone. In this experiment, SP estimates the depth of a search tree when no abstraction is used. RS is the length of the solutions found by Rolling Stone when tunnel macros and goal macros count as one step each. AS is the number of planning actions in solutions found in abstract Sokoban. AS is much smaller than SP, as one planning action in abstract Sokoban corresponds to several regular moves. The graph suggests that the global search space in abstract Sokoban is smaller than the main search space used in Rolling Stone. This is an important result, as it promises an exponential reduction in the search space.

When using tunnel Sokoban, TLPlan can seldom solve a problem entirely. In the 10-problem subset, only the simplest problem, which has 6 stones, can be solved. For this reason, comparison against tunnel Sokoban is made on subproblems of Sokoban puzzles. A subproblem is obtained by removing from the initial configuration some stones as well as an equal number of goal squares. Figure 7.8 shows results for solving subproblems of Problem #6. The number of expanded nodes in the main search is plotted on a logarithmic scale. Tunnel Sokoban is only able to solve subproblems with 7 or less stones. Compared to Rolling Stone, abstract Sokoban achieves a reduction that remains stable over the whole set of subproblems of Problem #6.

Table 7.1 presents a more detailed comparison between abstract Sokoban and tunnel Sokoban. Subproblem $x(y)$ is obtained from problem x by keeping y stones in the maze. The subproblems listed are the largest that tunnel Sokoban

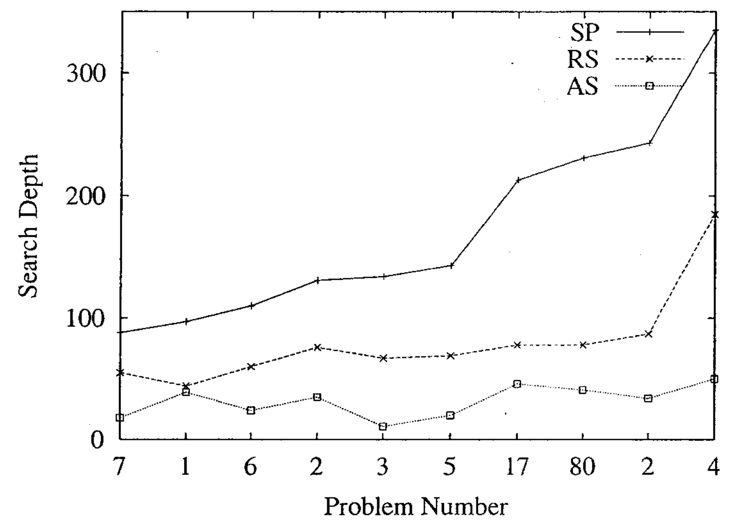


Figure 7.7: Depth of the main search in abstract Sokoban (AS) and Rolling Stone (RS). SP is the number of stone pushes.

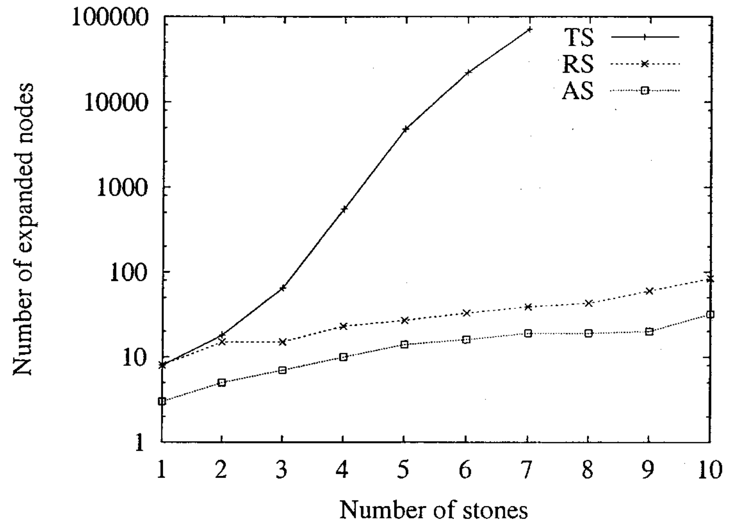


Figure 7.8: Nodes expanded in the main search for abstract Sokoban (*AS*), tunnel Sokoban (*TS*), and Rolling Stone (*RS*) for subproblems of Problem #6.

Subproblem	Abstract Sokoban PlN	PPN	Time	Tunnel Sokoban PlN	Time
1(6)	71	1,044	1.57	10,589	126.24
2(6)	24	61,113	0.93	80,740	9,490.21
3(7)	8	482	0.12	77,919	12,248.66
4(6)	9	41,065	0.80	27,514	3,061.94
5(6)	7	404	0.20	53,141	11,733.83
6(7)	19	54,317	1.06	71,579	8,189.77
7(8)	13	26,011	0.75	132	0.88
9(6)	13	245	0.25	35,799	4,883.55
17(5)	1,047	306,224	29.63	14,189	391.42
80(6)	10	395,583	3.02	14,266	949.98

Table 7.1: Abstract Sokoban vs. tunnel Sokoban.

