Planning with Arbitrary Monotonic Resource Functions

Tatiana Kichkaylo
New York University
719 Broadway, room 706
New York, NY 10003, USA
kichkay@cs.nyu.edu

Abstract

Many real-world planning problems involve resource conditions and effects expressed by complex functions. In this paper, we present an algorithm for the component placement problem, which is a real-world problem with a simple logical structure and arbitrary monotonic resource functions. We discuss challenges arising in solving this problem, possible approaches to addressing them, and ways of extending our algorithm to more general classes of planning problems.

Introduction

Many practical planning problems involve real valued resource variables. A number of algorithms have been developed for such problems (e.g. Zeno (Penberthy & Weld), RIPP (Koehler 1998), LPSAT (Wolfman & Weld 2000)). However, such algorithms usually restrict the kinds of expressions that can be used in resource preconditions and effects to simple linear functions (Zeno supports more general expressions, but postpones their processing until they are linearized due to binding of some variables). Such restrictions allow the planners to use fast specialized methods, such as the Simplex method, or reverse resource functions, i.e. compute values of parameters of a function given a result. In real world problems, it is not always reasonable to assume that resource functions satisfy these restrictions.

The component placement problem (CPP) arises in distributed component based frameworks. The goal in this problem is to choose a set of components, their locations, and connections between them in a resource-constrained network environment (see the next section for details). Since this problem involves choosing a set of components, it seems reasonable to apply AI planning to it.

Compilation of the CPP into a planning problem has a well-defined logical structure and no negative preconditions and effects. However, two features of this problem preclude use of existing algorithms.

First, although the size of the answer is usually small, the problem specification may be very large. Static preprocessing techniques are not effective in case of the CPP, and new techniques need to be developed.

Second, the resource functions in operator preconditions and effects are not necessarily linear and often non-reversible. For example, if a component merges two data streams to produce one, it may be impossible to calculate the bandwidth of each of the incoming streams given the bandwidth of the resulting stream.

The rest of this paper is structured as follows. First, we describe the component placement problem. Second, we present the Sekitei algorithm for the CPP. To be useful, such an algorithm needs to produce a solution within seconds. Currently, Sekitei combines regression and progression techniques to dynamically prune the set of operators. We are also exploring several approaches to dealing with arbitrary non-reversible monotonic resource functions. We conclude with a discussion of future work.

The Component Placement Problem

Component-based frameworks (Ivan *et al.* ; Fu *et al.* 2001; S. Gribble et al. 2001) allow distributed applications to be constructed from individual components. Dynamic selection and placement of components allows for adaptation to changing characteristics of the environment and user preferences, but requires solving the component placement problem. In this paper, we consider a special case of such applications, where components consume and produce data streams, and the data streams are sent over links between nodes in a wide-area distributed system.

For example, consider the following scenario (Figure 1). The server provides a combined media stream consisting of images and text. The client issues requests at a particular rate, which translates into the minimum bandwidth requirement. If the network between the client and the server has high bandwidth, a direct connection is made. However, in more resource-restricted situations additional components might be injected into the network: Figure 1 shows an example of such injection involving splitter, merger, and compression components. The CPP attempts to place these components automatically by viewing components as operating on typed data streams.

In general, the specification of the component placement problem consists of:

- a description of the network (a set of nodes, a set of links, properties of the nodes and links such as available CPU, link bandwidth, link security),

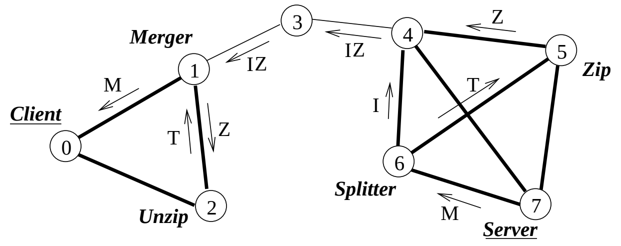

Figure 1: The webcast application. The network consists of two high-bandwidth LANs with a low bandwidth link between them. The Server located on node 7 produces a media stream, and the Client on node 0 wants the media stream with high bandwidth. This goal is achieved by splitting the media stream (M) into text (T) and image (I), zipping the text portion of the stream, so that the combined I+Z bandwidth is less than that of the original M stream, sending the I and Z streams to the client LAN, and performing the reverse transformations there.

- a set of available component types specified by required and produced types of data streams, resource requirements of the component as a function of parameters of the incoming streams and properties of the node where the component is to be deployed, and functions describing parameters of the produced data streams, and
- a user goal as a pair of a node and a component type to be deployed on that node.

