A Framework for Automatic Problem Decomposition in Planning

Qiang Yang
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
qyang@logos.uwaterloo.ca

Shuo Bai and Guiyou Qiu
National Research Center for
Intelligint Computing Systems
P. O. Box 2704, Beijing 100080
Peoples Republic of China
ncic5%bepc2scs.slac.stanford.edu

Abstract

An intelligent problem solver must be able to decompose a complex problem into simpler parts. A decomposition algorithm would not only be beneficial for traditional subgoal-oriented planning systems but also support distributed, multi-agent planners. In this paper, we present an algorithm for automatic problem decomposition. Given a domain description with a number of objects to be manipulated, our method constructs subspaces complete with subproblem descriptions and operators, and solves the subproblems concurrently. The solutions in individual subspaces are combined using a constraint satisfaction algorithm. The effectiveness of the approach is guaranteed by our careful analysis of the interactions among different subspaces. The results presented in this paper support parallel, distributed and multi-agent planning systems.

Introduction

The ability to decompose a complex problem into manageable subcomponents is a necessity to many intelligent problem-solving activities. A problem solver using a problem-decomposition or problem-reduction strategy would first decompose a given problem into subproblems, solve each subproblem and then combine the solutions to obtain a global solution to the original problem. In distributed problem solving, an agent could be assigned to solve each of the subproblems. The advantage of problem decomposition cannot be overstressed; it facilitates concurrent problem-solving and reduces search complexity.

An important problem is how to compute a problem decomposition automatically. So far, the problem has only been superficially considered. In the past, a number of nonlinear planning algorithms (Barrett & Weld 1992; Chapman 1987; McAllester & Rosenblitt 1991; Sacerdoti 1974; Wilkins 1984) have been proposed, all decomposing a problem simply by splitting a compound goal into subgoals. Furthermore, no concurrent problem-solving is done; most planners solve all subgoals together. As a result, the problem-solving efficiency is gained only through the inherent partial-order representation of plans, but not from a reduction of the problem itself. The analysis performed in (Korf 1985) discussed various computational complexity issues related to decomposing a compound goal into subgoals. However, decomposition by goals does not always lead to the best possible decomposition. A similar branch of work is also done in (Foulser, Li, & Yang 1992; Yang, Nau, & Hendler 1992), the purpose of these algorithms being to *merge* plans for separate goals that have already been decomposed.

In distributed AI, a more recently developed group of planners are exclusively aimed at solving a complex problem by working on individual parts by multiple agents in a distributed way. Most of them, however, depend on the users to provide a decomposition *before* the algorithms can be used. The COLLAGE system (Lansky 1993) generates plans concurrently based on regions of activity. The regions function as a decomposition of the problem domain provided by the user. Another theme of work in distributed planning is to assign agents to tasks in a more or less optimal way. Here a typical example is the DMVT planner (Durfee & Lesser 1987), which decomposes the agents' environment by ranges of camera angles, assuming that corresponding to each sensor a dedicated agent exists.

In this paper we provide an automated method for decomposition to improve problem solving efficiency. The method is based on the interaction between the individual objects that constitute a domain. The theory under which the decomposition is based answers some important questions in AI: What is the nature of problem decomposition? What is a good decomposition strategy and what is a bad one? How should decomposition be related to solution combination at a later stage? What is the relationship between problem-solving using decomposition and using abstraction?

In the following, we first illustrate the highlights of the paper using a simple example. Then we will consider the general properties of problem decomposition, and use the properties to syntactically describe an algorithm. The algorithm is able to automatically generate a domain decomposition with improved planning efficiency. Finally, we discuss conditions under which our approach will work well, and discussed relations to other problem solving methods in AI.

An Example

To begin with, we consider a simple example to illustrate the main points.

Consider the following example where two boxes can be moved around three rooms. Figure 1 (a) depicts such a domain. Suppose that from room R2, the only way to transport any box through to room R3 is to use a cart to push the box together with one other box.