Problem	Abstract Sokoban PlN	PPN	Time	Rolling Stone TLN	PSN	Time
1	71	1,044	1.57	50	1,042	**0.14**
2	635	62,037	16.10	80	7,530	**0.63**
3	12	19,948	2.04	87	12,902	**0.23**
4	128	69,511	**3.20**	187	50,369	3.27
5	36	297,334	23.14	202	43,294	**1.72**
6	36	54,414	1.37	84	5,118	**0.31**
7	54	35,813	1.57	1,392	28,460	**1.37**
9	35	7,607	**1.01**	1,884	436,801	22.17
17	8,091	444,073	166.98	2,038	29,116	**2.23**
80	47	877,914	4.56	165	26,943	**2.25**

Table 7.2: Abstract Sokoban vs. Rolling Stone.

can solve. *PlN* is the number of nodes expanded in the global search, and *PPN* are nodes in local room preprocessing. No local processing is performed in tunnel Sokoban. The time is measured in seconds. The data demonstrates a huge difference in terms of efficiency between the two approaches. Even if the values of *PPN* seem to be relatively large, preprocessing is fast, since no heuristic function has to be computed in a local search.

Table 7.2 shows a comparison between abstract Sokoban and Rolling Stone. For Rolling Stone, *TLN* is the number of nodes expanded in the top-level search and *PSN* is the number of expanded nodes in the pattern search. For many problems, the number of planning nodes *PlN* is smaller than *TLN*, which supports the claim that the global search space in abstract Sokoban is smaller than the one considered by Rolling Stone. In contrast, when comparing *PSN* and *PPN*, abstract Sokoban shows larger local searches, with the notable exception of problem #9. This is an effect of complete preprocessing of small rooms, even though only a small part of it will be required at the global planning level. Problem #9 shows that on-demand local computation can be very fast. This suggests that a better approach could be to compute local information on demand, as needed by the planner, for all (i.e., both small and large) rooms with no goal squares.

Rolling Stone is faster, with the exceptions of problems #4 and #9. Note that abstract Sokoban solves problem #9 20 times faster than Rolling Stone, for the reasons explained in the previous paragraph. The overhead of abstract Sokoban is determined by the local processing as well as the utilization of a general purpose planner. TLPlan uses a generic propositional representation of states, while the Sokoban-specific library represents states in a way that encodes knowledge about the domain. At each node in the main search, a conversion is made between the two representations, increasing the processing time per node. This is an inherent cost that has to be paid for using a generic planning engine. On the other hand, abstract Sokoban has the advantage that other planners that accept customized code can be used to solve the global planning problem, whereas Rolling Stone is a special-purpose system.

Abstract Sokoban can solve 10 problems from the standard set, while Rolling Stone solves 57. The difference is explained by the research and development effort invested in each system. Rolling Stone is a finely tuned application, developed in about two and a half years. Abstract Sokoban solved 10 problems after a development of about 6 months. In his thesis, Junghanns shows how the number of problems solved by Rolling Stone evolved as more effort was spent on research and development [49]. The data indicates that, after one year of effort, 12 problems were solved, with a jump from one problem to 12 problems within a two-month period at the half of the one-year interval. When Rolling Stone is restricted to a version based on a similar amount of effort as for abstract Sokoban, the two systems show similar performance in terms of number of problems solved.

To summarize the experiments, the results clearly show that abstract Sokoban is much more efficient than other planning representations of the game. No previous known planning attempts in Sokoban led to solving problems within the complexity range of the standard test suite [48]. In addition, abstract Sokoban is competitive with Rolling Stone on the 10-problem subset. However, parts of the abstracted architecture need improvement to scale up its performance. A few ideas are discussed in the next section.

7.4 Conclusions and Future Work

This chapter presented an approach that applies planning and abstraction to Sokoban. Abstract Sokoban is introduced as a hierarchical formulation of the domain obtained by decomposing a maze into rooms and tunnels. A global problem, solved with a standard planner such as TLPlan, provides a high-level solving strategy where stones are transferred between rooms and tunnels. Each room constitutes a local problem that solves the local constraints of abstract planning actions.

Many directions can be explored for future work. Many problems in the standard testset were not attempted because their goal rooms could not be processed with the current system. Better decomposition of a maze into rooms and tunnels is a challenging task that is expected to have great impact on the overall system performance. Treating several inter-connected rooms and tunnels as a single room can be beneficial, since all their interactions are removed from the global level. While the current decomposition process is quite rigid, it can be enhanced with a strategy aiming to optimize parameters such as the number of rooms and tunnels, and the interactions between rooms and tunnels. As pointed out previously, better study of the completeness is desirable. Finally, the global space can be further simplified by detecting deadlocks across several rooms and tunnels.

Chapter 8

Conclusions

Planning and heuristic search are fundamental areas of artificial intelligence reasearch, with a great number of potential real-life applications. Despite recent progress in these research areas, many problems of general interest remain too computationally challenging for the capabilities of current technology.

The topic of this thesis has been improving planning and search with automatic abstraction. Three frameworks, each with a different level of application-specific knowledge, served as testbeds for this research.

The first framework, domain-independent AI planning, is the topic of Chapters 3–5. A planner takes as input a domain and a problem expressed in a standard input language. Since one planner addresses many domains, including previously unseen ones, no additional application-specific knowledge can be provided by hand. This thesis introduced techniques that automatically learn new information about a domain and use it for faster planning in future problems. Empirical evaluation shows an improvement of orders of magnitude, as compared the state-of-the-art planner FF [42], in domains where specific knowledge can automatically be inferred. Participation in the international planning competition IPC-4 resulted in taking first place in 3 out of 7 attempted domains.