The goal is to find a (smallest) set of components and connections between them such that all requirements of all components are satisfied and the goal is achieved. For more details on the specification of the CPP see (Kichkaylo, Ivan, & Karamcheti 2003).

This problem can be compiled into a planning problem as follows.

- Availability of a data stream on a node and the deployment of a component on a node are described by propositions. For example, `avText(0)` means that a Text stream is available on node 0, and `placedZip(2)` means that a Zip component is deployed on node 2.
- Properties of links, nodes, and data streams on nodes are resource variables. For example, `cpu(0)` describes the available CPU on node 0, `ibwMedia(1)` is the bandwidth of the Media stream on node 1, and `lbw(1,2)` is bandwidth of the link between nodes 1 and 2.
- There are two types of operators: Placing a component on a node and crossing a link by a data stream. Operators have logical and resource preconditions and effects. For example, the operator `placeSplitter` (Figure 2) describes placing of a Splitter component on a node. Logically, the Splitter requires that a Media data stream be present on the node, and as a result the Splitter makes Text and Image data streams available on that node. From the resource point of view, placing the Splitter is possible if sufficient number of CPU units is available to process the incoming Media stream. The resource effect formulae describe the change in CPU availability and bandwidth of the produced streams. Similarly, the operator `crMedia` describes sending a Media stream from node `n1` to node `n2`. There are no resource preconditions, and resource effects in this case describe consumption of the link bandwidth and the bandwidth of the data stream at the destination.

```
operator placeSplitter(?n: Node)
Lreq: avMedia(?n)
Leff: avText(?n),
      avImage(?n),
      placedSplitter(?n)
Rreq: cpu(?n)>max(ibwMedia(?n), 10)
Reff: cpu(?n)-=max(ibwMedia(?n), 10)
      ibwText(?n):=ibwMedia(?n)*0.3
      ibwImage(?n):=
                min(ibwMedia(?n)*0.7,
                    sqrt(cpu(?n))*10)

operator crMedia(?n1, ?n2: Node)
Lreq: avMedia(?n1)
Leff: avMedia(?n2)
Rreq:
Reff: ibwMedia(?n2):=min(ibwMedia(?n1),
                         lbw(?n1,?n2))
      lbw(?n1,?n2)-=min(ibwMedia(?n1),
                         lbw(?n1,?n2))
```

Figure 2: Examples of operators in the compilation of a CPP into a planning problem.

The planning problem obtained from a CPP has a simple logical structure with no negative logical preconditions or effects (if a data stream reaches a node it does not get destroyed). What makes this problem hard is that the resource expressions may involve arbitrary non-reversible functions. The only assumption we make is that all functions are monotonic. For example, if bandwidth of a data stream at the source increases, the bandwidth at the destination will not decrease, and if a component can be deployed on a node with less resources, it still can be deployed on that node if more resources become available. These assumptions are true for the applications we are addressing.

Limiting the Search Space

The Sekitei algorithm

The first issue that an efficient planner for the CPP needs to address is the size of the problem. A problem instance can include hundreds of nodes and dozens of component types, which translate into component placement and link crossing operators. However, most of these operators will not be used in the shortest plan that achieves the goal. Standard preprocessing techniques (Blum & Furst 1997) do not remove these operators, because they can be included in some (long) sequence leading from the initial to the goal state. For example, when sending a data stream between two nodes in the same LAN, the operator for crossing a network link on the other side of the globe cannot be statically eliminated.

Sekitei (Kichkaylo, Ivan, & Karamcheti 2003) limits the search space by combining regression and progression approaches and adding resource checks *in layers*, so that resource functions are evaluated only for *promising* operators. Sekitei performs regression (backward search) and progression in the network structure similar to classic planners reasoning about time. The high-level algorithm is as follows:

1. Build a regression graph RG for the goal using only logical preconditions and effects of operators (see example below). Let N be the minimum depth of RG reaching the initial state.
2. Build a planning (progression) graph PG (Blum & Furst 1997) for N steps, using only operator and propositions belonging to the corresponding layers of the subgraph of RG rooted in the initial state. For each propositional layer of PG, compute an optimistic resource map as described below. When building an operator layer, execute each of the operators in the optimistic map of the preceding propositional layer.
3. If the last layer of PG contains the goal, search for a plan in PG using the procedure described in (Blum & Furst 1997). Symbolically execute the found plan to ensure soundness. If execution succeeds, return the plan.
4. Add one step to the RG. Set N=N+1. Go to step 2.

