Anytime Replanning Using Local Subplan Replacement

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Gabriel J. Ferrer

May 2002

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

Gabriel J. Ferrer

Approved:

Worthy N. Martin (Advisor)	John C. Knight (Chair)
John A. Stankovic	Gabriel Robins
Glenn S. Wasson	Dennis R. Proffitt

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

May 2002

Abstract

Planning is a very hard problem in general. Many interesting planning problems are NP-Complete, PSPACE-Complete, or undecidable. Furthermore, in the context of a planner being used to control a robot, events beyond the control of the robot may occur that cause its plan to fail. When a plan fails, it is necessary to devise a new plan that accommodates the exogenous event or events that caused the old plan to fail. This new plan must be devised in a timely manner.

Anytime planners seek to allow the system to exchange execution time for solution quality. I have analyzed how local subplan replacement can be utilized to implement an anytime planning system that devises new plans to accommodate plan failures. Local subplan replacement consists of selecting subsequences of operators centered around failure points and replacing them with new subsequences of higher quality. The key idea behind local subplan replacement is that it utilizes the results of the search process that produced the original plan in order to guide the search for a new plan.

An anytime planning algorithm requires two key components: the rapid generation of an initial plan and a means for establishing a progression of improved plans. Local subplan replacement begins by quickly reparing the flaws in the original plan. This results in a plan that achieves all of its goals, but may be of low quality. Local subplan replacement is then used to improve the quality of the plan. At low subplan levels, short subplans are replaced. These replacements are generated relatively quickly. At higher subplan levels, longer subplans are replaced. This requires more time, but results in plans of higher overall quality.

My experimental results demonstrate that utilizing the results of the earlier search process is effective for organizing anytime planning. I also describe some theoretical and practical limitations of this approach and suggest directions for further research.

Contents

List of Figures

1

Introduction

1.1 The Planning Problem

Planning, that is, generating a sequence of actions for an agent to perform in order to achieve a set of goals, is a very hard problem in general. Many interesting planning problems are NP or PSPACE-Complete [33] [17], or undecidable [18].

A large number of different approaches and heuristics that have appeared in the planning literature attempt to make planning practical. All these approaches attempt to find a way to limit the space that is searched for a plan. Sacerdoti [54] devised a system that began by creating abstract plans. Detailed plans were created by his system once the abstract plans were complete. Plans with unsuitable abstractions were ignored by the search process, a considerable computational savings. Many planning systems (e.g. [9] [42], [40]) use domain-independent search heuristics to limit the space searched. Many other planning systems (e.g., [7] [59] [47]) enable the user to specify domain-dependent search control rules to limit the space searched.

Reactive control systems (e.g. [16]) seek to eliminate online search entirely. In such systems, all planning happens at design time, when the control system is configured. At run-time, the control system functions essentially as a lookup table, with actions indexed by the sensor inputs. Hybrid approaches (such as [13] [5] [15]) have demonstrated the need for incorporating planning into a robot control system and have provided mechanisms for enabling planning and reactive control systems to work together. These architectures leverage the best features of both types of action selection system; the reactive components focus on action selection with short time horizons, while the planning systems address long-term system control issues.

Many planning domains involving robots have the property that events beyond the control of the robot may occur that cause the robot's plan to fail. A *plan failure* occurs when, during the execution of the plan, an operator's precondition is not met, making it impossible for the plan to continue. When a plan fails, it becomes necessary to devise a new plan that accommodates the exogenous event or events that caused the old plan to fail. This new plan must be devised in a timely manner. In a *valid plan*, all operator preconditions are true when applied, and all goals get achieved when the plan completes, assuming that the plan runs to completion without the occurrence of any plan failures.

Many systems currently exist that attempt to utilize some of the operators present in a failed plan in order to replan. Existing algorithms for replanning either fail to guarantee that the resulting plan will complete all specified goals [51], require human input [27], or make no guarantees about when a valid plan will become available [61] [38] [31]. Algorithms in the latter category can, in many interesting situations, require exponential search to guarantee a valid plan.

Theoretical work by Nebel and Koehler [49] has shown that, in the worst case, attempting to utilize operators from an existing plan in order to replan is just as computationally complex as generating a new plan from scratch. They acknowledge, however, that there are many situations in practice where utilizing operators from the original plan is efficient.

Motivated by the need to have an executable plan generated online and available upon demand, anytime planners (e.g. [63] [2] [24]) seek to allow the system to exchange execution time for solution quality. That is, the quality of a plan is proportional to the amount of time such a planner has available for execution. Should the planner be interrupted while executing, the best plan available at the point of interruption is what the planner returns. Anytime planners typically select one aspect of a plan to progressively improve. Some planners (e.g. [63]) improve the level of detail in a plan as time permits, initially creating only a very abstract plan. Other planners (e.g. [24] [28] [29]) progressively improve the probability of plan success. The interruptibility of these latter planners relies on being able to incrementally improve the probability of plan success. In order for such planners to function properly, it is important to model the probability of operator success as accurately as possible.

I have investigated a technique for the implementation of an anytime replanning algorithm. The technique uses a form of plan improvement I call *local subplan replacement.* The idea is to improve a plan by replacing progressively larger plan segments.

An anytime planning algorithm should possess the following characteristics:

- An executable plan should be available almost immediately.
- As the algorithm progresses, the quality of the new plan should improve, or at least it must not get worse.

As a means of achieving the second property, local subplan replacement seeks to use operators from the original plan in order to improve the repaired plan quality at all stages. Throughout this process, plan validity is preserved.

In order to achieve the first property, local subplan replacement needs a valid plan to be available in polynomial time. This is not possible for every planning problem. For some planning problems, finding a valid plan is NP-Complete, PSPACE-Complete, or even undecidable. However, there do exist many planning problems for which finding an optimal plan may be NP-Complete, but for which a valid plan can be found in polynomial time.

Such planning problems have been classified in several ways. Bylander [17] and Erol et al. [33] classified them based on restricting operator preconditions and post-conditions. Korf [43] showed how planning problems where the goals and subgoals are independent can be solved in polynomial time by finding a plan for each goal and subgoal and then concatenating the resulting plans. Yang et al. [62] extended this idea by showing how certain types of goal interactions could be handled efficiently.

Devising effective autonomous agents that employ anytime planning requires solutions for two related but distinct problems: designing effective anytime planning algorithms, and designing algorithms that determine how much time is to be allocated to the anytime planner (e.g. Boddy and Dean [12] and Zilberstein and Russell [64]). The second problem, deliberation scheduling, is beyond the scope of this work.