The second framework, path-finding on grid maps, is the topic of Chapter 6. Partial application-specific knowledge is assumed, since application domains in this class contain a topological structure that can be exploited by a solver. In principle, one program can tackle multiple applications with topological structure. Hierarchical Path-Finding A*, the main contribution to this domain, is shown to be up to 10 times faster in exchange for a 1% degradation in path quality, as compared to A*.

Chapter 7 has introduced an approach that applies planning and abstraction to Sokoban. No limitation is imposed on the amount of domanin-specific knowledge that can be used. A topological abstraction strategy decomposes a map into rooms connected by tunnels. This allows for the decomposition of a hard initial problem into several simpler sub-problems. A prototype implementation of abstract Sokoban is shown to be competitive with the state-of-the-art specialized solver Rolling Stone, on problems that the abstracted planning system can tackle.

Future work ideas were presented in previous chapters. In AI planning, ideas for new theoretical contributions are contained in Sections 3.4 and 4.4. A claim is made in Section 5.4 that planning research should faster expand from the narrow area of pure research towards solving more classes of real-life problems. Section 6.3 suggests ideas for improving the performance of HPA*. Similar ideas can be applied to related applications such as robot navigation, transportation, etc. Finally, Section 7.4 points out directions for future work in abstract Sokoban.

Bibliography

[1] F. Bacchus. AIPS'00 Planning Competition. *AI Magazine*, 22(3):47–56, 2001.

[2] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 16:123–191, 2000.

[3] F. Bacchus and Q. Yang. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence*, 71(1):43–100, November 1994.

[4] A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, (90):281–300, 1997.

[5] B. Bonet and H. Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[6] A. Botea. Macro-FF Website. `http://www.cs.ualberta.ca/~adib/macroff/`.

[7] A. Botea. Using Abstraction for Heuristic Search and Planning. In S. Koenig and R. Holte, editors, *5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Artificial Intelligence*, pages 326–327. Springer, August 2002.

[8] A. Botea. Reducing Planning Complexity with Topological Abstraction. In *ICAPS-03 Doctoral Consortium*, pages 11–15, Trento, Italy, June 2003.

[9] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-FF. In *Booklet of the Fourth International Planning Competition IPC-4*, pages 15–17, June 2004.

[10] A. Botea, M. Müller, and J. Schaeffer. Using Abstraction for Planning in Sokoban. In J. Schaeffer, M. Müller, and Y. Björnsson, editors, *3rd International Conference on Computers and Games (CG'2002)*, volume 2883 of *Lecture Notes in Artificial Intelligence*, pages 360–375, Edmonton, Canada, July 2002. Springer.

[11] A. Botea, M. Müller, and J. Schaeffer. Extending PDDL for Hierarchical Planning and Topological Abstraction. In *ICAPS-03 Workshop on PDDL*, pages 25–32, Trento, Italy, June 2003.

[12] A. Botea, M. Müller, and J. Schaeffer. Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, 1(1):7–28, 2004.

[13] A. Botea, M. Müller, and J. Schaeffer. Using Component Abstraction for Automatic Generation of Macro-Actions. In *Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04*, pages 181–190, Whistler, Canada, June 2004. AAAI Press.

[14] A. Botea, M. Müller, and J. Schaeffer. Learning Partial-Order Macros From Solutions. In *Fifteenth International Conference on Automated Planning and Scheduling ICAPS-05*, pages 231–240, Monterey, CA, USA, June 2005.

[15] A. Botea, M. Enzenberger M. Müller, and J. Schaeffer. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[16] T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

[17] D. Z. Chen, R. J. Szczerba, and J. J. Urhan Jr. Planning Conditional Shortest Paths Through an Unknown Environment: A Framed-Quadtree Approach. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and System Human Interaction and Cooperation*, volume 3, pages 33–38, 1995.

[18] A. Coles and A. Smith. Marvin: Macro Actions from Reduced Versions of the Instance. In *Booklet of the Fourth International Planning Competition*, pages 24–26, June 2004.

[19] J. Culberson. SOKOBAN is PSPACE-complete. Technical Report TR97-02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997. `ftp://ftp.cs.ualberta.ca/pub/TechReports/1997/TR97-02`.

[20] J. Culberson and J. Schaeffer. Efficiently Searching the 15-puzzle. Technical Report TR94-08, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1994.

[21] J. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, 14(4):318–334, 1998.

[22] S. Edelkamp. Website of the Fourth International Planning Competition IPC-4. `http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/`.

[23] S. Edelkamp. Planning with Pattern Databases. In *Proceedings of European Conference on Planning ECP-01*, pages 13–34, Toledo, Spain, 2001.

[24] S. Edelkamp. Symbolic Pattern Databases. In *Proceedings of International Conference on AI Planning and Scheduling AIPS-02*, pages 274–293, Toulouse, France, 2002.

[25] M. Enzenberger. Path Finding in Computer Games Website. `http://www.cs.ualberta.ca/~games/pathfind/`.

[26] K. Erol, J. Hendler, and Dana S. Nau. HTN Planning: Complexity and Expressivity. In *AAAI-94*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[27] A. Felner, U. Zahavi, J. Schaeffer, and R. Holte. Dual Lookups in Pattern Databases. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 103–108, Edinburgh, Scotland, July - August 2005.

[28] R. E. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 5(2):189–208, 1971.

[29] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI'99*, pages 956–961, 1999.