Execution of an operator changes values of resource variables as described by the operator's resource effects. Let $V = \{v_1, ..., v_n\}$ be the set of all resource variables. A **state** is described by a set of name-value pairs for all variables:

$$S = \{(v_1, c_1), ..., (v_n, c_n)\}, \text{where } \forall i\ c_i \in \mathbb{R}$$

Execution of an operator *op* in a state produces a new state where values of some variables are changed:

$$exec(op, S) = S'$$

A **resource map** is a mapping of each variable in V to a minimum and maximum value.

An **optimistic resource map** $lmap(l)$ for a given layer l of the planning graph is defined recursively as follows. $lmap(0)$ maps each variable into its minimum and maximum value in the initial state. For $l > 0$, $lmap(l)$ maps resource v to the minimum and maximum value of v over all states that result from applying any operator of layer l of the progression graph to any state consistent with $lmap(l-1)$.

Since all resource functions are monotonic, it is sufficient to construct states using only boundaries of the intervals. Let $single(map)$ be a set of all such states for the map:

$$single(map) = \{S_j\}$$

where

$$map = \{(v_1, cm_1, cM_1), ..., (v_n, cm_n, cM_n)\}$$
$$S_j = \{(v_1, c_1), ..., (v_n, c_n)\}, \forall i\ c_i \in \{cm_i, cM_i\}$$

Now the optimistic resource map can be computed as follows.

1. $lmap(0) = \{(v_i, cm_i, cM_i) | v_i \in V\}$, where cm_i and cM_i are minimum and maximum values for resource v_i in the initial state.

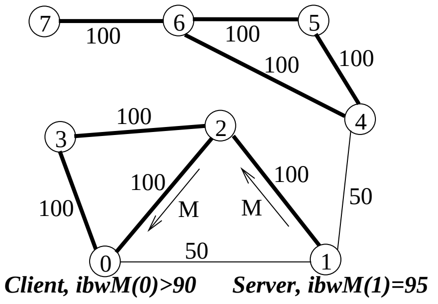

Figure 3: In the network of 8 nodes connected with links of different bandwidth, it is required to send a data stream from node 1 to node 0 so that the bandwidth of the stream at the destination is at least 90. The shortest plan to achieve this goal involves sending the stream through node 2, because the direct link between nodes 0 and 1 does not provide sufficient bandwidth.

2. Let $ops(l)$ be the set of operators, including no-ops, of layer $l > 0$ of the planning graph. Then

$$\begin{aligned} &lmap(l) = \{(v_i, cm_i, cM_i)| \\ &\quad cm_i = min\ c, cM_i = max\ c, \\ &\quad (v_i, c) \in exec(op, S), op \in ops(l), \\ &\quad S \in single(lmap(l-1))\} \end{aligned}$$

Sekitei relies only on monotonicity of resource functions. We do not assume that the functions are reversible (i.e. that we can compute the arguments given the result). Therefore, we cannot propagate goal intervals backwards during the plan extraction phase (step 3) as done in (Koehler 1998). Symbolic execution must be performed after the plan extraction to ensure soundness.

Example

To illustrate how the regression-progression approach helps to limit the search, consider the example shown on Figure 3. The goal is to place the Client on node 0, which requires sending a data stream M produced by the Server running on node 1 to node 0 so that the bandwidth of the stream on the client node is at least 90. It is easy to see that, although there is a direct link between nodes 0 and 1, the shortest plan to achieve the goal involves crossing two links.

Sekitei starts by building a layered regression graph starting with the goal state (Figure 4). Only the logical part of operator specifications is taken into account, so that the smallest subgraph rooted in the initial state corresponds to the single-link plan (shown in bold).

Given the operators and propositions of the subgraph found at the previous step, Sekitei builds a planning graph. Execution of the `placeClient(0)` operator in the optimistic resource map containing `(ibwM(0),50,50)` fails, and Sekitei adds one more layer to the regression graph. The new subgraph of RG (shown in thin solid lines) contains the two-link plan. The corresponding progression graph (Figure 5) reaches the goal, and contains the solution `(crM(1,2), crM(2,0), placeClient(0))`,

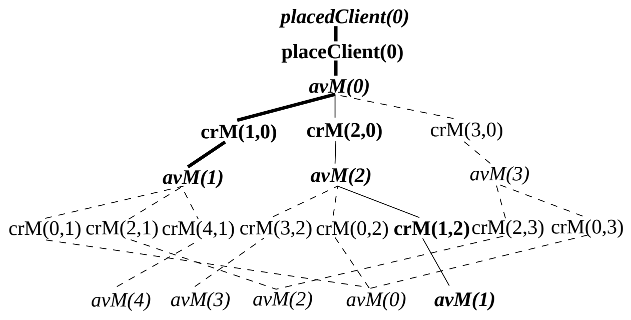

Figure 4: The regression graph for the problem shown on Figure 3. Bold links correspond to the smallest subgraph of RG rooted in the initial state, solid lines correspond to the final subgraph. Propositions are shown in italics, operators in normal font. Propositions and operators used for construction of the progression graph are shown in bold.