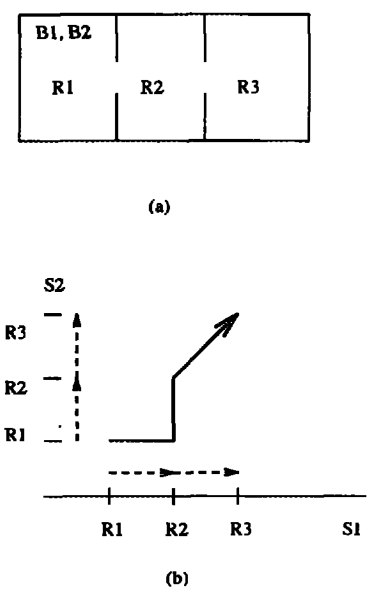

Figure 1: A robot box domain.

In this domain, a typical conjunctive goal is to rearrange the boxes in different rooms. As an example, a goal state in which both boxes B1 and B2 are in room R3 can be described as:

G1 $\wedge$ G2, where G1=Inroom(B1,R3) and G2=Inroom(B2,R3).

A conjunctive goal planner solves this problem by planning the two subgoals together. For example, in SNLP (McAllester & Rosenblitt 1991) or TWEAK (Chapman 1987) the two goals will be considered as the preconditions of a special goal operator, which is part of every plan in the search space. During the achievement of any one subgoal, a check must be made in the entire plan to see if any other goal is violated.

In contrast, a planner based on a problem-decomposition method separates the planning process for the movement of the two boxes. In each decomposed subdomain, it forms a solution plan for each box individually, and then combine the two plans to form a single global plan. In this example, this separation might correspond to solving each subgoal G1 or G2 concurrently.

Thus, according to the above decomposition, in box B1's view it is the only box in the domain. A plan for moving this box is:

move B1 from R1 to R2, then push B1 from R2 to R3.

Likewise, a plan for moving the box B2 is

move B2 from R1 to R2, then push B2 from R2 to R3.

Once the domain is decomposed and the subsolutions are found, the plans are next combined. Observe that we have a constraint for Room2: the cart can operate only when both boxes are in position. This requirement enforces that the two operators "move B1 from R2 to R3" and "push B2 from R2 to R3" be *merged* into one action: " push B1 and B2 from R2 to R3". Later we will see how this "merging" operation can be done automatically.

From this example we can see the advantages of problem decomposition. Since the operator set in each subproblem is smaller than the original one, the search is more manageable for a subproblem. Also because the problem is separated into parts with no precedence relation among them, concurrent processing is now possible to generate subsolutions in parallel. Furthermore, when more than one agent is available in a domain, decomposition makes distribution of tasks much more natural.

The Nature of Problem Decomposition

Subspaces and Projections

We envision decomposition as the following problem-solving activity:

1. Partition a given problem domain into N subspaces, $S_i, i = 1, 2, \ldots, N$.
2. Obtain a representation of the problem-solving operators with respect to each individual subspace S_i. The result is N classes of problem-solving operators.
3. Obtain a representation $P_i, i = 1, 2, \ldots, N$ of the input problem $\mathcal{P}$ in each subspace.
4. Solve the subproblems P_i using the operators in its corresponding space S_i.
5. Combine the solutions to the subproblems to obtain a global solution for the original problem $\mathcal{P}$.

An example is the quicksort algorithm in computer science.

In the above algorithm, steps 2 and 3 will be referred to as *operator projection* and *problem projection*, respectively. They correspond to decompose an operator set or a problem into N classes. In the simple robot domain, the operator projection separates the operators for moving box B1 and B2. The problem projection into the subspace containing B_1 consists of the initial state and goal state pair $(Init', G')$, where *Init'* are initial facts relevant to only B_1, and $G' = Inroom(B1, R3)$.

Steps 4 and 5 of the algorithm are tightly coupled, in that interactions exist among the decomposed subspaces. When a global plan is being obtained from the subplans, the subplans cannot merely appended to yield the final solution. Steps in the subplans may have to be interleaved.