[30] M. Fox and D. Long. Extending the Exploitation of Symmetries in Planning. In *Proceedings of AIPS'02*, pages 83–91, 2002.

[31] M. Fox and D. Long. PDDL2.1: An Extension of PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[32] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning Theory and Practice.* Elsevier, 2004.

[33] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *EEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[34] M. Helmert. A Planning Heuristic Based on Causal Graph Analysis. In *Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04*, pages 161–170, Whistler, Canada, June 2004.

[35] M. Helmert and S. Richter. Fast Downward - Making Use of Causal Dependencies in the Problem Representation. In *Booklet of the Fourth International Planning Competition IPC-4*, pages 41–43, June 2004.

[36] B. Helmstetter and T. Cazenave. Searching with Analysis of Dependencies in a Solitaire Card Game. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 343–360, November 2003.

[37] I. Hernádvölgyi. Searching for Macro-operators with Automatically Generated Heuristics. In *Fourteenth Canadian Conference on Artificial Intelligence*, pages 194–203, 2001.

[38] J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *IJCAI-01*, pages 453–458, Seattle, Washington, USA, 2001.

[39] J. Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Sixth International Conference on Artificial Intelligence Planning and Scheduling AIPS-02*, pages 379–387, Toulouse, France, 2002.

[40] J. Hoffmann and S. Edelkamp. The Classical Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

[41] J. Hoffmann, S. Edelkamp, R. Englert, F. Liporace, S. Thiébaux, and S. Trüg. Towards Realistic Benchmarks for Planning: the Domains Used in the Classical Part of IPC-4. In *Booklet of the Fourth International Planning Competition*, pages 7–14, June 2004.

[42] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[43] R. Holte, C. Drummond, M. Perez, R. Zimmer, and A. MacDonald. Searching With Abstractions: A Unifying Framework and New High-Performance Algorithm. In *Proceedings of the Canadian Artificial Intelligence Conference*, pages 263–270, 1994.

[44] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up Problem Solving by Abstraction: A Graph Oriented Approach. *Artificial Intelligence*, 85:321–361, 1996.

[45] R. Holte, J. Newton, A. Felner, R. Meshulam, and D. Furcy. Multiple Pattern Databases. In *Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04*, pages 122–131, Whistler, Canada, June 2004. AAAI Press.

[46] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *AAAI-96*, pages 530–535, 1996.

[47] Glenn A. Iba. A Heuristic Approach to the Discovery of Macro-Operators. *Machine Learning*, 3(4):285–317, 1989.

[48] A. Junghanns. Sokoban Website. `http://www.cs.ualberta.ca/~games/Sokoban/`.

[49] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, University of Alberta, 1999.

[50] A. Junghanns and J. Schaeffer. Single-Agent Search in the Presence of Deadlock. In *AAAI-98*, pages 419–424, Madison, WI, USA, July 1998.

[51] A. Junghanns and J. Schaeffer. Domain-Dependent Single-Agent Search Enhancements. In *IJCAI-99*, pages 570–575, Stockholm, Sweden, August 1999.

[52] A. Junghanns and J. Schaeffer. Sokoban: Enhancing Single-Agent Search Using Domain Knowledge. *Artificial Intelligence*, 129(1–2):219–251, 2001.

[53] S. Kambhampati. *Machine Learning Methods for Planning*, chapter Supporting Flexible Plan Reuse, pages 397–434. Morgan Kaufmann, 1993.

[54] H. Kautz and B. Selman. Planning as Satisfiability. In *ECAI*, pages 359–363, 1992.

[55] Craig A. Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[56] R. Korf. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 97:97–109, 1985.

[57] R. Korf. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[58] R. Korf. Linear-Space Best-First Search. *Artificial Intelligence*, 62(1):41–78, 1993.

[59] R. Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 700–705, 1997.

[60] J. Kvarnström and P. Doherty. TALplanner: Temporal Logic Based Forward Chaining Planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[61] D. Long and M. Fox. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003. Special Issue on the 3rd International Planning Competition.

[62] S. Markovitch. Applications of Macro Learning to Path Planning. Technical report CIS9907, Technion, 1999.

[63] T. A. Marsland. A Review of Game-Tree Pruning. *International Computer Chess Association Journal*, 9(1):3–19, 1986.

[64] T. L. McCluskey and J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence*, 95:1–65, 1997.

[65] D. McDermott. PDDL, the Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998. `ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`.

[66] D. McDermott. Using Regression-Match Graphs to Control Search in Planning. *Artificial Intelligence*, 109(1–2):111–159, 1999.

[67] D. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35–55, 2000.

[68] S. Minton. Selectively Generalizing Plans for Problem-Solving. In *IJCAI-85*, pages 596–599, 1985.

[69] S. Minton. Learning Search Control Knowledge: An Explanation-Based Approach. Hingham, MA, 1988. Kluwer Academic Publishers.

[70] R. Mooney. Generalizing the Order of Operators in Macro-Operators. In *Fifth International Conference on Machine Learning ICML-88*, pages 270–283, June 1988.

[71] A. Moore, L. Baird, and L. Kaelbling. Multi-Value-Functions: Efficient Automatic Action Hierarchies for Multiple Goal MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '99)*, pages 1316–1323, Stockholm, Sweden, 1999.

[72] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[73] X. Nguyen and S. Kambhampati. Reviving Partial Order Planning. In B. Nebel, editor, *IJCAI-01*, pages 459–466, Seattle, Washington, USA, 2001.