My dissertation research is concerned with the first problem, that is, investigating the behavior of techniques that can be used for the design of effective anytime planning algorithms; specifically, analyzing the performance of a replanning system using local subplan replacement in order to determine whether it is suitable for implementing an anytime replanning system. Part of the purpose of this is to achieve an understanding of the dynamics of how local subplan replacement handles different situations.

1.2 Thesis Statement

Local subplan replacement provides an effective means for organizing an anytime algorithm for planning, in the context of repairing a preexisting plan that has encountered a failure. It utilizes the results of the search process that produced the preexisting plan in order to guide the search for a new plan.

1.3 Local Subplan Replacement

In the search for the original plan, many alternative plans were considered, the best of which was returned. Local subplan replacement attempts to use this original plan to guide its search for a new plan. If only a small amount of time is available, it will only consider plans that are very similar to the original plan. If time permits, local subplan replacement will consider plans that are increasingly different from the original plan.

Local subplan replacement provides a particular means of organizing the search for a new plan. Global replanning provides a more traditional means of organizing the search for a new plan. Both can be implemented as anytime algorithms. I have used global replanning as a baseline algorithm for assessing the performance of local subplan replacement; in particular, to determine whether exploiting elements of the search that was used to generate the original plan is preferable, and if so, under what circumstances.

1.4 Contributions

Here are the contributions of this dissertation:

- No existing systems for plan repair (e.g. [27] [61] [20] [38] [53]) implement anytime planning so as to guarantee always having a valid plan available. Local subplan replacement is the first anytime replanning algorithm to do this. On average, the initial valid plans are available within two seconds.

- For the Repair Robot planning problem, local subplan replacement is demonstrated empirically to generate plans that do a better job of maximizing factory production than those generated by a global replanning algorithm.

- Limitations of local subplan replacement are demonstrated empirically. High values for the repair penalty, a greedy measure of the distance in plan space between the original failed plan and a valid repaired plan, are shown to correspond with local subplan replacement requiring large amounts of search to generate a plan of lower cost than the corresponding baseline.

- The effect of using the original plan to guide the search is quantified and measured empirically. The plans that local subplan replacement generates contain, on average, significantly more operators from the original plan than those generated by the baseline. In many cases, this utilization corresponded to a lower plan cost than the baseline, empirically demonstrating the benefit of utilizing material from the original plan.

- The global replanning algorithm used as a baseline is itself a new anytime planning algorithm. Unlike existing systems, it will always have available a valid plan, and it takes responsibility for generating the first plan in its anytime progression.

- Existing theoretical results regarding categories of planning problems for which valid plans can be found in polynomial time are unified. Delete effects involving resource consumption are identified as a major contributor to the intractability of finding valid plans in polynomial time for certain planning problems.

1.5 Guide to This Dissertation

In Chapter 2, I give a rigorous definition of the planning problem, as well as definitions for plan repair and anytime planning. I describe the characteristics for what I consider to be a good anytime planning system. I describe what is known about the computational complexity of planning. I describe characteristics of target problems of my research and give several examples. I then define the Repair Robot planning problem. This planning problem is used as a running example throughout my dissertation.

In Chapter 3, I describe examples of different types of planning systems from the literature. Existing systems that perform replanning and anytime planning are discussed in detail.

In Chapter 4, I describe the anytime planning system that will be evaluated in order to investigate the thesis statement. Of particular importance is local subplan replacement, described in Section 4.4, which is the focus of my analysis in this research. Also described in Chapter 4 is an algorithm for repairing failed plans (Section 4.3), and the planning algorithms I used for implementing and evaluating local subplan replacement (Section 4.2).

In Chapter 5, I present the results of the experiments I conducted in order to evaluate the thesis statement. I describe the experimental design used, I show the results of my experiments, and I discuss how the results demonstrate the utility of local subplan replacement in the context of anytime planning.

Chapter 6 summarizes the contributions of this dissertation and gives directions for future work.

2

The Planning Problem

2.1 Planning and Robotics

The term "planning" is used in two major ways in the robotics community. One type of planner used in robotics is called a *path planner*. A path planner attempts to construct a geometrical path for a robot leading from its initial position to its target position. This could apply to a mobile robot attempting to travel from one position to another, and it could apply equally well to a robot manipulator maneuvering to position itself to grab an object. The world model used by a path planner will typically consist of a geometrical model of the environment and a geometrical model of the robot.

A more abstract type of planner, used in robotics applications as well as many other domains, is called a *classical planner* [58] or *task planner*. Such a planner attempts to construct a sequence of operators (or actions) that will enable the robot to achieve its goals. This sequence of operators is constructed based on reasoning about a world model. Many different approaches to representing the world can be found in the literature. An assertional database is the most commonly used approach for representing the world in this paradigm. My dissertation research focused specifically on task planning, the subject of the rest of this chapter.

2.2 Definition of the Planning Problem

2.2.1 Propositional STRIPS Planning

For the purposes of this dissertation, and loosely following the definitions given by others (e.g. [34] [17]), I define *STRIPS planning* as follows:

A *STRIPS planning domain* consists of two finite sets: a *predicate set* and an *operator set*. The predicate set represents a state using predicate notation. Each predicate in the set is a propositional variable that can take the values `true` and `false`. Each operator in the operator set consists of two satisfiable conjunctions of predicates from the predicate set, the *preconditions* and the *postconditions* (the latter, also called the *effects*), each term of which is called a *precondition* or *postcondition*, respectively. Among the postconditions, each term has a truth value listed, those terms with true listed are called *add effects* and those terms with false listed are called *delete effects*.

A *STRIPS planning problem* consists of a STRIPS planning domain together with an *initial world state* and a *goal state*. A *world state* is a set of assignments of values to each predicate in the predicate set of the STRIPS planning domain. A *goal state* is a satisfiable conjunction of predicates from the predicate set. Each term in the goal state is called a *goal*.

An operator is *applicable* in a given world state only if its preconditions are true. To *apply* an operator to a world state creates a new world state by assigning to each predicate in the world state that corresponds to an effect the boolean value listed for the effect. An operator may be applied to a world state only if it is applicable.

A *STRIPS plan* (alternatively called a *total-order STRIPS plan*) is a totally ordered finite set of STRIPS operators where:

- The plan contains n operators, numbered from 0 to $n-1$
- The *current state* for operator 0 is the initial world state
- The *current state* for operator i, $i \geq 1$, is the state resulting from applying the postconditions of operator $i-1$ to the current state for operator $i-1$

A STRIPS plan is *valid* if and only if:

- Every operator is applicable in its current state; and
- The goal state is true when the postconditions of operator $n-1$ are applied to its current state.