The method of decomposition as described above can be illustrated graphically. Suppose that we have two subspaces after the decomposition is done, S1 and S2. Each of these spaces can be depicted as an axis on a two-dimensional space. Every point in the 2-D space corresponds to a state describing the original system. A projection of a point onto an axis corresponds to that of a problem. For example, the problem projection of the goal state onto S1 in the above example corresponds to point R3 on axis S1 (see Figure 1 (b)). An operator in the original problem space is an arc from one point of the space to another. The projection of an operator also corresponds to its geometrical counterpart. For example in Figure 1 (b), the operator "move two boxes from R2 to R3" is shown as the diagonal arc. The projection of this operator on both subspaces B1 and B2 are shown in dashed lines in that figure.

A plan can also be projected onto a subspace by means of the above projection operation. When subplans are combined, the two projections of the diagonal arc are "merged" into one action in the original space.

The above description can also be generalized to an n-dimensional view for any $n > 2$. Given this intuitive picture of the nature of problem decomposition, problem solving activities have a corresponding intuitive interpretation. Recall that once a problem is decomposed subsolutions are then sought for in each dimension. To solve the original problem, a combination phase corresponds to the construction of the a path in the original space from the initial state to the goal state , given the subsolutions in all dimensions. This path construction, as we will describe later in the paper, can be best described by a *constraint satisfaction* process.

The Quality of Decompositions

Consider again the above algorithm for decomposition. To make sure that the complexity of solving an individual subproblem is small enough for the overall gain to be worthwhile, we must make sure that the numbers of operator schemas in different subspaces are about the same. That is, when the decomposition is *even*, we will not get into a situation where a large amount of search is required for just a few subproblems. This could ensure that the concurrency property of the approach is fully realized.

For step 5 to have a low complexity, we would like the amount of interactions between any two subdimensions to be limited. We call this the *effective* property. Put together, to have a *good* domain decomposition, we desire

Even Decomposition: the difference in the number of operators between any pair of subspaces is no greater than a user-defined constant γ.

Effective Decomposition: the amount of interactions between any two subspaces during planning is no larger than a user-defined constant θ.

A Decomposition Algorithm

We now consider how to obtain a good domain decomposition in detail.

Algorithm DECOMPOSE

Input: a set of domain objects with types, a set of operators, an initial state and a goal state and a threshold value θ for interaction control.

Output: a set of subspaces, $S_i, i = 1, 2, \ldots N$. Each space contains a set of objects, a sublanguage for describing the domain, and a set of operator projections.

1. Select a view V of the domain. The definition of a *view* is presented in Section 4.2.
2. Partition the objects in V into subspaces $S_i, i = 1, 2, \ldots, |V|$.
3. Perform operator and goal projections onto the subspaces.
4. Merge any pair of subspaces S_i and S_j for which the amount of interactions is greater than θ. Repeat this step until no such S_i and S_j can be found.
5. Perform topological analysis of the graph of subspaces.
6. Output the resultant set of spaces $S_i, i = 1, 2, \ldots, N$.

Below, we explain the steps in this algorithm in detail.

Domain Language

We use an extended STRIPS operator language to describe a domain, with *Precondition*, *Add* and *Delete* lists of literals. A plan is described as a partially ordered set of operators. A *correct plan* is one in which every total order of the plan can be executed successfully while satisfying all operator preconditions under the STRIPS Assumption.

In addition to the operators, it is assumed that the input consists of a set of objects in the domain, together with a set of *types*. Each type simply consists of a set of objects in a domain that share some common properties. In our robot box domain, there are two types of object, **Box** and **Room**. A predicate $\texttt{Inroom}(x, y)$ states that the object x of type **Box** is located in another object y, of type **Room**. The operators in this domain describe the movement of one or several boxes from one room to another. As an example, the operator for moving two boxes $?b_1$ and $?b_2$

from a room R2 to R3 is described as follows (we follow the convention that variable parameters are preceded by a ? sign):

Operator move23 ($?b_1, ?b_2$), where ($?b_1 \neq ?b_2$).
Type: $?b_1, ?b_2$: **Box**; R2,R3: **Room**;
Preconditions: Inroom($?b_1$,R2),Inroom($?b_2$,R2);
Effects: Inroom($?b_1$,R3),Inroom($?b_2$,R3),
$\neg$ Inroom($?b_1$,R2),$\neg$ Inroom($?b_2$,R2).