[74] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[75] I. Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, (1):193–204, 1970.

[76] D. Precup, R. Sutton, and S. Singh. Planning with Closed-loop Macro Actions. In *Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems*, pages 70–76, 1997.

[77] S. Rabin. A* Aesthetic Optimizations. In Mark Deloura, editor, *Game Programming Gems*, pages 264–271. Charles River Media, 2000.

[78] S. Rabin. A* Speed Optimizations. In Mark Deloura, editor, *Game Programming Gems*, pages 272–287. Charles River Media, 2000.

[79] B. Reese and B. Stout. Finding a Pathfinder. `http://citeseer.nj.nec.com/reese99finding.html`.

[80] E. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings IJCAI-75*, pages 206–214, 1975.

[81] H. Samet. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. NATO ASI Series, Vol. F40, 1988.

[82] S. Shekhar, A. Fetterer, and B. Goyal. Materialization Trade-Offs in Hierarchical Shortest Path Algorithms. In *Symposium on Large Spatial Databases*, pages 94–111, 1997.

[83] B. Stout. Smart Moves: Intelligent Pathfinding. *Game Developer Magazine*, October/November 1996.

[84] A. Tate. Generating Project Networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[85] P. Tozour. Building a Near-Optimal Navigation Mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[86] M. Veloso and J. Carbonell. *Machine Learning Methods for Planning*, chapter Toward Scaling Up Machine Learning: A Case Study with Derivational Analogy, pages 233–272. Morgan Kaufmann, 1993.

[87] V. Vidal. A Lookahead Strategy for Heuristic Search Planning. In *Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04*, pages 150–159, Whistler, Canada, June 2004.

[88] V. Vidal. The YAHSP Planning System: Forward Heuristic Search with Lookahead Plan Analysis. In *Booklet of the Fourth International Planning Competition IPC-4*, pages 56–58, June 2004.

[89] D. Wilkins and M. desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.

[90] A. Yahja, A. Stentz, S. Singh, and B. Brummit. Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments. In *Proceedings, IEEE Conference on Robotics and Automation, (ICRA)*, pages 650–655, Leuven, Belgium, May 1998.

[91] P. Yap. Grid-Based Path-Finding. In R. Cohen and B. Spencer, editors, *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence*, pages 44–55, Calgary, Canada, May 2002.

Appendix A

Algorithmic Details of CA-ED

This appendix auguments Section 3 with the following details: The pseudocode of static graph construction is provided in Section A.1. Section A.2 describes how static facts are determined in domains with hierarchical types. Section A.3 shows pseudocode for the component abstraction method, in addition to the high-level description provided in Section 3.1.2.

A.1 Pseudocode of Static Graph Construction

Pseudocode for building the static graph of a planning problem is shown in Figure A.1. In the main method *buildStaticGraph*(), the first step is to identify static domain predicates. A predicate is static if no operator includes it among its effects. For simplicity, the STRIPS domain formulation is assumed so that each operator o has a list of add effects $Add(o)$ and a list of delete effects $Del(o)$. Unary facts and facts with two variables of the same type are ignored as discussed in Section 3.1.1.

The next step of the main method labels with "static" all facts in the initial problem state s_0 that are instantiations of static predicates. Finally, a static graph is generated based on the problem static facts. Each argument of a static fact becomes a node in the graph. Arguments of each static fact are linked pairwise by graph edges. Each edge is labeled with the name of the corresponding fact.

For simplicity, method *identifyStaticPredicates*() is called each time a static graph is built. However, an actual implementation can be optimized. The

```
void buildStaticGraph(Graph &$g$) {
   identifyStaticPredicates(&$stPreds$);
   identifyStaticFacts($stPreds$, &$stFacts$);
   for (each static fact $f \in stFacts$) {
      // add nodes to the graph
      for (each constant $c \in$ Args($f$)) {
         if ($c \notin$ Nodes(g))
            addNode($c$, &Nodes(g));
      }
      // add edges to the graph
      for (each $c_1, c_2 \in$ Args($f$), $c_1 \neq c_2$) {
         addEdge($c_1, c_2$, Name($f$), &Edges(g));
      }
   }
}
void identifyStaticPredicates(Preds &$stPreds$) {
   $stPreds = \emptyset$;
   for (each domain predicate $p$) {
      $static$ = true;
      if(Arity($p$) == 1 $\vee$ Symmetric($p$))
         continue;
      for (each domain operator $o$) {
         if($p \in$ Add($o$) $\cup$ Del($o$))) {
            $static$ = false;
            break;
         }
      }
      if($static$)
         $stPreds = stPreds \cup \{p\}$;
}
void identifyStaticFacts(Preds $stPreds$, Facts &$stFacts$) {
   $stFacts = \emptyset$;
   for (each fact $f \in s_0$) {
      if(Pred($f$) $\in stPreds$)
         $stFacts = stFacts \cup \{f\}$;
}
```

Figure A.1: Static graph construction in pseudo-code.

results of this method depend only on the current domain, not on the current problem instance. Therefore, it is enough to call it once and reuse its results for several instances in a domain.