A *partial-order STRIPS plan* is a directed acyclic graph with a finite number of nodes n where:

- Each operator in the plan is a node
- An edge from $node_x$ to $node_y$ indicates that a postcondition of $node_x$ establishes a precondition of $node_y$

A partial-order STRIPS plan is *valid* if and only if each STRIPS plan corresponding to a topological sort of the partial-order STRIPS plan is valid.

In the worst case, finding a plan for a STRIPS planning problem is PSPACE-Complete [17] [33]. A more detailed description of the complexity of STRIPS planning under different conditions is given in Section 2.3.2.

The *cost* of a STRIPS plan is the number of operators it contains. An *optimal* STRIPS plan contains the minimum number of operators that can transform the initial world state into a world state in which the goal state is true.

It is also possible for each operator to have a distinct cost value. In this case, the cost of a STRIPS plan is the sum of the costs of the operators that constitute it. It is further possible for the cost of an operator to be a function of the state in which that operator is applied. Again, the total plan cost will be the sum of the costs of its operators.

2.2.2 Plan Repair

The concept of plan validity (as defined above in Section 2.2.1) assumes that all changes to the world state are made by the agent executing the plan. Such an environment is a *static environment*. In a *dynamic environment*, changes to the world state can occur from sources other than the robot. Throughout this dissertation I will use "valid" assuming the context of a static environment even if the robot is actually in a dynamic environment. For the static case the system can determine if a plan is valid without actually executing the plan; rather the system need only model the execution, almost as a "thought experiment".

A *plan failure* occurs when, during plan execution, a precondition of the operator about to be applied is found to be false, thus rendering the plan invalid. A plan failure may result either from the planning agent failing to properly apply one of its operators during plan execution, or from an exogenous event. Either cause of a plan failure can be considered to be a feature of a dynamic environment.

A *replanning problem* is a planning problem in which the initial world state is the result of the application of a proper subset of the operators of a previously-generated plan, combined with a state change corresponding to a plan failure. A *plan repair* *algorithm* uses an existing failed plan to guide the search for a solution to a replanning problem.

Nebel and Koeler [49] did an extensive study of the theoretical benefits of generating a plan by utilizing parts of an old plan, as opposed to generating a new plan from scratch. Their work shows that it is not possible to achieve a provable efficiency gain by reusing parts of an old plan to generate a new plan. In other words, plan repair is computationally just as hard a problem as plan generation.

2.2.3 Anytime Planning

The *anytime algorithm* was originally defined as a distinct category of algorithm by Dean and Boddy [23] (who also gave many examples of algorithms from the literature that belong to this category) based on these two characteristics:

- The algorithm can be terminated at any time and will return some answer; and
- The answers returned improve in some well-behaved manner as a function of time.

Anytime planning systems are planning systems that have the characteristics of an anytime algorithm. From these characteristics, we can see that each anytime planner must have some means of quickly generating an initial plan, combined with a means for finding improved plans. The manner in which an anytime planner explores the search space for improved plans I call the *anytime progression*.

In addition to the defining characteristics of an anytime planning system given above, I consider the following to be desirable characteristics of an anytime planning system:

1. Any plan returned will always achieve all goals; in effect, only valid plans will be considered as possible solutions.

2. The planning system does not depend upon a specific type of plan evaluation metric.

3. The planning system is applicable to an appreciable range of task planning problems.

Item 1 is important because you want to know that before you execute the first operator that all the goals are achievable. Anytime planners that return partial plans often can provide no assurance as to when the operators to finish the plan will become available. Having an initial plan that achieves all the goals removes this uncertainty.

There exists a certain tension between items 1 and 3, in that being able to guarantee that a plan will achieve all of its goals excludes many task planning problems. Section 2.3.2 describes a number of categories of planning problems, including a discussion as to whether or not it is tractable for them to meet the constraint stated in item 1. For examples of domains for which this guarantee is tractable, see Section 2.3.7.

2.3 Computational Complexity of Planning

The planning problem is undecidable in general [18]. Even under very restricted conditions it is often PSPACE-Complete or NP-Complete [17] [33]. This section will give an overview of the conditions that induce varying levels of computational complexity in planning problems and discuss some of the properties essential to the proofs of these complexities.

There is an important distinction to be made in how a planning problem is viewed that affects how we speak of its complexity. The same planning problem can be perceived as having drastically different computational complexity depending on the way in which it is classified. Some planning problems can be placed in two different categories, and in some cases those two categories can be shown to have two different worst-case computational complexities. The point is that demonstrating that a particular planning problem belongs in a particular category of computational complexity does not necessarily demonstrate a tight lower bound on the computational complexity of that particular problem. It may be that a tighter lower bound could also be demonstrated for it.

2.3.1 Decidable and Undecidable Planning Problems

David Chapman [18] proved the first important results about classical planning. He devised a planner (TWEAK) that represented a rigorous mathematical reconstruction of previous nonlinear planners. TWEAK is both a formal semantics for planning and a working, implemented planning system. Chapman proved that TWEAK is correct and complete; that is, if TWEAK terminates and returns a plan, the plan produced does in fact solve the problem, and if TWEAK returns signaling failure or fails to halt, no solution exists.

He then used the TWEAK formalism to show that planning is undecidable in general. Chapman's first undecidability theorem states that planning is undecidable in TWEAK. The key element of the proof is that Chapman shows that a Turing machine with its input can be encoded as a planning problem in the TWEAK representation. This can be done in part because TWEAK can represent an infinite (though still recursive) initial condition and an infinite number of constants.

The plan language TWEAK uses is restricted to contain constant symbols and predicate symbols. If function symbols are allowed, the theorem proving TWEAK to be correct and complete is no longer valid. Additionally, once function symbols are allowed, Chapman's second undecidability theorem states that planning is undecidable even if the initial state is restricted to be finite. To prove this theorem, Chapman encoded a two-counter machine as a planning problem in the TWEAK representation augmented with function symbols. Because any recursive function can be computed by a two-counter machine, TWEAK with function symbols can also compute any recursive function.

Erol, Nau and Subrahmanian [33] show that the presence or absence of conditional operators in the plan language has no effect on the decidability of the planning domain. The distinction rests on the fact that a conditional operator is defined to have a finite set of mappings from a finite input set to a finite output set, restrictions that are not placed on function symbols.

