Extending PDDL for Hierarchical Planning and Topological Abstraction

Adi Botea and **Martin Müller** and **Jonathan Schaeffer**
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{adib,mmueller,jonathan}@cs.ualberta.ca

Abstract

Despite major progress in AI planning over the last few years, many interesting domains remain challenging for current planners. Topological abstraction can reduce planning complexity in several domains, decomposing a problem into a two-level hierarchy. This paper presents LAP, a planning model based on topological abstraction. In formalizing LAP as a generic planning framework, the support of a planning language more expressive than PDDL can be very important. We discuss how an extended version of PDDL can be part of our planning framework, by providing support for hierarchical planning and topological abstraction. We demonstrate our ideas in Sokoban and path-finding, two domains where topological abstraction is useful.

Introduction

AI planning has recently achieved significant progress in both theoretical and practical aspects. The last few years have seen major advances in the performance of planning systems, in part stimulated by the planning competitions held as part of the AIPS series of conferences (McDermott 2000; Bacchus 2001; Fox & Long 2002). However, many hard domains still remain a great challenge for the current capabilities of planning systems.

Abstraction is a natural approach to simplify planning in complex problems. For instance, humans often create abstract plans that they try to follow during their search. In this paper we present topological abstraction, a technique for reducing planning complexity in hard domains. Based on this abstraction model, we also discuss extending the PDDL language so that it supports hierarchical planning and abstraction. In exploring how PDDL can be extended, the meaning of "abstraction" can be more general than our topological approach. Topological abstraction is only one of a general class of hierarchical relationships that PDDL may not be able to express well.

Our abstraction approach reformulates the state representation, grouping related low-level features in local clusters. The clustering aims to catch local relationships inside clusters and keep cluster interactions as low as possible. In effect, the initial problem is decomposed into a two-level hierarchy of sub-problems, each being much simpler than the initial one. At the local level, each cluster has associated a local problem that solves the local constraints. There is also a global planning problem where clusters are treated as black-boxes and local state features are hidden away. The reason why we call this topological abstraction is that an important class of applications of which the approach is suitable have a spatial structure such as a grid. In such a domain, we group atomic grid squares into abstract clusters such as rooms in a building.

Motivation

Many interesting domains are hard to deal with when no abstraction is present. Examples of such domains are Sokoban and path-finding. In these domains, a hierarchical problem decomposition based on topological clustering can lead to significantly better performance. Our preliminary work using these domains as a testbed has already shown an impressive potential of the topological abstraction.

Sokoban is a puzzle with many similarities to a robotics application. In this domain, a man in a maze has to push stones from their initial positions to designated destinations called *goal squares* (see Section 3 for a detailed description of the rules). Both the AI planning and the single-agent search communities agree that this is a hard domain. The game is difficult for a computer for several reasons including deadlocks (positions from which no goal state can be reached), the large branching factor (can be over 100 – if we consider as moves all the stone pushes in the man reachable area), and long optimal solutions (can be over 600 moves). Another problem is that all known lower-bound heuristic estimators for the solution length are either of low quality, or expensive to compute.

Humans, who solve Sokoban puzzles much easier than state-of-the-art AI applications, abstract the maze into rooms and tunnels and use this high-level representation to create abstract plans. Following the humans' example, an AI application can cluster atomic squares into more abstract features such as rooms connected by tunnels, reducing the complexity of the hard initial problem. In effect, a large number of atomic squares is replaced by a few abstract, more meaningful features such as rooms and tunnels.

In the domain of path-finding, an agent on a map has to find a (shortest) path from its current position to a destination position. The map topology can have many forms, such as a battlefield, the interior of a building, etc. The problem is important in commercial computer games, robot planning, military applications, etc. The efficiency of the path-finding algorithms is often crucial, as they have to produce solutions in real-time and use limited resources. The classical solving strategy represents the maze as a grid of atomic cells and uses a search algorithm such as A* on that graph. An action is to move to an adjacent cell that is not part of an obstacle. The representation of states in the search space greatly influences the efficiency of the search. A fine granularity of the map leads to a large search space, requiring serious time (and possibly space) resources. A much more efficient problem representation is to abstract the map into connected clusters such as rooms, large obstacle-free areas, bridges, etc. As in Sokoban, the abstract map representation is a small graph of connected clusters, with a much reduced search space.

