Plan Fragment Libraries

Roman van der Krogt André Bos Cees Witteveen

Delft University of Technology
{R.P.J.vanderKrogt,A.Bos,C.Witteveen}@ITS.TUDelft.NL

Abstract

One of the important aspects of agents is their ability to construct plans to reach their goals. This ability allows them to operate autonomously, without the intervention of a (human) operator. Unfortunately, the planning process is a very complex task. One way to reduce this complexity is to use case-based planning, which stores past plans and adapts these past plans to new situations. We do not want to store complete plans, but smaller, frequently occurring combinations of actions that solve only a few number of goals. We will call such a combination a *plan fragment*. These plan fragments can then be combined to form a plan.

These plan fragments are stored in and retrieved from a library which is accessible to the agent. This paper focuses on such plan fragment libraries. We will show a simple algorithm that uses such a library to create plans. We also investigate a way to order the plan fragments according to some preference relation. The initial algorithm is extended to one that can use such an ordered library in an efficient way, making use of *tabus*.

1 Introduction

Agents often have to achieve a number of given goals without a predefined way to accomplish them. Therefore, they have to make a plan that consists of a number of actions that, starting with the current situation, brings the agents to a state in which they have reached their goals. In general, these *planning* tasks are very hard, in fact, they are PSPACE-complete when a propositional planning language (such as STRIPS [3]) is used [7] and undecidable for less restricted languages [2]. One of the things that have been suggested is to use *case-based planning* [5, 6]. A case-based planner stores past plans in its memory, and uses this past experience to guide its search. With each plan is associated a list of obtained goals and possible failures that the agent has experienced. When an agent gets a set of goals to satisfy, it searches its memory for a plan that obtains similar goals. This plan is then adapted, using certain plan adaptation rules, to a plan that satisfies the current conditions. The memories of past failures are used by the planner to prevent them from happening again.

Whereas a case-based planner selects one plan and adapts it, we propose to store smaller (sub)plans in memory that can be used to satisfy a smaller number of goals. For example, the sequence of actions $load(truck, cargo)$, $move(truck, here, there)$ and $unload(truck, cargo)$ can be used to satisfy the goal of bringing a load to another location. These *plan fragments*, as we call them, are then combined to

form a plan for all goals. In order to efficiently combine them, it may be necessary to first remove some actions from the plan fragments. This may be because the plan fragment has a common part with the existing plan, or because the plan fragment contains actions for goals that are not needed.

The plan fragments are stored in a library, which is simply a list of plan fragments. This plan fragment library is consulted for possible plan fragments during the search for a plan, and as such plays a central role in the planning process. We will use plan fragment libraries to solve a specific kind of planning problem: Given plan fragments pf_i that can be used to satisfy goals G_i, how can we construct a plan that satisfies goals $G \subseteq \bigcup G_i$. The paper will focus on this problem, and on how we can order the plan fragments according to some preference relation. This ordering accomplishes two things: (i) it enables the planner to distinguish between different plan fragments, other than by their preconditions and effects, which can be used to steer the search process and (ii) by searching a small set of preferred plan fragments first, and gradually expanding this set when no solution is found, we can find plans that use only preferred plan fragments much quicker, as we don't have to search all fragments with a lower preference.

The remainder of this paper is organized as follows: First, we describe the notions of plan fragments and a plan fragment library in more detail. Also, we describe a simple search algorithm that uses a plan fragment library to solve (re)planning problems. Then, in section 3 we will investigate ordered libraries. Section 4 will describe future work after which we will conclude.

2 Plan Fragments

Normally, a planning or replanning problem is specified by giving the initial and goal states and a list of actions that can be taken. The (re)planning algorithm then searches for a combination of actions that will bring the agent from the initial state to the goal state. In most search algorithms, knowledge about actions that often occur together is not taken into account. If the search algorithm would know about this, this could reduce the search time. A *plan fragment* is such a combination of actions that can occur together.