2.3.2 Tractable and Intractable Planning Problems

Bylander [17] explored a series of restrictions that can be placed on planning within the realm of decidable STRIPS planning problems, and devised a hierarchy of complexity results for planning. The planning problems he examined were classified into PSPACE-complete, NP-complete, and polynomial time categories. Ways of varying planning problems include:

- Limiting the numbers of preconditions and postconditions
- Disallowing postconditions that negate the state change caused by the postcondition of another operator (a negative postcondition)
- Disallowing preconditions that require the value of a predicate symbol to be `false` (a negative precondition)
- Limiting the total number of goals the planner can achieve

Bylander gives complexity results for both the problem of determining whether a plan exists (PLANSAT) and whether a plan of k steps or less exists (PLANMIN). Planning problems Bylander showed are PSPACE-complete (for both PLANSAT and PLANMIN) include:

- No restrictions on preconditions and postconditions
- One postcondition
- One precondition
- Two positive preconditions and two postconditions

Problems shown to be NP-complete for both PLANSAT and PLANMIN include:

- Only positive postconditions
- One precondition and one positive postcondition

Problems where PLANSAT is polynomial but PLANMIN is NP-complete:

- Positive preconditions and one postcondition
  - A version of the well-known Blocks World problem [36] falls in this category
- Positive preconditions and positive postconditions
- No preconditions and unrestricted postconditions
- No preconditions and two postconditions
- No preconditions and three positive postconditions

Unsurprisingly, the PLANMIN polynomial time planning problems possess extremely severe restrictions:

- One precondition and a constant upper bound on achievable goals
- No preconditions and one postcondition
- No preconditions and two positive postconditions

Although Bylander's complexity analysis is done in terms of propositional STRIPS planning [34], his results specify how hard it is to find a sequence of actions that accomplishes a set of goals, regardless of how the action sequence is generated. Bylander's analysis is domain-independent; a practical planning system could attempt to utilize domain-specific knowledge in order to reduce the computational complexity of a given problem. For example, the fact that two goals are independent of each other can be directly exploited by the planning algorithm of Korf [43] (see Section 2.3.3 for details).

As we can see in the categories above, restricting postconditions to being positive tends to cause a decrease in computational complexity. The presence of negative postconditions tends to inhibit the ability to find valid plans in polynomial time. This is because a negative postcondition can delete an already-achieved goal (or subgoal). Since this can happen as an undesired side effect of an operator that achieves a different goal, it can be difficult to discover an operator sequence in which these conflicts do not occur. These conflicts are called *deleted-condition interactions.*

If none of the delete effects are able to delete the predicate added by an add effect, then the domain can be transformed to have no delete effects [17], potentially improving its worst-case computational complexity. If alternative means can be found for eliminating conflicts between positive and negative postconditions, the worst-case computational complexity can also be improved. See Sections 2.3.4 and 2.3.5 for examples of this.

One interesting category in which finding a valid plan is still tractable even with allowing negative postconditions is domains in which all preconditions are positive and there is only one postcondition per operator. As was shown by Bylander [17], a polynomial time algorithm for finding a valid plan for such a domain is to first achieve all the positive goals, then all the negative goals. Because all preconditions are positive, the only predicates that will be negated are predicates whose negation is a goal condition. Hence, if any deleted-condition interactions occur, the operators interacting can simply be swapped in order to eliminate the interaction.

Another interesting category consists of domains in which operators have at most one precondition, any number of postconditions, and a constant upper bound g on the number of goals. In those domains, a minimum cost plan can be found in polynomial time. With n predicates in the initial world state, the worst-case running time is multiplied by the constant n^g.

The key to this category is the restriction to one precondition. If more than one precondition were permitted, the resulting subgoal expansions could result in more than g goals needing to be achieved, violating that constraint.

In addition to Bylander's categories, Erol et al. [33] proved valid plans could be found in polynomial time for planning domains in which operators were limited to any number of positive preconditions and positive postconditions. Finding minimum length plans remains NP-Complete in this category.

Backstrom and Nebel [8] introduce a planning formalism called SAS+ to prove certain categories of planning problems to be tractable. Their tractability criteria are based on the following restrictions:

- **Post-uniqueness**: For each predicate, there exists at most one operator that achieves that predicate. That is, the goals and preconditions uniquely determine the operators to be used.

- **Single-valuedness**: Predicates that are operator preconditions that are not changed by the operator are called *prevail-conditions*. If a predicate is a prevail-condition for an operator, it may only be a prevail-condition for another operator if that other operator expects exactly the same value.

- **Unariness**: Each operator only changes one state variable.

- **Binariness**: All state variables have exactly two values.

They proved that planning problems for which post-uniqueness, single-valuedness, and unariness hold can be optimally solved in polynomial time. If post-uniqueness is relaxed (and it is a severe restriction), then a valid plan can be found in polynomial time, but finding an optimal plan is intractable.

2.3.3 Restricting Goal Interactions

Korf [43] observed that if the goals to be achieved are independent (that is, none of the operators used for achieving any goal interact with the operators used for achieving any other goal), and if achieving each individual goal can be done in polynomial time, then a valid plan can be found in polynomial time by achieving the individual goals separately and then concatenating the operators together. As goal independence is a rather strong assumption, this result was generalized by Yang, Nau, and Hendler [62]. They presented an algorithm that can take a set of plans, where each plan achieves a single goal, and merge these plans into a single plan that achieves all the goals. They show that complete goal independence is not necessary for this merging to occur. Plan operators may have the following categories of interactions:

- *Action-merging* interactions occur when the operators used to achieve different goals have redundant effects and can be combined.

- *Action-precedence* interactions occur when an action in the plan for one of the goals must occur before some action in the plan for another goal. If this precedence is not kept, then the plan will fail.
- *Identical-action* interactions occur when the same operator must be used in different plans. This is a more specific type of interaction than an action-merging interaction.
- *Simultaneous-action* interactions occur when two different actions must occur at exactly the same time.

Korf's algorithm can be seen as a special case of this plan merging algorithm, where none of the plans to achieve the goals have any interactions.

2.3.4 Plan Merging and Operator Restriction

In this section, I present problem transformations that demonstrate the relationship between the results presented in Sections 2.3.2 and 2.3.3.

Korf's plan concatenation algorithm ([43]; discussed above in Section 2.3.3) can be transformed into a planning problem where all the operators have no preconditions and one postcondition (see Section 2.3.2). For each independently achievable goal, there exists an operator which applies a set of state changes that makes the goal true. Because each goal is independently achievable, any of these operators can be applied in either the initial state or in any state resulting from applying any of these operators.

When the plans to be concatenated are not completely independent, but their interactions are restricted to the action-merging and action-precedence interactions described by by Yang, Nau, and Hendler [62], then plan concatenation can be transformed into a planning problem where all the operators have positive preconditions and positive postconditions. Identical-action and simultaneous-action interactions require explicit representation of time points or intervals in action definitions, or some other similar temporal representations. Actions with these representations are not expressible using STRIPS operators, and hence are beyond the scope of this discussion.