To the best of our knowledge, in some commercial games the search space is abstracted by human experts, who define the abstract clusters by hand. Our contribution is an automated abstraction method, especially useful when human expertise is expensive or not available. For instance, a bomb can destroy a bridge, changing the landscape dynamically and invalidating the previous abstract representation. Also, the user of a game may be allowed to define new map configurations, which have to be abstracted from scratch.

In standard planning domains such as Logistics, topological abstraction of the real world is part of the domain definition. In Logistics, several packages have to be transported from their initial location to various destinations. A Logistics problem has a map of cities connected by airline routes. Transportation inside cities can be done by truck (there is one truck in each city). Cities are abstracted, being treated as black boxes. Inside a city, a truck can go from any point to any destination at no cost. However, in the real world, transportation within a city is a subproblem that can involve considerable costs. In this context, removing human expertise and automatically obtaining abstracted models of the real world is an important research problem.

The Planning Language Support

Topological abstraction is appropriate for several application domains. Our goal is to build a general planning framework where topological abstraction is automatically performed for different planning domains. In such a framework, the robustness of the planning language used to describe the domain and problem instances is very important. Many parts of our abstraction framework could more easily be expressed when using a more general planning language. The language support for hierarchical planning in general should deal with representing the abstraction levels, and modeling relationships and communication across the levels. The language support for abstraction should cover several issues, such as problem reformulation, automatic abstraction, adaptive abstraction, and a hybrid problem representation. In this paper, *hybrid representation* refers to using both low-level features (for the part of the problem representation space not abstracted yet) and abstract features (for the already abstracted part of the space) to represent a problem state. In our framework, problem reformulation means to replace a low-level domain and problem representation by an equivalent abstract representation, which is easier to solve. We want to represent the abstracted problem explicitly, as an independent planning problem written in a language such as PDDL. This allows solving the abstract problem with no interaction with the initial low-level formulation. Another advantage of representing abstraction as part of the PDDL formulation is that, at one moment, we can use a hybrid state representation, using both low-level and abstract features for state description. When planning is done repeatedly in a fixed environment, an adaptive abstraction, which is performed as the system learns more about the environment, is also valuable. For instance, in a path-finding problem the map is initially represented at the low-level. An adaptive abstraction algorithm builds the clusters gradually, as the planning agent discovers more and more parts of the map. Before the abstraction is completed, the planning is done using a state representation composed of both atomic squares (for the unexplored parts of the map) and abstract clusters (for the explored parts of the map).

Adaptive abstraction can naturally be related to planning with uncertainty. We can consider that the part of the problem not abstracted yet is in a sense unknown to the planner. Using abstraction this way also required the domain description (or, more generally, the planning and plan execution framework) to handle uncertainty. PDDL currently doesn't do this, and topological abstraction can't handle this without a treatment of uncertainty. Even if this is an interesting topic, in this paper we don't focus on how an extended PDDL can be used to better handle uncertainty. We keep our discussion limited to hierarchical planning and abstraction issues.

The rest of the paper is structured as follows: In the next section we review the related work. In the third section we highlight our abstraction framework and briefly describe how we applied it to Sokoban and path-finding. We point out some features that can easier be addressed using a more robust planning language. In the fourth section we discuss extending PDDL to support hierarchical planning and topological abstraction. The last section presents our conclusion.

Related Work

Abstraction is a frequently used technique to reduce problem complexity in AI planning. Automatically abstracting planning domains has been explored by Knoblock (Knoblock 1994). His approach builds a hierarchy of abstractions by dropping literals from the problem definition at the previous abstraction level. Bacchus and Yang define a theoretical probabilistic framework to analyze the search complexity in hierarchical models (Bacchus & Yang 1994). They also use some concepts of that model to improve Knoblock's abstraction algorithm. In this work, the abstraction consists of problem relaxation. In our approach, abstraction means to reformulate a problem into an equivalent hierarchical representation. The abstract problem is solved independently from the initial problem formulation.

Long *et al.* use *generic types* and *active preconditions* to reformulate and abstract planning problems (Long, Fox, & Hamdi 2002). As a result of the reformulation, subproblems of the initial problem are identified and solved by using specialized solvers. Our approach has similarities with this work. Both formalisms try to cope with domain-specific features at the local level, keeping the global problem as generic as possible. The difference is that we reformulate problems as a result of topological abstraction, whereas in the cited work reformulation is obtained by identifying various generic types of behavior and objects such as *mobile objects*.