As we will make use of our ARPF formalism throughout the rest of this paper, we will briefly describe it here. For a more extensive description, see for example [8]. Basic notions in our formalism are *resources* and *actions*. Actions are basic production processes of the form $\langle R_1, C \rangle \to R_2$, that consume a set R_1 of one or more input resources and produce a set R_2 of one or more output resources. Each action can have *constraints* C on its input resources, that describe in what situations the action can be used. The input resources of a skill s are denoted by $in(s)$, output resources by $out(s)$. A *goal* is a resource that should be produced. A *plan* is a partially ordered set of actions, that consumes a number of input resources, and produces a number of output resources. Some of these will be used to satisfy *goal resources*. Thus, a set of goals G is satisfied if the plan produces a set of resource $R \supseteq G$ that contains the goals G. The resources that are produced but not used for goals are called *free resources*. A plan fragment is a list of actions together with some constraints on the use of the plan fragment. The set of input resources of the plan fragment pf is denoted with $in(pf)$, and $out(pf)$ denotes the

resources that are produced by pf.

The plan fragments are stored in a *plan fragment library*. Such a library contains a list of plan fragments with the goals they attain. Plan fragments can be given in the domain description, or learned by examining past and current plans for frequently occuring combinations.

2.1 Searching with Plan Fragments

As said, the agent has a library of plan fragments that it has available. How can the agents use this knowledge to create a plan? Suppose the agent has plan fragments $pf_i, \ldots pf_n$ that consume resource sets $R_1, \ldots R_n$ to satisfy goal sets $G_1, \ldots G_n$. Furthermore assume that these goals G_i form the set of all possible goals $\mathcal{G}$ for that agent: $\bigcup G_i = \mathcal{G}$. We require that for each goal $g \in \mathcal{G}$ there is at least one plan fragment for which $g \in out(pf)$. The problem is to find a combination of plan fragments that together form a plan for the current goals $G \subseteq \mathcal{G}$ of the agent using a set of available resources $avail$.

To add plan fragments to a plan, we introduce the $\oplus$ operator. This operator takes a plan P and a fragment pf and returns a plan $P' = P \oplus pf$, which is the combination of P with pf. Adding a plan fragment pf to a plan P involves finding out which part of pf is already available in the plan, and then add the remainder of pf to P. This is done in such a way that the number of new input resources is minimized. The use of this operator is illustrated in Figure 1 (for a formal description of $\oplus$, see [8]). A simple backtracking algorithm using $\oplus$ to find such a combination is shown in Algorithm 1. Given a partial plan P, goals G, available resources $avail$ and a plan fragment library Pf, it tries to add each possible plan fragment pf on the partial plan P. This will change the available resources, resulting in a new set I: The resources that pf consumes are no longer available, while pf produces free resources that are now available for other plan fragments. The set of goals will change too: Some goals are satisfied by output resources of pf, whereas some input resources of pf are not available and have to

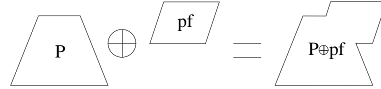

Figure 1: Example use of the $\oplus$-operator.

Algorithm 1 A simple search algorithm

SEARCH(plan P, goals G, available resources $avail$, library Pf)

1. **if** $G = \emptyset$ **then**
   1.1. **return** P
2. **for** *all* $pf \in Pf$ *for which* $out(pf) \cap G \neq \emptyset$ **do**
   2.1. **let** $O = (G \setminus out(pf)) \cup (in(pf) \setminus avail)$
   2.2. **let** $I = (avail \setminus in(pf)) \cup (out(pf) \setminus G)$
   2.3. **if** SEARCH($P \oplus pf, O, I, Pf$) *returns a plan* P' **then**
      2.3.1. **return** P'
3. **return** *failure*

be provided too. These new goals will be called O. Then it tries to search further, until the set of goals to be satisfied is empty. If all fragments have been unsuccessfully tried, the algorithm backtracks, and tries to find another combination. Note that the final plan may produce a number of resources that are not used to satisfy goals. This means that the plan may have to be examined for superfluous actions. Then, the agent may store the plan as a plan fragment in its library.