A.2 Static Facts in Domains with Hierarchical Types

In a domain with hierarchical types, instances of the same predicate can be both static and fluent. Consider the Depots domain, a combination of Logistics and Blocksworld. This domain uses such a type hierarchy. Type LOCATABLE has four atomic sub-types: PALLET, HOIST, TRUCK, and CRATE. Type PLACE has two atomic sub-types: DEPOT and DISTRIBUTOR. Predicate (AT ?L - LOCATABLE ?P - PLACE), which indicates that object ?L is located at place ?P, corresponds to eight specialized predicates at the atomic type level. Predicate (AT ?P - PALLET ?D - DEPOT) is static, since there is no operator that adds, deletes, or moves a pallet. However, predicate (AT ?C - CRATE ?D - DEPOT) is fluent. For instance, the LIFT operator deletes a fact of this type.

To address the issue of hierarchical types, a domain formulation is used where all types are expressed at the lowest level in the hierarchy. Each predicate is expanded into a set of *low-level predicates* whose arguments have low-level types. Similarly, *low-level operators* have variable types from the lowest hierarchy level. Component abstraction and macro generation are done at the lowest level. After building the macros, the type hierarchy of the domain is restored. When possible, a set of two or more macro operators that have low-level types is replaced with one equivalent operator with hierarchical types.

A.3 Pseudocode of Component Abstraction

Figure A.2 shows pseudo-code for component abstraction, which identifies small clusters in a problem static graph g given as a parameter. *Types*(g) contains all types of the constant symbols used as nodes in g. Given a type t, *Preds*(t) is the set of all static predicates that have a parameter of type

```
componentAbstraction(Graph $g$) {
  for (each $t \in Types(g)$ chosen in random order) {
    resetAllStructures();
    $Open \leftarrow t$;
    for (each $c_i \in Nodes(g)$ with type $t$)
      $AC \leftarrow$ createComponent($c_i$);
    while ($Open \neq \emptyset$) {
      $t_1 \leftarrow Open$;
      $Closed \leftarrow t_1$;
      for (each $p \in Preds(t_1) \setminus Tried$)
        $Tried \leftarrow p$;
        if $\neg$(predConnectsComponents($p, AC$)) {
          extendComponents($p, AC$);
          for (each $t_2 \in Types(p)$)
            if ($t_2 \notin Open \cup Closed$)
              $Open \leftarrow t_2$;
        }
    }
    if (evaluateDecomposition() = OK)
      return $AC$;
  }
  return $\emptyset$;
}
```

Figure A.2: Component abstraction in pseudo-code.

t. Given a static predicate p, *Types*(p) includes the types of its parameters. *Facts*(p) are all facts instantiated from p.

Each iteration of the main loop tries to build components starting from a seed type $t \in$ *Types*(g). The sets *Open*, *Closed*, *Tried*, and *AC* are initialized to $\emptyset$. Each graph node of type t becomes the seed of an abstract component (method *createComponent*). The components are greedily extended by adding new facts and constants, such that no constant is part of any two distinct components. The method *predConnectsComponents*(p, AC) verifies if any fact $f \in$ *Facts*(p) merges two distinct abstract components in AC. If so, no fact from *Facts*(p) will be used for component extension.

Method *extendComponents*(p, AC) extends the existing components using all static facts $f \in$ *Facts*(p). For simplicity, assume that a fact f is binary and has constants c_1 and c_2 as arguments. In the most general case, four possible relationships can exist between the abstract components and elements f, c_1, and c_2:

1. Both c_1 and c_2 already belong to the same abstract component ac:

   $$\exists(ac \in AC) : c_1 \in Nodes(ac) \wedge c_2 \in Nodes(ac).$$

   In this case, f is added to ac as a new edge.

2. Constant c_1 is already part of an abstract component ac (i.e., $c_1 \in Nodes(ac)$) and c_2 is not assigned to a component yet. Now ac is extended with c_2 as a new node and f as a new edge between c_1 and c_2.

3. If neither c_1 nor c_2 are part of a previously built component, a new component containing f, c_1 and c_2 is created and added to AC.

4. Constants c_1 and c_2 belong to two distinct abstract components:

   $$\exists(ac_1, ac_2) : c_1 \in Nodes(ac_1) \wedge c_2 \in Nodes(ac_2) \wedge ac_1 \neq ac_2.$$

   While possible in general, this last alternative never occurs at the point where method *extendComponents* is called. This is ensured by the previous test with method *predConnectsComponents*.

The result is evaluated at the end of each iteration. If a good decomposition is found starting from t, the procedure returns with success. Otherwise, the process restarts from another seed type.

Consider the case when a static graph has two disconnected (i.e., with no edge between them) subgraphs sg_1 and sg_2 such that $Types(sg_1) \cap Types(sg_2) = \emptyset$. In such a case, the algorithm shown in Figure A.2 finds abstract components only in the subgraph that contains the seed type. To perform clustering on the whole graph, the algorithm has to be run on each subgraph separately.

Appendix B

Domains used in Planning Experiments

This appendix summarizes the planning domains used in experiments in Chapter 5. Rovers, Depots and Satellite were used in the third international planning competition IPC-3 [61]. Satellite, Promela, Airport, PSR and Pipesworld were benchmarks in the fourth competition IPC-4 [40, 41].

B.1 Rovers

In the Rovers domain, rovers can be equipped with photo cameras and stores where rocks and soil can be collected and analyzed. Rovers have to gather pictures and data about rock and soil samples, and report them to their base. Waypoints and connections between them define a map on which rovers navigate between locations of interest. Such locations include waypoints that contain rock and/or soil samples, waypoints that photo objectives are visible from, and waypoints that allow communication with the base.