Using topological abstraction to speed-up planning in a reinforcement learning framework has been proposed in (Precup, Sutton, & Singh 1997). In this work, the authors define macro actions as *offset-casual* policies. In such a policy, the probability of an atomic action depends not only on the current state, but also on the previous states and atomic actions of the policy. Learning macro actions in a grid robot planning domain induces a topological abstraction of the problem space.

Previous experiments showed that planning in a low-level Sokoban formulation was too hard for state-of-the-art generic planners (McDermott 1997; Junghanns & Schaeffer 1999). Culberson performed a theoretical analysis of Sokoban, showing that this domain is PSPACE-complete (Culberson 1997). The state-of-the-art Sokoban solvers are Junghanns' *Rolling Stone* (Junghanns 1999; Junghanns & Schaeffer 2001) and *deep green*, developed inside the Japanese Sokoban community (Junghanns 1999). These applications can find solutions for two thirds of the standard 90-problem test suite [1].

Local Abstraction in Planning

In this section we present an overview of our abstraction model, called LAP (Local Abstraction in Planning). We also show how the model can be applied to domains such as Sokoban and path-finding. We use the model and these domains as a basis to motivate the need for a PDDL extension supporting hierarchical planning and abstraction. Thus, implementation details and analysis of experimental results are not our focus here. We rather consider issues such as hierarchical planning, automatic clustering, adaptive clustering, and hybrid state representation.

The Model Overview

LAP is a planning model based on a topological abstraction of the state representation. A clustering of the problem representation space is used to group related low-level features. The goal of the clustering process is to group together related atomic pieces and keep cluster interactions low. The abstraction allows us to decompose the initial problem into a hierarchy of sub-problems in a divide-and-conquer manner. For each cluster we define a local problem, which solves the local constraints of that cluster. The global problem uses an abstract problem description, where global states are characterized by states of abstract features. Each feature is a cluster that represents several atomic elements of the space.

At the global level, our abstraction approach leads to a much more compact state representation. For instance, a *room* in a robot planning domain is an abstract feature encoding many low-level objects such as atomic-size *squares*. Since one cluster is a complex feature representing several atomic features, cluster states can have many possible values. It is therefore natural to represent the global abstract states as tuples of cluster values. Using this representation, our abstraction model can be defined as a special case of the Simplified Action Structures (SAS) model (Bäckström & Klein 1991; Bäckström & Nebel 1995). In a SAS model, the global state space is a cross product of sub-spaces describing states of the problem features. Actions have associated three types of feature sets: *precondition* sets, *effect* sets, and *prevail* sets. The precondition set identifies the features used to check the action preconditions, the effect set contains the features whose states are changed by the considered action, and the prevail set contains the features whose values are preserved after the action has been applied. Below we point out the properties that differentiate our model from other existing SAS structures. First, in the LAP formalism, an abstract action changes either one state component or two components, leaving the rest of the tuple unchanged. In other words, the planning agent is only allowed to do local processing inside a cluster or perform an action affecting two clusters. Second, a transition between two clusters is possible only if the two clusters are neighbors. Third, when checking the preconditions of an operator, the only preconditions that matter are the values of the components that are changed by the action effect. For instance, in the Sokoban domain, when transferring a stone between two adjacent rooms A and B, the local stone configuration of other rooms is not relevant.

At the global level, we use abstract planning actions called macro-actions. Checking the preconditions of a macro-action uses cluster states rather than states of atomic features. The effects of a macro-action also change cluster states. The model does not guarantee the solution optimality. If for each action of an optimal abstract solution we compute an optimal sequence of atomic moves, the resulting low-level solution is not guaranteed to be optimal.

LAP in Sokoban

Sokoban is a single player game created in Japan in the early 1980s. Figure 1 shows an example of a Sokoban problem. The puzzle consists of a maze which has two types of squares: inaccessible *wall squares* and accessible *interior squares*. Several *stones* are initially placed on some of the interior squares. There is also a *man* that can walk around by moving from his current position to any adjacent *free* interior position. A free position is an interior square that is not occupied by either a stone or the man. If there is a stone next to the man and the position behind the stone is free, then the man can *push* the stone to that free square. The man moves forward to the initial position of the stone. The goal of the puzzle is to push all the stones to some specific marked interior positions called *goal squares*.

The game is difficult for a computer for several reasons including deadlocks, the large branching factor, and long optimal solutions. Also, all known lower-bound heuristic estimators are either of low quality, or expensive to compute.