The problem with this simple algorithm is, obviously, the time it requires to find a solution. Trying all possible combinations until a right one is found, is not a very smart way to search. A possible improvement is to *order* the plan fragments, as we will show in the next section.

3 Ordered Libraries

An important aspect of a plan fragment library is its size: It should be big enough to make sure that all problems can be solved, yet at the same time it should be small to reduce the search time. To establish this trade-off, we propose to use an *ordered* library. The idea is to order the fragments according to preference.[1] Then, we first search using only the most preferred fragments, gradually adding more (less preferred) fragments when the goals cannot be satisfied. For example, in the transportation domain, highways could have the highest preference, followed by provincial roads and others. Thus, ordering the plan fragments allows to distinguish different plan fragments that satisfy the same set of goals, but in a different way. If the preferred plan fragments do not fit, the library may contain redundant fragments with a lower preference that could be used.

Algorithm 2 A simple search strategy using an ordered library.

SIMPLE-ORDERED-SEARCH(Plan P, goals G, available resources *avail*)

1. *preference:=highest*
2. **while** *solution not found* **do**
   - 2.1. **let** $Pf = \{pf | preference(pf) > preference\}$
   - 2.2. SEARCH$(P, G, avail, Pf)$
   - 2.3. *decrease preference*

A simple search algorithm using ordered libraries could look like the one that is given in Algorithm 2. This algorithm simply searches a growing set of plan fragments until it finds a solution. Note that the SEARCH-procedure hasn't changed, it is just called repeatedly, with a growing library. There is one main drawback to this algorithm: Because the set of plan fragments is growing, the planner examines the plan fragments of higher preference again in subsequent searches with lower preference. This can be solved by replacing the SEARCH-procedure by the TABU-SEARCH procedure of Algorithm 3. It uses a memory to store partial plan solutions and *tabus* [4] as follows: The algorithm works the same as SEARCH, i.e.

[1] We assume the preference order over the set of plan fragments to be a weak order; equivalence classes are denoted by integers in the usual way; the preference of a plan fragment then equals the preference number of its equivalence class.

Algorithm 3 A search strategy incorporating tabus.

TABU-SEARCH(plan P, goals G, available resources $avail$, library Pf)

1. **if** $G = \emptyset$ **then**
  1.1. **return** P
2. **for** *all* $pf \in Pf$ *for which* $out(pf) \cap G \neq \emptyset$ **do**
  2.1. **if** $\exists \langle P, I, O\rangle \in \text{memory} \cdot in(pf) \supseteq I \wedge out(pf) \subseteq O$ **then**
    2.1.1. *skip this plan fragment*
  2.2. **else**
    2.2.1. **let** $O = (G \setminus out(pf)) \cup (in(pf) \setminus avail)$
    2.2.2. **let** $I = (avail \setminus in(pf)) \cup (out(pf) \setminus G)$
    2.2.3. **store-tabu** $(\langle P, in(pf), out(pf)\rangle)$
    2.2.4. **if** TABU-SEARCH$(P \oplus pf, O, I, Pf)$ *returns a plan* P' **then** **return** P'
3. **return** *failure*

it takes an (empty) partial plan to start from and refines it through a sequence of partial plans to the final plan. The differences are in lines 2.1 and 2.2.3. In line 2.2.3 each partial plan P that is considered is stored in memory, together with the inputs and outputs of the plan fragment pf that was used to refine it. This tuple $\langle P, in(pf), out(pf)\rangle$ is called the *tabu*. The tabu will be used to prevent examining duplicate situations as follows: When a partial plan P is considered, the memory is first consulted for tabus $\langle P, I_x, O_x\rangle$ for this plan. If such tabus are in memory (line 2.1), then certain plan fragments are prevented from being considered. Plan fragments pf' for which $in(pf') \supseteq I_i$ and $out(pf') \subseteq O_i$ (for some i) are using the same or more resources to produce the same or fewer output resources. By disregarding these plan fragments, only different or "more powerful" plan fragments are taken into consideration. For example, suppose that we have just started the search, and refine the empty plan $\emptyset$ with a plan fragment $A, B \rightarrow C$.[2] Now we add $\langle \emptyset, \{A, B\}, \{C\}\rangle$ to the memory. Next time that we examine the empty plan for refinement, we have that:

- $A, B, D \rightarrow C$ need not be considered, since $\{A, B, D\}$ is a superset of $\{A, B\}$, the inputs of the tabu. Considering this plan fragment would not lead to new situations, as any situation in which $\{A, B, D\}$ can be consumed, so can be $\{A, B\}$. Thus, we can safely ignore this fragment.
- $A, B \rightarrow C, D$ needs to be considered, for $\{C, D\}$ is not a subset of $\{C\}$. This fragment uses the same resources, but produces an extra resource D. This is a new situation that is to be explored, since the D resource could be vital to finding a plan.
- $E, F \rightarrow G, H$ should also be considered, because neither $\{E, F\} \supseteq \{A, B\}$

[2] "$A, B \rightarrow C$" means: A and B are inputs to a plan fragment that produces C.

nor $\{G, H\} \subseteq \{C\}$. This clearly is a new situation that should be investigated, as none of the consumed or produced resources was used before.

This way, we ensure that no partial plans are tested that have already been examined. The algorithm sketched here is one of the algorithms that we use for replanning purposes. It can be used to satisfy additional goals to an existing plan.

Besides taking the changing world into account in our (re)planning algorithms, we assume that the ordering is not static, as there are events in the world that may change the preferences of certain plan fragments. For example, a monitoring agent can alert the other agents that two out of three lanes of a highway are closed due to road works, likely causing a traffic jam at that point. As a result, the plan fragment gets a lower precedence and is moved down in the hierarchy. If the times for such an obstruction are known in advance, it is even possible to replace the fragment by two other fragments: One with a normal precedence and a constraint that forbids the use of this plan fragment during the period of road works, and another fragment with a lower precedence and a constraint which only allows it to be used during the obstruction. This way, the plan fragment library can be used to anticipate possible failures.

Besides reacting to external events, there is another reason to change preferences of plan fragments. If an agent frequently encounters a problem for which it uses a fragment, this fragment should get a higher preference, in order to speed up the search next time. By changing the preferences, the agent can learn from its planning. Also, the library can be ordered according to the problem at hand: For example, plan fragments that are able to satisfy some of the current goals, or use available resources can be given higher priority. This gives higher preferences to plan fragments that use available resources to produce goals.

Finally, if the agent creates a new fragment, it has to give this a new preference. The things the agent has to account for when ordering the new fragment, are:

- The (number of) goals the plan fragment satisfies. The new fragment should be ordered relatively of fragments that are able to satisfy roughly the same fragments. Plan fragments that are able to satisfy a lot of resources should have a higher priority. This way, the search procedure can satisfy a lot of resources at once.

- The (number of) resources that the plan fragment requires. In general, the less resources that the fragment uses, the higher its preference. Furthermore, resources that are available, or for which there are a lot of possibilities (plan fragments) to produce it are preferred.

- The number of other fragments that are able to produce the same resources. Fragments that produce very scarce resources (i.e. few other plan fragments are able to produce it) should have a higher priority, thus assuring that the search procedure can find a way to produce the resource quickly.

Furthermore, it should be noticed that the preference of existing plan fragments changes as a consequence of the addition of new ones.

4 Future Work

There are two areas into which we like to expand our current work: Besides extending our work on replanning (of which the work described here is part) we would also like to investigate using plan fragment libraries in a multi-agent environment.