Multiple Views

Our domain decomposition process starts by assigning each object in the problem domain a subspace. Initially, a domain with n objects is decomposed into n separate subspaces. When put together, the status of all n objects completely describe an entire state.

On a closer look, the above decomposition results in a certain amount of redundancy, because it is possible that a subset of the objects can also completely describe a domain. In the robot-box example, the objects consist of both the boxes and the rooms. Once the location of all boxes are fixed, a state of the domain is certain. It would be redundant to further specify the status of the room objects in terms of the set of boxes contained in each room.

In general, to completely specify the states of a domain, one can take different points of view depending on what objects to choose. A *view* is a subset of objects the specification of which can completely determine a domain state. In the robot-box example, one can take the box point of view and specify a state by two subspaces, one for each box. Within the subspace for B1 (or B2), a state is described using a derived predicate *Inroom*($B1, y$), where y is a variable that ranges from R1 to R3. Likewise, one can take a point of view of rooms, in which case a subspace corresponding to a room Ri is specified by *Inroom*(x, *Ri*), where x is either B1 or B2.

A domain with n objects has 2^n candidates for alternate views, one for each subset. Only some subsets are views that can completely describe a domain. The following algorithm greedily finds out a view for a given domain. If we let it run until it exhausts all objects then we can find out all alternate views of a domain.

1. Suppose that the user has already classified the objects into T different types. Sort T by the number of objects associated with a type. Set V to be the empty set. V will eventually be a view of the domain.
2. Let t be the first element in the sorted list T. Remove t from T, and add it to V.
3. We have a complete view of a domain when the set of operators that can apply to the objects in V is equivalent to the operator set in the domain. If this is true, stop and output V. Else go to step 3.

Computation of Projections

Given a view V, we can consider each object in V as an individual dimension. Our task will be to find out those dimensions that closely interact with each other and merge them into single ones. To do this, we must perform a projection of the operator set and of the problem description.

The projection of an operator is similar to the construction of an ABSTRIPS operator. Let v be a set of objects and α an operator. The projection of α on a subspace v, denoted by Proj(α, v), is computed separately with its preconditions and effects. The precondition of Proj(α, v) is the set of literals in $Pre(\alpha)$ that contain only objects in v as an argument. The projection of add and delete lists is similarly defined.

An operator β in a subspace v can also have *inverse* projections, which are the set of operators in the original space whose projection is β. Although an operator has a unique projection, the inverse projection of an operator back to the original space can result in several operators. The set is denoted by InvProj(β).

Interaction Analysis

After establishing a view of a domain and having computed projections of operators in each subspace, we next analyze the criteria for a good domain decomposition. Intuitively, if two subspaces, or axis, have the potential of introducing a lot of interactions when subsolutions are combined, then it is better to merge them into one subspace. Our subsequent analysis is done by considering pairs of axes to ensure that the amount of interactions between them is limited below a certain bound.

Let X and Y be two axes under consideration. There can be three types of interactions between them:

operator merging interaction This occurs when the inverse projections of an operator x in X and of an operator y in Y correspond to the same operator u in the original space.

deleted condition interaction This occurs when an inverse projection of an operator x in X deletes a precondition of an operator u, whose projection y is in Y.

added condition interaction This occurs when an operator x in X establishes a precondition of an operator u in original space, whose projection y is in Y.

Each of the three types of interactions adds an extra level of complexity for problem-solving, and they all contribute to the complexity in different ways. Therefore, they should be treated with care.