[1] The test suite is available at `http://xsokoban.lcs.mit.edu/xsokoban.html`.

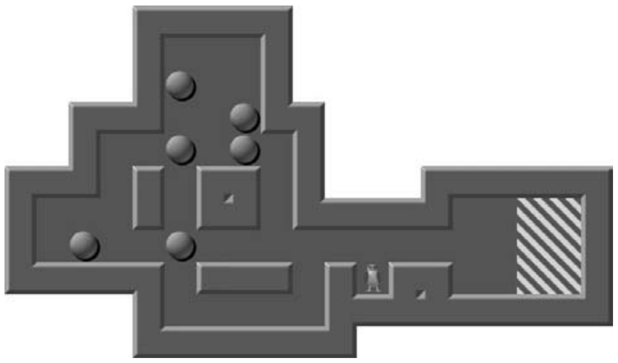

Figure 1: Problem #1 in the standard 90 problem Sokoban test suite. The six goal squares are the marked ones at the right end of the maze.

A simple planning representation of Sokoban can be obtained by translating all the low-level properties of the game into a planning language such as PDDL. For instance, a one-square push becomes one planning action. We call this *plain Sokoban*. This simple representation leads to very poor results, as it does not allow for an efficient handling of the long-range properties of the game. The domain formulation can be significantly improved by applying the LAP model to plain Sokoban. We call the new abstracted formulation *abstract Sokoban*. In abstract Sokoban, the clustering produces a decomposition of the maze into *rooms* and *tunnels*. Rooms and tunnels become clusters in the abstract representation. If there are k clusters extracted from the maze, the abstract description of a state is a tuple of integer values

$$s = (s_1, s_2, ..., s_k)$$

where s_i represents the internal state of cluster i. The internal state of a cluster is a complete description of the stone configuration and, in the case of rooms, the area reachable by the man inside that room. The abstract planning actions are: (a) to re-arrange stones inside a room, so that the man can walk between two designated entrances and (b) to transfer a stone between two rooms, or between a room and a tunnel. In case (a) we have *unary* operators, as they only change the status of one cluster. In case (b) we have *binary* operators, as they change the status of two adjacent clusters.

Figure 2 illustrates how our abstraction model works in Sokoban. At the global level, a search problem is transformed into a graph (R_i, T_j), where the nodes R_i represent rooms and the edges T_j represent tunnels. In effect, the global problem has a much smaller search space. Besides the global planning problem, we also obtain several local search problems, one for each room. Local problems check the action preconditions for the global planning problem. For instance, if the abstract action is to transfer a stone from room A to room B, we have to check that the local stone configurations allow this macro-action. Tunnels are so simple that the associated local problems are trivial.

State of the art in solving Sokoban is about 60 out of 90 problems solved by *Rolling Stone* and *deep green*. Our system, called *Power Plan*, has solved 25 problems so far. We consider our preliminary results very encouraging, as they show a great reduction of the problem complexity. For comparison, we could not solve any problem from the standard test-suite by using a non-abstracted representation of Sokoban. Using a partial abstraction (i.e., only tunnels were abstracted), we solved only one problem. We believe that we can further improve our performance in Sokoban in the future. The limitations are on the lack of domain-specific knowledge of Power Plan, not on the abstraction approach.

LAP in Path-Finding

In path-finding, an agent on a map has to find a (shortest) path from his current position to a destination position. The map topology can have many forms, such as a battlefield, the interior of a building, etc. The problem is important in computer games, robot planning, military applications, etc. The classical solving strategy is based on single agent search. In this approach, the map is represented as a grid of atomic cells, and a search algorithm such as A* is used to search for the solution. An action is to move to an adjacent cell, if the destination cell is not an obstacle (Yap 2002).

Our abstraction model groups atomic cells into abstract clusters, reducing the size of the search space dramatically. Actions become moving between two entrances of a cluster (crossing a cluster), rather than moving from one cell to the next. In our experiments, we split the maze into equal-size boxes which become abstract clusters. For example, a 100×100 map can be decomposed into 100 10×10 boxes (clusters). For each edge common to two adjacent clusters, we identify entrances for communication between clusters. An entrance is an obstacle-free part of the edge bounded by two obstacles. For each entrance, we define one *transition point* at the middle of that entrance. No points other than the transition points can be used for moving from one cluster to another.