There are three components to the plan-merging algorithm:

- Generate a plan for each goal in polynomial time.
- Create a partially ordered graph encoding the action-precedence and action-merging interactions. Creating this graph is $O(n^3)$ in the worst case, where n is the total number of operators across all the plans [62].
- Combine the individual plans into a single plan.

The presence of the graph containing the action-precedence interactions enables valid plans to be found just as easily as the case without these interactions. This is because a total ordering of the actions can be extracted from the precedence graph. When that ordering is used with the simple form of plan concatenation discussed above, computational complexity remains the same.

Taking advantage of action-merging interactions has the effect of improving the cost of the resulting plan; finding a valid plan is no easier or harder in the presence of such interactions. We can transform this into an operator-restricted planning problem with positive preconditions and positive postconditions as follows. An `orig-op` is an operator in the original planning domain; a `multi-op` is an operator in the transformed planning domain.

For each independently achievable goal, there exists a set of `multi-ops`, each of which applies a set of state changes that makes the goal true. At least one of the `multi-ops` in the set can be applied in any state. Different `multi-ops` in the set can be considered to have different lengths, reflecting the fact that each `multi-op` is a shorthand for a sequence of `orig-ops`. The different `multi-ops` are present to reflect the possibility that although the goals themselves are independent, the actions taken to achieve each goal might have side effects that make it shorter to achieve another goal.

2.3.5 Dead-End States

A *dead-end state* is a world state from which there does not exist an operator sequence that achieves the goals. It has been suggested by Hoffmann [39] that the presence of dead-end states can indicate that a planning problem is computationally difficult. He also observes that the ability to reliably recognize the presence of a dead-end state can make a domain computationally less difficult.

In some cases, dead-end states can be avoided based on action-precedence relationships (see Section 2.3.7.2 for an example). A category of dead-end states that cannot be solved by ordering constraints is caused by resource contention.

For example, assume that each plan operator consumes a certain amount of a given resource, and assume as well that a sufficient amount of the resource is initially available so that any given task may complete. Assume further that there exists a subset of these tasks containing at least two tasks such that the initial resources available are not sufficient for the needs of all tasks in the subset. If the resource is not replenished, the plan reaches a dead-end state.

Handling dead-end states resulting from resource contention requires the existence of a "replenish" operator that restores the resource. This operator cannot itself have a limit on the amount of replenishment it can provide; if it does, when it runs out of its resource, another dead-end state is encountered. See Section 2.3.7.3 for some examples of domains of this sort with and without reachable dead-end states.

2.3.6 Exponential Length Plans

Another source of plan intractability is the problem of exponential length plans. There exist domains for which the shortest valid plan contains an exponential number of operators. In such domains, it is not even possible to apply all of the plan operators in polynomial time, much less find a valid plan in polynomial time.[1]

An example of such a problem is the Towers of Hanoi problem, depicted in Figure 2.1. In this problem, the goal is to transport the tower of blocks from the first position to the third position. Only one block may be moved at a time, and stacking a large block atop a small block is not allowed. The first four steps of a plan to complete this task are also given in Figure 2.1. The complete plan to move a stack of four blocks contains 15 operators. As the number of blocks to be moved increases, the minimum length of a valid plan increases exponentially [36].

2.3.7 Polynomial Time Examples

This section contains examples of several interesting domains in which it is possible to find a valid plan in polynomial time. These domains have all appeared in the AIPS planning competitions [46] [6]. The Blocks World domain (which has also appeared in the planning competitions) was shown to belong to this category by Bylander, among others [17].

[1]Dean et al. [22] have observed that by using macro operators, it is possible to find exponential length plans in polynomial time for planning problems with a hierarchical structure such as the Towers of Hanoi. Plan execution remains intractable, however.

Figure 2.1: Towers of Hanoi

2.3.7.1 Package Delivery Domains

In package delivery domains, the goal is to transport packages from their starting locations to their goal locations using any of various means of transport available. In the domain from the AIPS-2000 planning competition, the available vehicles are trucks and airplanes. Trucks can transport packages within a city, and airplanes transport packages between cities. To get a package onto an airplane, it must first be transported by truck to an airport. Here is the operator set:

- `load-truck` package truck location
  - Preconditions: truck and package at location
  - Postconditions: package on truck, package not at location
- `load-airplane` package airplane location
  - Preconditions: airplane and package at location
  - Postconditions: package on airplane, package not at location
- `unload-truck` package truck location
  - Preconditions: truck at location, package in truck
  - Postconditions: package not on truck, package at location
- `unload-airplane` package airplane location
  - Preconditions: airplane at location, package in airplane
  - Postconditions: package not in airplane, package at location
- `drive-truck` truck start-location end-location city

  - Preconditions: truck at start-location, start-location in city, end-location in city
  - Postconditions: truck at end-location, truck not at start-location

- `fly-airplane` airplane start-airport end-airport
  - Preconditions: airplane at start-airport
  - Postconditions: airplane at end-airport, airplane not at start-airport

One way to generate a valid plan in polynomial time would be to transport each package to its destination one at a time, with the packages transported in an arbitrary order. Finding an optimal plan is NP-Complete, because an optimal plan cannot be longer than an arbitrary valid plan, and the length of an arbitrary valid plan is polynomially bounded. Hence the optimal plan can be found nondeterministically in polynomial time.

2.3.7.2 Elevator Logistics

The Miconic Elevator domain from the AIPS-2000 planning competition requires finding a plan to transport various people to the correct floors of a building. The main challenge in finding a valid plan is ensuring that passengers who must never be left alone on the elevator are accompanied by passengers classified as “attendants”. There are numerous other constraints as well; some passengers are “mutually exclusive”, some are “nonstop”, and some can be “VIPs”. Beyond that, it is a matter of optimizing retrievals based on passenger origins and destinations so that the elevator makes the fewest stops.

The available operators are `stop`, `up`, and `down`. Applying `stop` causes all passengers for which this floor is the destination to disembark and all passengers for which this floor is the start to board.

In order to guarantee a valid plan, preconditions must be added to `stop` to guarantee that there must always be an attendant on board until all never-alone passengers have been delivered to their destinations.

2.3.7.3 Logistics with Resource Restrictions

The logistics domain described in section 2.3.7.1 can be varied by placing capacity limits on the vehicles and by causing the vehicles to expend fuel when they move. The MYSTERY domain from the AIPS-98 planning competition [46] incorporates these ideas. In that domain, fuel cannot be replenished. Consequently, with those modifications, finding a valid plan in polynomial time is likely intractable.