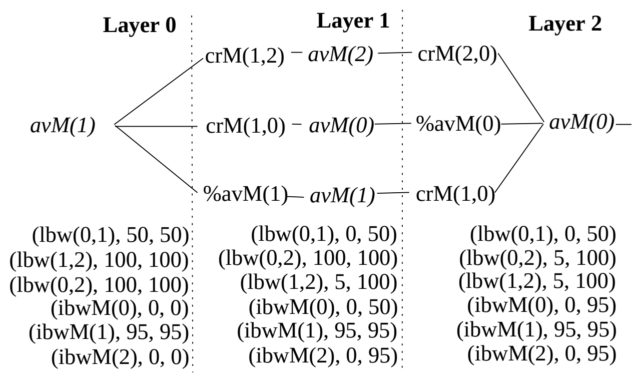

Figure 5: The progression graph containing the solution. The optimistic resource maps for each step are shown below the graph. The third layer `placeClient(0) - placedClient(0)` is not shown. `%avM(n)` is a no-op operator for proposition `avM(n)`.

i.e. cross links from node 1 to 2 and from 2 to 0 with interface M, and place the client on node 0.

Note, that because of the layered structure of the algorithm, only the portion of the network around nodes 0 and 1 was visited, and the resource functions were evaluated only for three links.

Evaluation

We implemented Sekitei in Java and tested it on a variety of component placement problems. The regression-progression approach does indeed help in pruning the set of operators, and the algorithm scales well with respect to the size of the network and the number of component types not used in the final plan (see (Kichkaylo, Ivan, & Karamcheti 2003) for detailed results).

However, the performance of the planner degrades quickly in scenarios where steps are added to the plan solely because of resource restrictions. For example, if Figure 1 is part of a larger graph, the planner may consider placing useless Zip and Unzip components off the main path.

This problem stems from the fact that symbolic execution is performed after plan extraction. This means that many resource conflicts are detected very late, which leads to poor performance on problems involving injecting multiple components and using multiple data streams. If the operator that fails during the symbolic execution is close to the end of the plan, then the same plan prefixes are evaluated many times. For example, in the webcast example (Figure 1) all plan prefixes succeed up to the placement of the client.

Reducing the Number of Function Evaluations

One solution to the above problem is to save intermediate results. GraphPlan-based algorithms use a technique called memoization: for each of the layers of the planning graph sets of propositions not achievable together are memoized, so that they do not get checked more than once. Similar to this, we use **positive memoization** to save good sets of propositions along with corresponding resource maps.

Positive memoization

The high-level goal of positive memoization is to detect resource conflicts earlier during the plan extraction phase by executing plan tails in the optimistic resource maps. In the Sekitei algorithm described above, the maps are built per layer. To make resource conflict detection more effective, we need to calculate resource maps at finer granularity.

Similar to the optimistic resource map for the whole layer, we define an optimistic resource map $smap(q, l)$ for a subset of propositions q at layer l of a planning graph.

1. $smap(q, 0) = lmap(0)$ for all q.
2. Let $ops(q, l)$ be a set of smallest subsets of operators, including no-ops, at layer l that together achieve q.

   Let $precs(o, l)$ be a set of preconditions (propositions at level $l - 1$) of the set of operators o at level l.

   Then the optimistic resource map $smap(q, l)$ for $l > 0$ is defined as follows:
   $$\begin{aligned} &smap(q, l) = \{(v_i, cm_i, cM_i) | \\ &\quad cm_i = min\ c, cM_i = max\ c, \\ &\quad (v_i, c) \in exec(op, S), op \in O, O \in ops(q, l), \\ &\quad S \in single(smap(precs(O, l), l - 1))\} \end{aligned}$$

   In words, each subset of operators achieving q is executed in the optimistic resource map for the union of preconditions of these operators, and then the map for q is computed as a union of the resulting maps.

After the optimistic map is computed for the goal state, the plan extraction phase proceeds as usual, except every time a subset of operators o is chosen at some level l, the plan tail including o is replayed in the optimistic map of o's preconditions $smap(precs(o, l), l - 1)$.