We can identify a global problem and several local problems, one for each cluster. Processing performed inside a cluster is part of the local problem associated to that cluster. For each pair of transition points on the border of the same cluster, we compute an optimal path between them that is contained in that cluster. Since in path-finding different problem instances use a fixed map but different $(start, target)$ node pairs, this pre-processing phase is performed once and re-used for *many* problem instances. For $n \in \{start, target\}$, we also compute optimal paths from n to the transition points located on the border of the cluster that contains n.

At the global level, we define the *abstract search graph*, whose nodes are $start$, $target$, and the transition points. Optimal paths between the nodes become weighted edges. The abstract graph can easily be updated for different problem instances, as we only have to update information about $start$ and/or $target$. Since the map if fixed, the rest of the abstract graph is fixed too. Searching in the abstract graph is the global problem. In Figure 3 we illustrate this abstraction process on a 20×20 map.

We compared our method with A* performed at the atomic level. Our first results show a great reduction in the search effort. Searching in the abstract graph expands one

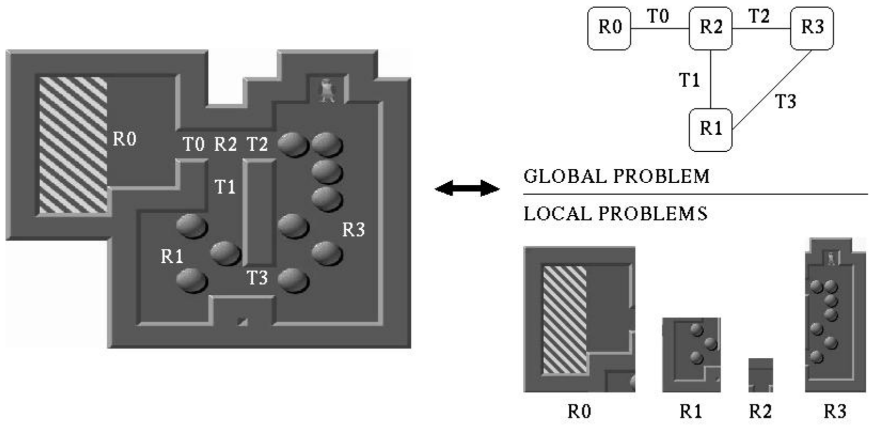

Figure 2: A Sokoban problem (#6 of the test suite) is decomposed into several abstract sub-problems. There is one global problem as well as one local problem for each room. Rooms and tunnels are denoted by R and T, respectively.

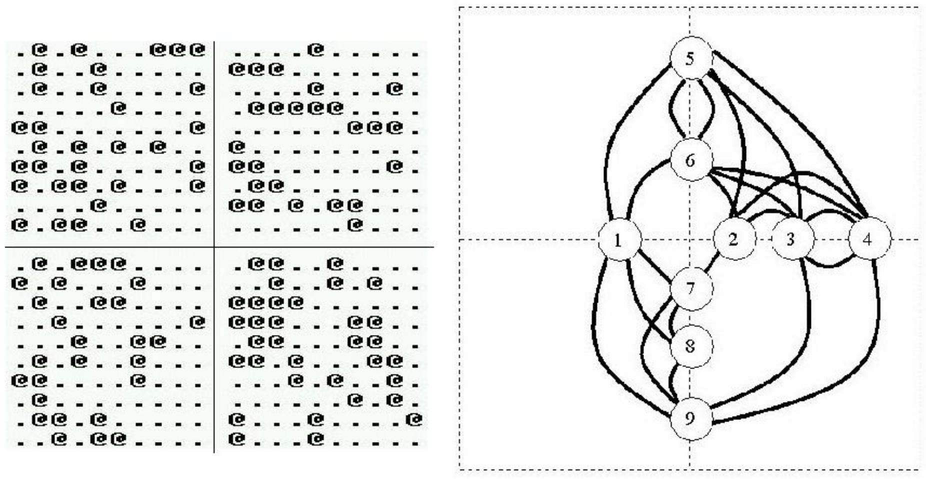

Figure 3: A 20×20 map is decomposed into 4 abstract clusters. The obstacles on the map are represented by '@'. A search space having 280 non-obstacle squares is replaced by a small abstract graph having 9 nodes (not including $start$ and $target$). The abstract search space is shown on the right side of the picture.

order of magnitude less nodes than searching at the atomic level. The length of the solution computed with our method is very close to the optimal value. When parameters such as the maze size or the obstacle rate are varied, the average increase of the solution length (as compared to the optimal value) is consistently under 1%. These results are especially valuable as, in path-finding, the speed of the search algorithm is often crucial, while the solution optimality condition can be relaxed. For instance, in a commercial computer game, most of the CPU cycles are allocated to other game modules such as the graphics engine. In addition, solutions don't have to be optimal. Sub-optimal solutions that look realistic will do, too.