If a fuel supply depot with an inexhaustible supply of fuel is introduced, then it becomes possible to find a valid plan in polynomial time. Every operator is given an extra precondition that checks to make sure that the robot will have enough fuel to go to a refueling station after the application of the operator. If it does not, it sets a special predicate that triggers the application of a refueling operator.

Given this extra precondition, it should be possible to find a valid plan regardless of the order in which the goals are achieved. The extra precondition and the refueling operator have, in effect, removed the deleted-condition problem introduced by the fuel depletion effects of the operators.

2.4 Target Problems

2.4.1 Target Problem Characteristics

The planning problem as defined above in Section 2.2 is general enough for many problems to be encoded as planning problems. In this section, I describe some assumptions

I make about the types of planning problems that my research is attempting to address.

Theoretically, for any action selection problem, given a certain level of ability to acquire knowledge from the world, an agent could select all its actions from a lookup table rather than generating an action sequence in the form of a plan. The lookup table would be indexed by sensor inputs, dictating the correct action in any perceivable situation. This lookup table could also be indexed based on a world model containing information remembered from previous sensor inputs.

Planning becomes necessary when a lookup table containing responses for every possible situation of interest becomes too large. At that point, there is a tradeoff between space and time, and it is better to spend time generating actions on-the-fly rather than attempting to find more space to store a lookup table. It may also be intractable to determine the correct action for each entry in the lookup table, even if storing the table itself is not impossible.

In a related manner, there could also exist situations in which a lookup table is too large, but a relatively simple equation can still dictate actions. Servoing towards a target in a continuous space is an example of this. No lookup table would store all of the possible servo actions, but a compact control rule would still be sufficient for determining an action for any scenario for which the agent is equipped. However, if no such convenient rule (or set of rules) can be found, then a generative planner is needed.

Another issue is the number of distinct agents that the planner can employ towards achieving its tasks. If the number of tasks is less than or equal to the number of agents that can achieve them, then one agent can be assigned to each task, provided that there is no significant additional cost for using each agent. Actions for achieving the tasks can then be selected using a lookup table or control rule. If there is an additional cost for each agent employed, it can be worth considering alternatives to using all the agents, and we once again have a search problem on our hands. Likewise, if the number of tasks exceeds the available agents, alternative assignments of agents to tasks must also be considered.

The above discussion relates to determining whether a generative planner is of value at all. The remainder of the discussion relates to a discussion of the applicability of replanning and local subplan replacement.

As local subplan replacement is concerned with improving valid plans, a target problem should have a plan space containing a large number of valid plans of varying quality. Restricting the problem domain to those for which valid plans can be found in polynomial time ensures that a large number of valid plans are available. However, if most or all of them are of the same quality, or if the variation in quality is not consequential in the domain, then local subplan replacement is a waste of time.

As local subplan replacement seeks to use the original plan to guide the search for a new plan, it relies upon good solutions to the new planning problem to be reasonably close to the original plan within plan space. If they are too far away from the original plan, then using that original plan to guide the search will not likely be helpful. In Section 5.3.2, an empirical measure of this distance is defined.

Local subplan replacement can handle many types of failures, but it is important that failures not be more frequent than the rate at which new plans can be generated and enough operators applied for some tangible progress to be made towards a goal. What that rate is depends heavily on the domain. The underlying concept is that if failures are so frequent that planning actions sequences does not bring the agent any closer to its goal, planning is not helpful. If the failure frequency is such that planning can be helpful, then local subplan replacement has the potential to be of benefit.

Another perspective on the issue of failures relates to the requirement that it always be possible to find a valid plan if one exists. Each goal from the original plan must still be achievable, assuming that it remains a goal after the failure. As each failure involves the negation of a precondition of an operator that contributes to achieving a goal, there must exist some other operator that can either achieve that precondition or achieve the contribution of the operator of the failed precondition towards achieving the goal. Without such an operator, local subplan replacement cannot be applied.

For example, assume that a robot has a goal of delivering a letter to an office. If on the way to the office the robot drops the letter into a paper shredder, there does not exist a plan the robot can apply in order to achieve this goal.

Theoretically, even if a failure is such that all of the operators from the original plan are useless, local subplan replacement will still work. In practice, however, it is presumed that the nature of the failure will be such that a substantial proportion of the operators from the original plan will still make a positive contribution towards achieving the plan goals. Improving a plan by replacing subplans presumes that the underlying structure of the original plan will match that of a high-quality plan. What precisely this means depends upon the specifics of the given domain. Some examples are given below in Section 2.4.2.

2.4.2 Potential Target Problems

The target planning problem for my experimental investigation of anytime replanning is the Repair Robot problem, which is described in Section 2.5. This section describes some other planning problems to which local subplan replacement could be applied.

In general, these target domains are logistics planning problems. In a logistics problem, objects get moved and rearranged in various ways. The example domains given in Section 2.3.7 are all logistics problems. Many mobile robot task planning problems are also logistics problems, in that they involve the robot traveling to a location and performing an action relative to an object. Examples of such planning problems include a robot giving tours of an area [41], robot reconnaissance (such as planetary exploration with robotic rovers [19]), and robot trash collection [35].

Here are some examples of plan failures that could occur in these domains:

- In the trucks and airplanes problem described in Section 2.3.7.1, one of the vehicles allocated for a delivery could break down, requiring modifications to the plan to either repair the vehicle or enable another vehicle to complete the delivery.

- In the elevator domain described in Section 2.3.7.2, a passenger could request a new destination.

- In the guided-tour robot domain, large crowds or construction work could necessitate the selection of an alternative path on the tour.

- In the robot trash collection domain, trash to be collected could be relocated, or more trash could be introduced into the environment.

In all of these domains, it is important that failures not be so frequent as to overwhelm the ability of the system to use a plan to make tangible progress towards its goals. For example, in the trash collection domain, if each piece of trash is kicked out of the robot's reach every time it tries to grab it, planning is of little value. However, if the majority of attempts to grab trash succeed, then planning action sequences to obtain the trash can be of value. In the elevator domain, if passengers request new destinations too frequently, it may be best to simply travel to the nearest destination floor before a change of mind occurs. If, on the other hand, new requests are relatively infrequent, the original plan may still be of value.