The first type of interaction increases the complexity level by the requirement of mapping the two operator projections x and y into the same operator U in the original space. To preserve completeness, this operation consists of a decision on whether to merge the inverse projection of x and y back to the same operator, or to keep two identities of the same operator u in the solution plan. The second type of interaction increases the complexity in a different way, by requiring that the two subsolutions in X and Y be *shuffled* to avoid the deletion of y's precondition. Since there are usually several ways to avoid a given deleted condition interaction, the complexity is also added up exponentially.

The third type of interaction makes it necessary for the subsolution of X to contain the needed operator in order to achieve the precondition of operator y in the subsolution of Y. Thus, intuitively, it is better to find a subsolution in Y first before a solution in X is found. In other words, the occurrences of added-condition interactions signals the need for *abstraction*-based methods. The quality of a domain decomposition depends on how the three types of interaction are limited.

We control the first two types of interactions using a *counting* method. Let θ be a user-defined threshold factor on how many interactions are allowed between any two axes. If the number of the first two types of interaction between X and Y exceeds this threshold value, then the two axes are combined into a single one: $X \cup Y$. This *subspace merging* operation is repeated until every pair of axes satisfies the requirement of limited interactions. If $|V|$ is the initial number of objects in a view of the domain, then the interaction analysis is done in time $O(|V|^3)$.

The third type of interaction can usually be taken care of by the use of abstraction hierarchies. We will return to this issue later in the paper.

Counting Interactions

In order to determine whether two subspaces should be merged, it is necessary to count the total number of interactions existing between any two dimensions. Let X and Y be any two dimensions. Currently we have found two ways to count the number of interactions, each according to a different assumption about the occurrence of actions in a plan.

Counting by Operator Schemas The first type of counting simply makes use of the operator schemas provided by the user. Given an operator schema α, if the projections of α on X and Y are both non-empty, then there could be possibility of operator-merging interaction. On the other hand, if there are two schemas α and β such that the projection of α on X and the projection of β on Y are both non-empty, and if α deletes a precondition of β, then there is a possibility of deleted-condition interaction.

If any of the above situations occurs for operator schemas, then the interaction count between X and Y should be incremented by one. The interaction count should reflect the average amount of interactions occurring during planning between the two subspaces. Therefore, for this count to be accurate, the following assumption must hold:

> The frequency of interaction occurring during planning is uniform for all operator schemas.

Counting by Example Plans One can think of cases where the above estimate is not accurate. For such cases, an operator instance may appear in a plan more often than the other instances of operators. If we have a set of example plans that reflect the relative frequency of operators occurring during planning, then we could obtain an estimate from these examples.

Let Π be an example plan. We can compute the projection of Π on both X and Y. Then a count may be conducted for each type of interaction.

In addition to the above domain-independent method for decomposition, a number of domain-dependent knowledge can be further enhance the effectiveness of the decompositions. One such domain knowledge takes into account the information about geographical regions. Suppose that we have some extra knowledge that object sets A and B belong to different regions that never or rarely cross each other. In that case, we can conclude that objects in A and B should belong to separate subspaces with little possibility of being mergeable. In such a case, the interaction analysis can be made more efficient by skipping the counting step of interactions between subspaces in A and B.

Solution Combination

Once solutions are found in each space we can then start combining the subsolutions into a global solution. The solution combination problem can be further split into two subproblems, each can be modeled as a Constraint Satisfaction Problem (CSP). The first problem is that of selecting a candidate solution from each subspace so they can be merged. The second problem is that, given a set of candidate solutions, how they can be best merged through conflict resolution methods. Both problems can be solved using methods provided for solving general CSPs, and from conflict resolution strategies. Both these methods have been implemented in a system called WATPLAN (Yang 1992).