We plan to extend our work in path-finding in many directions, including natural clustering, adaptive abstraction and hybrid map representation. As we show in the next section, when exploring these directions in a generic planning framework, the support of the planning language can be very important. Even if our technique for maze clustering turned out to be quite efficient, we still want to explore how to discover more natural clusters such as rooms inside a building. Natural clusters can lead to simpler local problems. For instance, if the cluster does not contain any obstacle square, then optimal paths between entrances are computed instantly, reducing the pre-processing costs. In a path-finding problem, parts of the map can be unknown for the planning agent. For instance, this is the case when the map is so large that it does not fit into the computer's memory. In this case, the feature clusters can be built dynamically, as the planning agent discovers more and more parts of the map. When the map is partially abstracted, the search can be performed in a space representation containing both abstract local clusters (for the already explored parts of the map) and atomic-size squares (for the unknown parts of the map).

Extending PDDL

In this section we discuss in more detail how a more powerful version of PDDL would be beneficial for using abstraction and hierarchies in AI planning. To better express our vision, we use Sokoban and path-finding as case-study domains.

Since topological abstraction is useful in several domains, we plan to extend our work in the direction of developing a generic planning framework based on this type of abstraction. A significant part of this generic framework can be provided by a more general planning language. As we have pointed out before, the language support that we need mainly refers to hierarchical planning and topological abstraction. The support for hierarchical planning should allow us to express an abstracted problem as a sum of subproblems having different abstraction levels. The support for abstraction includes modeling the relation between low-level and abstract features, automatic abstraction, adaptive abstraction, and using hybrid state representations. The language should also support users to guide the abstraction process by encoding hints that should be considered by the abstraction algorithm.

The rest of the section is structured in two subsections. The first subsection identifies challenges about extending PDDL. The second subsection shows concrete steps toward solving some of the issues pointed out in the first subsection.

Challenges

In this subsection we identify some challenges that we have faced that could be more easily addressed by using a more general planning language. In our experiments with abstract Sokoban (Botea, Müller, & Schaeffer 2002; Botea 2002), we used TLPlan (Bacchus & Kabanza 2000), a generic planner which allows users to plug in domain specific knowledge as C code libraries. Next we show the parts of our framework that had to be implemented as custom code.

Since the planning language did not support hierarchical planning, we represented only the global component as a planning problem (i.e., formulated in PDDL and solved by the generic planner). The solver for the local problems was implemented as custom code in C. This approach definitely has the advantage of efficiency. On the other hand, it clearly points out the need for a unified hierarchical planning framework. In such a framework, users should have the opportunity to describe domains using linked levels of abstraction. As opposed to hierarchical task networks, our abstraction approach defines boundaries between problem components. Users should be supported in expressing topological features of domains such as connectivity of spatial structures within the domain.

All the abstraction levels of a problem (i.e., in our approach, both the global problem and the local problems associated to clusters) should be represented as part of a single PDDL problem formulation. This formulation also encodes relationships between problem components, as well as other useful information that users may consider for explicit representation. The issue of PDDL expressiveness should be kept separated from the strategy adopted for solving planning problems. However, we want to point out that a unified framework may also mean a common solving strategy. In the unified model, we could perform both the high level planning and the low-level computation. The same solving engine can be used to solve both levels of the problem.

The translation of a planning problem from the low-level representation to the hierarchical representation is also part of the planning framework. This means that the abstraction process is also integrated in the model. PDDL could be extended to formalize the codification of rich control knowledge, including hints about how best (in the encoder's view) to decompose a problem into components.

Integrating the abstraction process within the model leads to the interesting and more general debate whether PDDL should be a user-level language or a machine-level language.

In the first scenario (i.e., user-level), the language is used only as an interface through which the planner communicates to the outside world. The planner input (i.e., domain and problem definition) is formulated in PDDL. The planner's output (i.e. a sequence of actions representing the problem solution) can also be considered as a PDDL sequence. All the internal problem representations used by the planner are not part of the generic planning framework. In this scenario, there is a gap in the framework between the low-level representation and the abstracted representation of a problem. Since internal problem representations are hidden and planner specific, we cannot have both a low-level and an abstract PDDL formulation for the same problem. We either start the solving process with an abstracted PDDL formulation, or perform the abstraction internally, being unable to access the internal abstracted problem representation. In our previous experiments in Sokoban we performed the abstraction *a priori*, as a separate pre-processing step. The input of the planner was a PDDL formulation of the global component of the abstracted problem.