Distinct from the idea of frequency is the utility of the structure of the original plan. For example, if an original plan for the elevator problem transports two people, and requests for the transportation of eight more people arrive, the structure of the plan that transports two people will not provide much guidance towards finding a high-quality plan for transporting eight people. A plan for package delivery that uses eight airplanes and one truck per city will not provide much guidance if seven of the airplanes are rendered inoperable. In contrast, if only two airplanes are disabled the means by which the other six airplanes transport packages may still be of value. One or two may be redirected, but the remaining airplanes can complete their deliveries without alteration of their routes. Furthermore, the planes to be redirected can be selected based on route compatibility. Determining how severe a failure can be handled by local subplan replacement requires separate evaluation for each domain.

2.5 The Repair Robot Domain

2.5.1 Problem Description

The Repair Robot planning domain is used as a running example throughout my dissertation. Its computational complexity is discussed below in Section 2.5.3.

In the repair robot domain, a single robot has to keep a set of machines operating. The machines produce widgets in a factory environment. Different machines produce different numbers of widgets for each time step. The robot's goal is to maximize overall machine productivity.

The machines break down from time to time. When a machine breaks down, the robot must replace all of that machine's broken parts so that the machine is able to resume operation. The robot can carry a limited number of parts at a time, and can obtain parts from three different storehouses. Machines may require different types of parts. Some types of parts might only be available from certain storehouses.

The robot is able to perform the following actions:

- Grab a part from a storehouse or machine
- Place a part in a machine or storehouse
- Travel from any machine or storehouse to any other machine or storehouse
- Activate or deactivate a machine

Whenever any machine breaks down, the robot generates a plan for repairing all broken machines. The cost of a plan is the total number of widgets that the machines *fail* to produce. For example, if two machines are broken, with the first machine producing 1 widget/time step and the second machine producing 2 widgets/time step, and the plan to repair the machines fixes the first machine after 3 time steps and the second machine after 7 time steps, the cost of that plan would be 17.

The cost of a plan is determined by adding the cost of each operator in the plan. The cost of an operator is the number of widgets that fail to be produced over the course of the duration of the operator. The cost of traveling between locations is proportional to the distance between those locations. All other operators are assumed to take one time step, and have a cost proportionate to the amount of widgets not produced during that interval.

Once a plan has been generated, plan failures may occur upon each attempt to activate a machine. Each failure consists of some number of the parts in the machine being activated breaking, thus rendering it impossible to activate that machine. The robot must then modify its plan to replace the newly broken parts for that machine.

2.5.2 Formal Problem Definition

The predicate set contains two groups of predicates. The first group contains type identifiers. Each predicate is specified by a predicate name and a list of variables (prefixed by question marks) indicating the predicate's parameters. These predicates are:

- `is-machine ?machine`
- `is-storehouse ?storehouse`
- `is-part-type ?type`
- `is-robot-part-slot ?slot`
- `is-machine-part-slot ?machine ?slot`
- `machine-part-slot-type ?machine ?slot ?type`

Note that "storehouses" are also considered "machines" that produce nothing, for the purposes of these bindings.

The second group contains predicates that represent the changeable world state:

- `not-machine-works ?machine`
- `machine-works ?machine`
- `robot-at ?machine`
- `machine-slot-full ?machine ?slot`

- `machine-slot-empty ?machine ?slot`
- `robot-slot-full ?slot ?type`
- `robot-slot-empty ?slot`

The four variables below are used in the operator cost functions and range over the positive integers:

- `cost`
- `num-machines`
- `production-rate ?machine`
- `travel-time ?machine-1 ?machine-2`

Also used by the operator cost functions in addition to standard arithmetic operators, there is a function called `widgets-not-produced`. Its return value is the sum of `production-rate ?machine` for all bindings of `?machine` where `not-machine-works ?machine` is true.

The operator set contains five operators:

- The operator `get-part-from` (Figure 2.2) transfers a part from a machine or storehouse slot to a slot aboard the robot.
- The operator `place-part-at` (Figure 2.3) transfers a part carried in one of the robot's part slots to a slot in a machine or storehouse.
- The operator `travel` (Figure 2.4) moves the robot between machines or between a storehouse and a machine. Its cost depends upon the distance between the origin and destination locations. With a longer distance, more production is lost.

- The operator `activate-machine` (Figure 2.5) returns a machine to production.
- The operator `deactivate-machine` (Figure 2.6) halts machine production, enabling the removal of its parts.

2.5.3 Computational Complexity

A valid plan can be found for any repair robot problem by retrieving and installing the parts for each machine in an arbitrary order. Both the order of part installation and the order of machine repair can be arbitrary. The length of the plan will be linear in the number of parts that need to be installed.

Finding an optimal plan is NP-Complete. It is in NP because a valid plan can be found in polynomial time. An instance of the traveling salesman problem can be transformed into an instance of Robot Repair. Map each city to a machine. Each machine has one broken part. The robot has a number of carrying slots equal to the number of cities. The robot starts at the storehouse, which contains all the required parts. All machines have a production rate of 1. The travel times between cities map to the travel times between machines. Hence, if we can find an optimal plan for the repair robot, we also have an optimal TSP tour.

`Get-part-from ?machine ?type ?robot-slot ?mach-slot`

- Preconditions:
  - `not-machine-works ?machine`
  - `robot-at ?machine`
  - `machine-slot-full ?machine ?mach-slot`
  - `robot-slot-empty ?robot-slot`
- Add Effects:
  - `machine-slot-empty ?machine ?mach-slot`
  - `robot-slot-full ?robot-slot ?type`
- Delete Effects:
  - `machine-slot-full ?machine ?mach-slot`
  - `robot-slot-empty ?robot-slot`
- Cost Function:
  - `cost` = `cost` + `widgets-not-produced`

Figure 2.2: The operator `get-part-from`

`Place-part-at ?machine ?type ?robot-slot ?mach-slot`

- Preconditions:
  - `robot-at ?machine`
  - `machine-slot-empty ?machine ?mach-slot`
  - `robot-slot-full ?robot-slot ?type`
- Add Effects:
  - `machine-slot-full ?machine ?mach-slot`
  - `robot-slot-empty ?robot-slot`
- Delete Effects:
  - `machine-slot-empty ?machine ?mach-slot`
  - `robot-slot-full ?robot-slot ?type`
- Cost Function:
  - `cost` = `cost` + `widgets-not-produced`

Figure 2.3: The operator `place-part-at`

`Travel ?start ?finish`

- Preconditions:
  - `?start` $\neq$ `?finish`
  - `robot-at ?start`
- Add Effects:
  - `robot-at ?finish`
- Delete Effects:
  - `robot-at ?start`
- Cost Function:
  - `cost` = `cost` +
    (`travel-time ?start ?finish` * `widgets-not-produced`)