B.2 Depots

In Depots, crates have to be transported by truck between locations of two types: depots and distributors. A truck can move between any two locations in one step and transport any number of crates at a time. Each location has one or more pallets, where crates can be stacked, and one or more hoists that can transfer a crate from a truck to the top of a stack and back. A hoist can hold at most one crate at a time. To transfer a crate from a truck to a stack or back, the stack, the hoist and the truck have to be at the same location.

B.3 Satellite

In the Satellite domain, satellites have instruments that can take pictures in different modes. When a satellite is equipped with several instruments, only one instrument can be powered on at a time. A satellite together with an instrument on board can take an image of an objective in a given mode when the satellite is oriented into the direction of the objective, and the instrument is calibrated, powered on, and supports that picture mode.

B.4 Promela

Promela is the input language of a model checker called SPIN [40]. A model defined in Promela is a set of processes (i.e., automata) that communicate through message queues. A Promela planning problem is a PDDL representation of a Promela model. Promela Dining Philosophers and Promela Optical Telegraph, the two domains used in IPC-4 and in this thesis research, are PDDL adaptations of two original Promela models.

B.5 Airport

The goal of an Airport problem is to schedule the incoming and outgoing traffic on an airport. The topology of an airport is modeled as a set of segments and an adjacency relationship between segments. A segment can host one plane at a time. If the engines of a plane are running, one or several segments behind the plane cannot be occupied by another plane.

The available actions in this domain are to move an airplane between two adjacent segments, to start or stop the engines of a plane, to push a plane back from its parking position, and to take off.

B.6 Power Supply Restoration

Power Supply Restoration (PSR) models a power distribution network where electric lines are connected by switches that can be opened or closed. One or several power sources provide the network with electricity. When an electric line becomes faulty, its power source is disconnected from the network and all lines supplied by this source lose power. The goal of a PSR problem is to restore the power supply on all non-faulty lines by changing the status of network switches.

B.7 Pipesworld

In Pipesworld, batches of different types of oil products have to be transported through a network of pipes and reservoirs. A pipe contains a constant number of batches. Inside a pipe, two batches can be adjacent only if their types are compatible with each other. When a batch is pushed in at one end of a pipe, all batches inside the pipe are shifted and the batch at the other end is pushed out.

Several versions of Pipesworld were introduced in IPC-4. See [40] for details. This thesis work contained experiments with two versions of this domain: Pipesworld Notankage Nontemporal and Pipesworld Tankage Nontemporal. In the first version, reservoirs have unlimited capacity, whereas tankage restrictions exist in the latter version.

Appendix C

Algorithmic Details of HPA*

This appendix provides low-level details about HPA*, including the main functions in pseudo-code. The code can be found at [25]. First preprocessing, which abstracts a grid map into a multi-level graph is presented. Then details about on-line search, which performs hierarchical search in a multi-level graph are provided.

C.1 Preprocessing

Figure C.1 summarizes the preprocessing. The main method is *preprocessing*(), which abstracts a map, builds a graph with one abstract level and, if desired, adds more levels to the graph.

C.1.1 Abstracting the Maze and Building the Abstract Graph

In the initial stage, abstraction consists of building the 1-clusters (i.e., clusters at level 1) and the entrances between clusters. Later, when more levels are added to the hierarchy, the maze is further abstracted by computing clusters of higher levels. In method *abstractMaze*(), $C[1]$ is the set of 1-clusters, and E is the set of all entrances defined for the map.

Method *buildGraph*() creates the abstract graph of the problem. First it creates the nodes and the inter-edges, and next builds the intra-edges. Method *newNode*(e, c) creates a node contained in cluster c and placed at the middle of entrance e. For simplicity, assume there is one transition per

```
void preprocessing(int $maxLevel$) {
  abstractMaze();
  buildGraph();
  for ($l = 2; l \leq maxLevel; l{+}{+}$)
    addLevelToGraph($l$);
}

void abstractMaze(void) {
  $E = \emptyset$;
  $C[1]$ = buildClusters(1);
  for (each $c_1, c_2 \in C[1]$) {
    if (adjacent($c_1, c_2$))
      $E = E \cup$ buildEntrances($c_1, c_2$);
  }
}

void addIntraEdges(int $l$) {
  for (each $c \in C[l]$)
    for (each $n_1, n_2 \in N[c], n_1 \neq n_2$) {
      $d$ = searchForDistance($n_1, n_2, c$);
      if ($d < \infty$)
        addEdge($n_1, n_2, l, d$, INTRA)
    }
}

void buildGraph(void) {
  for (each $e \in E$) {
    $c_1$ = getCluster1($e$, 1);
    $c_2$ = getCluster2($e$, 1);
    $n_1$ = newNode($e, c_1$);
    $n_2$ = newNode($e, c_2$);
    addNode($n_1$, 1);
    addNode($n_2$, 1);
    addEdge($n_1, n_2$, 1, 1, INTER);
  }
  addIntraEdges(1);
}

void addLevelToGraph(int $l$) {
  $C[l]$ = buildClusters($l$);
  for (each $c_1, c_2 \in C[l]$) {
    if (adjacent($c_1, c_2$)) {
      for (each
          $e \in$ getEntrances($c_1, c_2$)) {
        setLevel(getNode1($e$), $l$);
        setLevel(getNode2($e$), $l$);
        setLevel(getEdge($e$), $l$);
      }
    }
  }
  addIntraEdges($l$);
}
```

Figure C.1: The preprocessing phase in pseudo-code. This phase builds the multi-level graph, except for S and G.

entrance, regardless of the entrance width. Methods *getCluster1*(e, l) and *getCluster2*(e, l) return the two adjacent l-clusters connected by entrance e. Method *addNode*(n, l) adds node n to the graph and sets the node level to l. Method *addEdge*(n_1, n_2, l, w, t) adds an edge between nodes n_1 and n_2. Parameter w is the weight, l is the level, and $t \in \{INTER, INTRA\}$ the type of the edge.