Properties of the Decomposition Algorithm

We would like to enforce completeness of our decomposition method: if there is a solution to the original problem, then we wish that a candidate subplan exists in each subspace such that the subplans can be combined into a correct global solution. This guarantee is formally summerized as follows. Suppose that (1) a solution plan Π exists for the problem $(Init, Goal)$, and that (2) the projection of *Goal* on any axis is non-empty. Then a candidate solution plan exists in each subspace S_i such that they can be successfully combined to obtain Π. Due to lack of space we will not provide a formal proof here.

Relationship to Theories of Abstraction

In addition to the basic algorithm, we have also explored a number of domain heuristics that can make the decomposition more effective. Some such heuristics include decomposition by geographical regions, by topology of the interaction among subspaces, and by learning. In addition, we have investigated a mechanism by which the user can specify the threshold value of interaction count θ in an intelligent way. These extensions are summarized in a separate report. Here, we take a close look at the relationship between problem-solving methods using decomposition and abstraction.

Our framework of domain decomposition has a number of similarities with abstract planning theory. Planning with abstraction starts with a hierarchy of abstract spaces, each having its own operators. An abstract plan is constructed at each abstraction level, in a top-down fashion. Each successively higher level of abstraction is a simplified version of the original space. In the robot-box example, plans for moving boxes might be constructed before plans for both moving boxes and opening doors.

Like abstraction, each of the subspaces as a result of the domain decomposition is also a simplified version of the original space. However, unlike abstraction, a simplified plan in each subspace is computed *concurrently* in decomposition rather than successively. The solution plans in each subspace are computed in a distributed manner and combined together in a last step, instead of one refinement at a time. Thus, in the robot-box example, a subplan for moving each box is built first. Then all subplans are combined and merged in a final step.

Abstract planning builds sequences of operators first at the abstract levels. In this way, each abstract plan partitions the original problem into a succession of subproblems. Because the subproblems are embedded in the abstract solution, there may be strong temporal precedence relationship between these subproblems. Domain decomposition, on the other hand, separates a problem in a different manner, breaking down a complex problem into a set of subproblems not in terms of temporal constraints, but by the inherent topological structure of the problem itself. As a result, the subproblems cannot be simply appended to each other as done in ABSTRIPS, but must be interleaved with respect to each other.

Our algorithm for domain decomposition also has a number of similarities to algorithms for automatically constructing abstraction hierarchies (Knoblock 1994; Bacchus & Yang 1992); they are all based on analysis of interactions. Again, significant differences exist. Rather than requiring that between any two subspaces, no interactions can occur, our decomposition algorithm aims at limiting the amount of interactions between them. This difference between abstraction and decomposition was aptly pointed out by Lansky in (Lansky 1993).

Acknowledgement

We thank NSERC and the China-Overseas Center for Computer Science for support, and Drs. MingFa Zhu, Guojie Li and Guanghua Li for valuable comments.

References

Bacchus, F., and Yang, Q. 1992. Downward refinement and the efficiency of hierarchical problem solving. Technical Report cs-92-45, Department of Computer Science, University of Waterloo, Waterloo, Ont. Canada, N2L 3G1. *Artificial Intelligence*, to appear.

Barrett, A., and Weld, D. 1992. Partial order planning: Evaluating possible efficiency gains. Technical Report 92-05-01, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98105.

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

Durfee, E., and Lesser, V. 1987. Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, 875–883.

Foulser, D.; Li, M.; and Yang, Q. 1992. Theory and algorithms for plan merging. *Artificial Intelligence* 57(2).

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* to appear.

Korf, R. 1985. Planning as search: A quantitative approach. *Artificial Intelligence* 33:65–88.

Lansky, A. L. 1993. Localized planning with diversified plan construction methods. Technical Report FIA-93-17, NASA Ames Research Center, AI Research Branch.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, 634–639. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Wilkins, D. 1984. Domain-independent planning: Representation and plan generation. *Artificial Intelligence* 22.

Yang, Q.; Nau, D. S.; and Hendler, J. 1992. Merging separately generated plans in limited domains. *Computational Intelligence* 8(4).

Yang, Q. 1992. A theory of conflict resolution in planning. *Artificial Intelligence* 58(1-3).