The more interesting scenario is to consider PDDL as a machine-level planning language. This allows us to integrate the abstraction process into the framework. Planners can use the language to express internal problem representations at various stages of abstraction. Several possible problem representations, having different abstraction degrees, can be formulated in PDDL and used internally by the planner. This sets a better framework for planners to automatically discover useful abstractions and represent them in PDDL. This also induces greater task modularity and standardization. Abstracted problem representations can be produced and used by other solvers. Last but not the least, when written in PDDL, internal problem representations are easier to understand for humans.

Language Extension Ideas

In this subsection we propose some solutions to the challenges presented above. First, we show how an abstracted problem can be formulated in an unitary framework. Second, we provide concrete ideas about supporting the abstraction process. We include language features that allow users to guide the abstraction.

When expressed in the extended PDDL, the abstract problem formulation should actually be a set of inter-dependent sub-problems defined in the same PDDL file. There is a global problem and several local problems, one for each cluster. The file should also contain inter-relationships occurring across the abstraction levels. To identify the sub-problems, we can introduce two new keywords to the language: `global` and `local`. These keywords are used for the problem formulation, not the domain formulation. The first keyword is part of the global problem header. This problem is described using abstract state features and abstract actions. When we use a hybrid state representation, low-level features can be part of the global state description too. However, to keep our presentation simple, we ignore this possibility for now. The keyword `local` introduces the description of a local problem. For instance, the statement:

```
local: room1
```

can be the header of the local problem associated to the abstract object called `room1` (which was defined inside the global problem). The header should be followed by the PDDL description of the problem. This problem description is at the low level.

Since the global states are described using new abstract features, we need new types of global actions, that change cluster states. The global actions should also be included in the global problem definition. The initial low-level actions become part of the local problems. We point out again that how to define abstract actions should be kept separated from the problem of extending the language. What we consider more relevant is the mechanism for computing action preconditions and effects in the global problem. This mechanism actually establishes the relationship between the local level and the global level of a problem.

The local problems do not interact directly. The only problem interaction allowed in our model is between the global problem and a local problem. At the global level, the clusters are treated as black boxes. When solving the global problem, the planner may need information about the clusters. This information is necessary to check action preconditions (i.e., whether the current state of a cluster allows performing a certain action) and compute action effects (i.e., the resulting internal state of a cluster when a certain action is performed). The cluster information is provided by the local problem associated with the corresponding cluster.

The planner starts solving the global problem. When information about a cluster is necessary, the planner stops solving the global problem and computes the needed piece of knowledge by performing a search in the corresponding local problem. After the information needed at the global level becomes available, the solving of the global problem resumes. There are several ways to optimize this problem solving approach at the local level. First, when local problems are small enough, they can be solved *a priori* (i.e., compute and store all the information about the corresponding cluster that may be needed for the global problem). Second, the results of on-demand local computations can be cached and re-used when needed again. Third, several equivalent cluster states can be merged to compose one abstract state of a cluster.

In Sokoban, we performed the problem abstraction as an application-specific method, with no interference with the generic planning framework. However, we want to develop generic abstraction methods, integrated in our planning model. Since it is often hard to find "good" abstractions by using generic methods only, we consider that language features allowing users to guide the abstraction process are useful.

At one extreme, the user's hints could actually complete the abstraction process. For instance, for each atomic square $square_i$ we can declare:

$$\texttt{(hint (belongs\_to square}_i\ \texttt{room}_j\texttt{))}.$$

This series of statements shows precisely how to build the abstract clusters.

On the other hand, the abstraction process can be automated. The rest of our discussion focuses on this case. The user can assist the abstraction process by encoding hints about how best to decompose a problem into components. A very simple example is the following:

```
(hint (= (max_cluster_size 10))),
```

stating that a cluster should contain at most 10 atomic squares.