Method *searchForDistance*() searches for a path between two nodes and returns the path cost. This search is optimized as shown in Section C.2.2.

C.1.2 Creating Additional Graph Levels

The hierarchical levels of the multi-level abstract graph are built incrementally. Level 1 has been built at the previous phase. Assuming that the highest current level is $l - 1$, level l is built by the method *addLevelToGraph*(l). Groups of clusters at level $l-1$ form a cluster at level l in method *buildClusters*(l), $l > 1$. $C[l]$ is the set of l-clusters.

C.2 On-line Search

C.2.1 Finding an Abstract Solution

Figure C.2 summarizes the steps of the on-line search. The main method is *hierarchicalSearch*$(S, G, maxLevel)$, which performs the on-line search. First S and G are inserted into the abstract graph, using method *insertNode*(n, l). Method *connectToBorder*(n, c) adds edges between node n and the nodes placed on the border of cluster c that are reachable from n. Method *determineCluster*(n, l) returns the l-cluster that contains node n.

Method *searchForPath*$(S, G, maxLevel)$ performs a search at the highest abstraction level to find an abstract path from S to G. If desired, the path is refined to a low-level representation by method *refinePath*$(absPath)$. Finally, method *smoothPath*$(llPath)$ improves the quality of the low-level solution.

```
void connectToBorder(node $s$, cluster $c$) {
  $l$ = getLevel($c$);
  for (each $n \in N[c]$)
    if (getLevel($n$) $\geq l$) {
      $d$ = searchForDistance($s, n, c$);
      if ($d < \infty$)
        addEdge($s, n, d, l$, INTRA);
    }
}
void insertNode(node $s$, int $maxLevel$) {
  for ($l = 1; l \leq maxLevel; l$++) {
    $c$ = determineCluster($s, l$);
    connectToBorder($s, c$);
  }
  setLevel($s, maxLevel$);
}

path hierarchicalSearch(node $s, g$, int $l$) {
  insertNode($s, l$);
  insertNode($g, l$);
  $absPath$ = searchForPath($s, g, l$);
  $llPath$ = refinePath($absPath, l$);
  $smPath$ = smoothPath($llPath$);
  remove($s$);
  remove($g$);
  return $smPath$;
}
```

Figure C.2: On-line processing in pseudo-code.

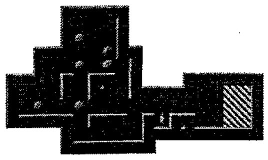
C.2.2 Searching in a Multi-Level Graph

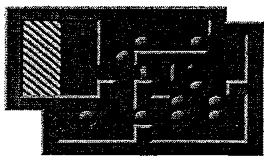
In a multi-level graph, search can be performed at various abstraction levels. Searching at level l reduces the search effort by exploring only a small subset of the nodes. The higher the level, the smaller the part of the graph that can potentially be explored. When searching at a certain level l, only nodes having level $\geq l$, intra-edges having level l, and inter-edges having level $\geq l$ are considered.

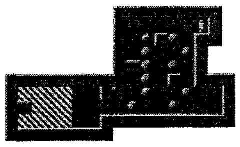
The search space is further reduced by ignoring the nodes outside a given cluster. This applies to situations such as connecting S or G to the border of their clusters, connecting two nodes placed on the border of the same cluster, or refining an abstract path.

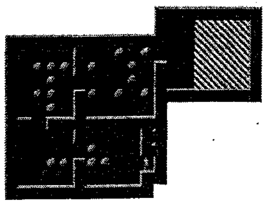
Appendix D

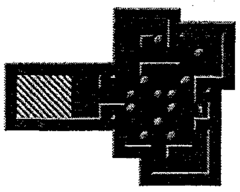
Sokoban Test Suite

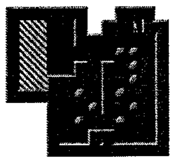

Problem #1

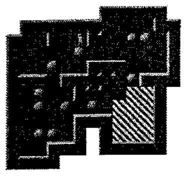

Problem #2

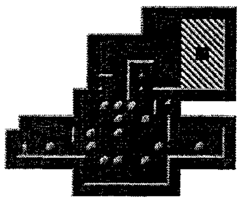

Problem #3

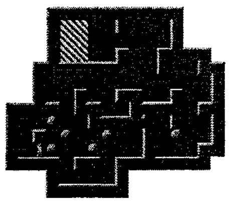

Problem #4

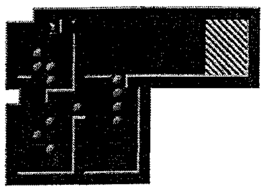

Problem #5


Problem #6


Problem #7


Problem #9


Problem #17


Problem #80