4.1 Replanning

The work presented here is a part of broader work in the area of replanning. Our work on replanning focuses on three plan operators: (*i*) the removal of plan fragments from a plan, in case goals are no longer needed, (*ii*) the creation of plans by combining plan fragments as discussed here, and (*iii*) the replacement of plan fragments in a plan with others. A single algorithm has been found, which is able to adapt a plan to arbitrary changes in the set of goals using plan fragments and tabus. This algorithm has to be further developed to allow it to be used on a wider variety of problems. Also, we are currently working on an implementation with which we can test our ideas.

4.2 Multi-agent environments

If there is more than one agent, the agents sometimes need to cooperate in order to solve problems. For example, if a transportation agent that only has trucks available gets an order to transport something overseas, it will need the help of an agent who can use boats or airplanes. In this case, shared libraries can be used. This section will briefly discuss two ways in which agents can share libraries.

One way to do this, would be to have a single, shared library in which agents can "advertise" their services. Agents can add the services they can provide to the library, and retract services that they can no longer provide.[3] Now, when an agent cannot find a suitable plan fragment for its problem, it can consult the shared library to see if one of the other agents has an applicable fragment. The advantage of this scheme is its simplicity: There is a single point which keeps track of the available plan fragments. This single point is also the biggest drawback: It provides a single point of failure, and it is not scalable. Furthermore, agents cannot decide beforehand which agent they want to allow to use their services. For example, different branches of the same organization would like to share their fragments with each other, but not with other companies.

Another way in which agents can share their plan fragments, is by directly including them in each other's libraries. This scheme has the drawback of requiring more communications, since it needs to inform all the other agents in the group of changes. There is, however, also a big advantage: An agent has to search only one library, and does not have to switch between libraries. When using an ordered library, the agent can decide for himself at what level of preference the foreign plan fragments should have, thus specifying at what moment to start considering the plan fragments of others.

Which method of communication is preferable is not obvious. One or more shared libraries offer a method with low communication costs, whereas writing

[3] Agents that do not want the details of their plan fragments revealed, may choose to only present the input and output resources of their fragments.

directly in the library of other agents eases the search process. Experiments will have to show which of the methods is to be preferred in certain conditions.

5 Conclusions

This paper investigated an idea similar to that of case-based planning. Whereas case-based planning searches its memory for one plan to adapt, our method retrieves several plan fragments from memory, and combines those into a single plan. We provided a simple algorithm that uses this method to create plans for goals it has already solved before. Then we described the concept of an ordered library of plan fragments, in which preferred plan fragments are considered first. We extended our algorithm to one that uses tabus to efficiently search such a hierarchy of plan fragments. Here, the tabus are used in order to prevent that equivalent plan fragments of different levels in the hierarchy are considered twice. We showed that in dynamic environments the ordering in this hierarchy should also be dynamic. The preference of plan fragments changes due to events in the world, but also because of the experiences of the agent with planning. The agent may find that some fragments are hardly used, in which case they should get a lower preference, whereas other are used frequently, justifying a higher preference. The preferences also change when the agent learns new plan fragments. This is also the main difference between our paper and existing work using plan fragments (such as [1]), where the library of plan fragments is static and not ordered.

We think this is an interesting variation to case-based planning. In the future, we hope to further develop this method in our work on replanning, and by investigating possible use in multi-agent environments.

References

[1] S. Bhansali, G.A. Kramer, and T.J. Hoar. A principled approach towards symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, 1996.

[2] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[3] 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.

[4] F. Glover and M. Laguna. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*. Scientific Publications, Oxford, 1993.

[5] K. J. Hammond. Case-based planning: A framework for planning from experience. *Cognitive Science*, 14:385–443, 1990.

[6] S. Hanks and D.S. Weld. A domain-independent algorithm for plan adaptation. *Journal of AI Research*, 2:319–360, 1995.

[7] B. Nebel. the compilability and expressive power of propositional planning formalisms. Technical Report 101, Albert-Ludwigs-University Freiburg, 1998.

[8] R.P.J. Van der Krogt, A. Bos, and C. Witteveen. Replanning in a resource-based framework. To appear in *Post-course proceedings of 9th ECCAI Advanced Course on AI (LNCS/LNAI series)*. Springer Verlag, 2001.