Another possible language extension supporting automatic abstraction is the following. Let us assume that the language accepts the declaration:

```
(abstracts room square)
```

as part of the domain formulation. `square` is a predicate instantiated in the low-level problem description, `room` defines an abstract feature, and `abstracts` is a key word of the language. The semantic of this statement is that the domain can be topologically abstracted by building rooms out of (closely related) squares. When a problem is initially loaded to the planning system, no room object is instantiated. Since the system knows that squares can be grouped together to form rooms, a clustering method can be used to discover rooms for the given problem. The clustering algorithm could be either domain-specific or generic. The algorithm can use hints that the user formulates as part of the domain or problem definition. When formulated in the domain definition, the hints apply to all problem instances of that domain. When formulated in the problem definition, the hints apply to the considered problem.

The computed clusters replace the corresponding low-level features in the global problem representation. These low-level features become part of the local problems corresponding to those clusters.

The discussion on hierarchical planning and abstraction applies to both our test cases, Sokoban and path-finding. In addition, path-finding is a good test-bed for adaptive abstraction and hybrid state representation. As in the Sokoban example, the declaration:

```
(abstracts cluster square)
```

can be part of the domain definition in path-finding. Initially, no cluster object is instantiated, since the planning agent did not explore the map at all. On this map, the planning agent is requested to perform many searches, for different start and destination points. For instance, in a commercial game, there can be many characters that have to travel across the map. Moreover, one character can do many trips during one game. As the planning agent performs more and more searches, it also learns more about the map, being able to abstract the already explored parts. In effect, the state representation changes gradually, as more and more low-level squares are replaced by abstract clusters. After building one more abstract cluster, the global problem changes, replacing several low-level squares by an abstract feature. Also, one more local problem, corresponding to navigation within the newly created cluster, is added. Before the abstraction is completed, we need to be able to represent a state as a mixture of both low-level squares and abstract clusters. This means that the global problem accepts both abstract clusters and low-level squares for state representation.

Conclusion

Topological abstraction is a powerful technique for reducing problem complexity in AI planning and single-agent search. The method is based on a clustering of the initial problem representation space. The clustering catches local relationships inside clusters and keeps cluster interactions as limited as possible. In effect, the initial problem is decomposed into a two-level hierarchy of sub-problems, each being much simpler than the initial one. At the local level, each cluster has associated a local problem that solves the local constraints. There is also a global planning problem which uses clusters as features in the global state description.

Since this model is useful in several application domains, it is worth to build a generic planning framework using topological clustering. In such a framework, the expressiveness of the planning language can have a great importance. In this paper we discussed an extension of the PDDL language supporting hierarchical planning and topological abstraction. We pointed out challenges that could be better solved with a more general planning language. We also presented ideas about how to solve these challenges. We demonstrated our ideas using Sokoban and path-finding, two domains where a hierarchical approach based on topological abstraction can be beneficial.

Acknowledgment

This research was supported by NSERC and iCORE. Also, the authors would like to thank the reviewers of this paper. Their feedback truly helped us improve the quality of this paper.

References

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

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

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

Bäckström, C., and Klein, I. 1991. Planning in Polinomial Time: The SAS-PUBS Class. *Computational Intelligence* 7(3):181–197.

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS + Planning. *Computational Intelligence* 11(4):625–655.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using Abstraction for Planning in Sokoban. To appear in *Proceedings of the 3rd International Conference on Computers and Games (CG'2002), Edmonton, Canada*.

Botea, A. 2002. Using Abstraction for Heuristic Search and Planning. In Koenig, S., and Holte, R., eds., *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Artificial Intelligence*, 326–327.

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

Fox, M., and Long, D. 2002. The Third International Planning Competition: Temporal and Metric Planning. In *Preprints of The Sixth International Conference on AI Planning and Scheduling, Toulouse, France*, 115–118.

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

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

Junghanns, A. 1999. *Pushing the Limits: New Developments in Single-Agent Search*. Ph.D. Dissertation, University of Alberta.

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

Long, D.; Fox, M.; and Hamdi, M. 2002. Reformulation in Planning. In Koenig, S., and Holte, R., eds., *Proceedings of the 5th International Symposium on Abstraction, Reformulation, and Approximation*, volume 2371 of *Lecture Notes in Artificial Intelligence*, 18–32.

McDermott, D. 1997. Using Regression-Match Graphs to Control Search in Planning. `http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/mcdermott.html`.

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

Precup, D.; Sutton, R.; and Singh, S. 1997. Planning with Closed-loop Macro Actions. In Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems, 1997.

Yap, P. 2002. Grid-based path-finding. In Cohen, R., and Spencer, B., eds., *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence*, 44–55.