Note that the use of positive memoization does not put any additional restrictions on the form of resource functions; only monotonicity is required.

Adding positive memoization to Sekitei resulted in huge (several orders of magnitude) speedup on some instances of the webcast problem and a small increase of running time

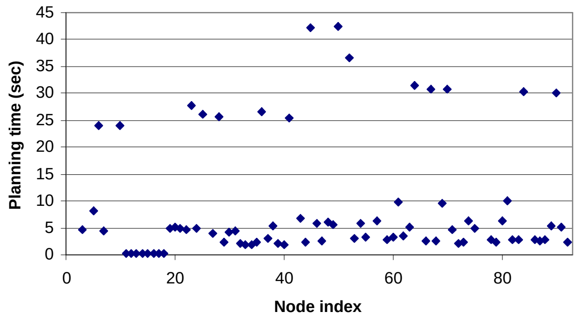

Figure 6: Planning time for placing a webcast client on each node of a 93-node network given a fixed location of the server.

on simple problems. Figure 6 shows planning time for the webcast application on a 93-node network. Intuitively, the use of positive memoization in planning with resources is similar to use of binary mutex relations in planning graph-based algorithms.

Unfortunately, positive memoization has high memory requirements. Having resource maps for all sets of propositions (essentially, most of the subsets of sets of propositions for each layer of the planning graph) leads to a worst case exponential memory explosion. In addition, now we need to explore all possible ways to achieve a set of propositions, which means we are doing some unnecessary work.

Focused Search

We are currently investigating two possible solutions to the memory explosion problem mentioned above. These include exploring only the most promising paths and saving optimistic maps per proposition rather than per set of propositions.

One possible way to identify the most promising paths in the component placement problem is to start by building a direct connection between the client and the server along the shortest path in the network, and then deviate from this path and add components only in case of a resource conflict. Currently, we are working on combining a regression-based evaluation function (Bonet & Geffner 1999) with positive memoization ideas for early resource conflict detection to produce a heuristic search-based planner that supports complex resource expressions.

Another way to improve performance of Sekitei is to use some properties of resources to prune search. It is often possible to distinguish between monotonic and general resources. A resource is **monotonic** if application of any operator changes its value in the same direction. If some operators can increase and others can decrease the value of a resource, we refer to such a resource as **general**. For example, available CPU is always a decreasing resource in the CPP, but the bandwidth of a data stream may be general if a caching component can be injected into the data path. We are currently investigating use of resource monotonicity information for early resource conflict detection.

Conclusions and Future Work

In this paper, we present the Sekitei algorithm for solving the component placement problem and possible ways to improve its performance. The CPP is a real-world problem, whose compilation into a planning problem is characterized by simple logical structure and arbitrary non-reversible monotonic resource functions. In addition, a planner for the CPP needs to cope with large number of irrelevant operators that cannot be removed by static preprocessing techniques.

Sekitei addresses the scaling problem by using a combination of regression and progression techniques to limit the search space. The positive memoization technique significantly increases performance of Sekitei by allowing early detection of resource conflicts. The main drawback of positive memoization is its high memory requirements. We discussed possible ways to address this problem.

Sekitei is designed and optimized specifically for the component placement problem. However, techniques developed for the CPP may be useful for other problems. We plan to extend our resource planners to support more general planning problems, namely, those containing operators with negative logical preconditions and effects.

The current version of our planner, as many other AI planners, minimizes the total number of parallel steps. In real world problems, such as the CPP, application of an operator usually involves some cost. It is more desirable to minimize the total cost of a plan rather than its parallel length. We plan to add support for operator cost into the next version of our planner.

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *AI* 90(1-2):281–300.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *ECP*, 360–372.

Fu, X.; Shi, W.; Akkerman, A.; and Karamcheti, V. 2001. CANS: Composable, Adaptive Network Services infrastructure. In *USITS*.

Helmert, M. Decidability and undecidability results for planning with numerical state variables. In *AIPS-2002*.

Ivan, A.-A.; Harman, J.; Allen, M.; and Karamcheti, V. Partitionable services: A framework for seamlessly adapting distributed applications to heterogenous environments. In *Proc. HPDC'02*.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2003. Constrained component deployment in wide-area networks using ai planning techniques. In *IPDPS*.

Koehler, J. 1998. Planning under resource constraints. In *ECAI-98*, 489–493.

Penberthy, J., and Weld, D. Temporal planning with continous change. In *Proc. AAAI'94*.

S. Gribble et al. 2001. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks* 35(4):473–497.

Wolfman, S., and Weld, D. 2000. Combining linear programming and satisfiability solving for resource planning.