Figure 2.4: The operator `travel`

`Activate-machine ?machine`

- Preconditions:
  - `robot-at ?machine`
  - `not-machine-works ?machine`
  - $\forall$ `mach-slot`,
    `is-machine-part-slot ?machine ?mach-slot` $\Rightarrow$
    `machine-slot-full ?machine ?mach-slot`
- Add Effects:
  - `machine-works ?machine`
- Delete Effects:
  - `not-machine-works ?machine`
- Cost Function:
  - `cost` $=$ `cost` $+$ `widgets-not-produced`

Figure 2.5: The operator `activate-machine`

`Deactivate-machine ?machine`

- Preconditions:
  - `robot-at ?machine`
  - `machine-works ?machine`
- Add Effects:
  - `not-machine-works ?machine`
- Delete Effects:
  - `machine-works ?machine`
- Cost Function:
  - `cost` = `cost` + `widgets-not-produced`

Figure 2.6: The operator `deactivate-machine`

2.6 Summary

In this chapter, I defined the planning problem and gave an overview of its computational complexity. I described different categories of planning problems for which valid plans can be found in polynomial time, and I demonstrated relationships between these categories. I articulated the impact of dead-end states and resource consumption on the possibility of finding valid plans in polynomial time. I described the characteristics of planning problems addressed by my research and gave several examples. Finally, I defined the Repair Robot problem, the running example problem found throughout my dissertation.

3

Planning Systems

This chapter provides an overview of planning systems described in the literature, with a particular focus on papers relating to the work presented in this dissertation. (For a general overview of the planning literature, see the collection of papers edited by Allen, Hendler, and Tate [1].) In particular, Section 3.2 focuses on planners that use search control to improve planner performance, Section 3.3 discusses different approaches to the problem of replanning, and Section 3.4 discusses anytime planning.

3.1 Domain-Independent Planning

This section describes a selection of significant domain-independent planning algorithms. A domain-independent planning algorithm only requires the user to specify the predicate set and operator set when defining a planning domain, and initial world state and goal state when specifying a planning problem to be solved. Many such planners exist other than the ones described in this section; the ones discussed here are included because of their wide influence and relationship to the work presented in this dissertation.

3.1.1 The STRIPS Planner

An early, highly influential planning system was the STRIPS planner of Fikes and Nilsson [34]. STRIPS introduced what is now called the STRIPS operator, containing a set of preconditions and postconditions. The preconditions are first-order predicate logic clauses, and the postconditions are simple predicates that are added to or deleted from a state. (See Section 2.2.1 for a formal description of STRIPS planning.)

The STRIPS algorithm searches for plans by starting at the goal state and determining what goal predicates are not true in the initial state. (This is called *means-ends analysis.*) A search begins for an operator that has as its postconditions one or more of the goal predicates. When such an operator is found, it becomes the last operator in the plan, and a search commences to find operators that achieve either one or more of the remaining goals, or one or more of the unsatisfied preconditions of the operators already present in the plan. The search continues, building the plan backwards, until either all goals and operator preconditions are satisfied either by operator postconditions or the initial state, or it is determined that it is not possible to add another operator to the plan. At that point, the algorithm backtracks and tries to find operators to establish preconditions at a later point in the plan.

STRIPS was reasonably effective for small domains, but did not scale up to larger problems. Bylander [17] has shown that general propositional STRIPS planning is PSPACE-Complete, a result discussed in more detail in Section 2.3.2.

3.1.2 Graph-based Planning

The Graphplan algorithm of Blum and Furst [9] makes use of a data structure called a planning graph to constrain the search process. It accepts STRIPS style operators in its plan language, and its state representation is restricted to predicates. The first stage of the algorithm is to generate the planning graph. This graph can be generated in polynomial time. The second stage of the Graphplan algorithm is to search the planning graph for a plan. It generates a partial-order STRIPS plan. While it is guaranteed to terminate, this search can take exponential time. For many domains, however, it finds plans in reasonable amounts of time.

The planning graph is a directed, leveled graph with two kinds of nodes and three kinds of edges. (A leveled graph is a graph in which the nodes can be partitioned into disjoint sets such that the edges only connect nodes in adjacent levels.) The first level is a *proposition* level and the levels from there alternate with *action* levels. The first proposition level contains the initial world state. The first action level contains all operators that could be applied in that state. The second proposition level contains all predicates that could possibly be true after applying the actions in the action level to the previous proposition level. Further levels are propagated in this fashion.

Edges represent relationships between actions and propositions. The action nodes in action level i are connected by "precondition-edges" to the predicates in proposition level i that are their preconditions, by "add-edges" to the predicates in proposition level $i+1$ that they add, and by "delete-edges" to the predicates in proposition level $i+1$ that they delete.

Two actions in an action level are considered to be *mutually exclusive* if no valid plan could possibly contain both. Graphplan determines whether actions are mutually exclusive in these two ways:

- Two actions *interfere* if either action deletes a precondition or add-effect of the other.

- Two actions have *competing needs* if there is a precondition of each action that are achieved by actions that are mutually exclusive of each other.

In the empirical studies of the authors, these rules were sufficient for Graphplan to find plans quickly in many domains. However, these rules do not exhaustively specify all possible mutual exclusion relationships. The authors observed that determining every single mutual exclusion relationship is equivalent to finding a legal plan, and hence computationally intractable.

The Graphplan algorithm incrementally grows the planning graph until all goal conditions are present in a proposition layer, and those goal conditions are not mutually exclusive. It then does a backwards-chaining search through the graph to find a plan. If it can't find a plan, it grows the graph another level and tries again.

A number of other planning algorithms have subsequently made use of planning graphs in various ways. One of the most interesting of these algorithms is the Fast Forward planner of Hoffmann and Nebel [40]. It plans by doing forward-chaining, that is, starting at the initial state and considering all operators whose conditions are satisfied in that state. It generates a total-order STRIPS plan. Upon adding an operator, it generates a new state and repeats the process until a state satisfying the goals is encountered.

The operator to use is selected based on solving a *relaxed* planning problem, that is, finding a plan that achieves all the goals while ignoring all delete effects. It uses Graphplan to do this. Because delete effects are ignored, no actions are mutually exclusive, and Graphplan runs in polynomial time. The operator for which the shortest relaxed plan can be found is selected, and the algorithm proceeds from there. It does not backtrack.

Fast-Forward does best in planning domains where deleted goal interactions are not a serious issue, such as logistics problems akin to the delivery domain described in Section 2.3.7.1. It does well on some Blocks World problems, but for some instances the ignoring of delete effects gets it stuck in local minima. The same phenomenon occurs with domains containing dead ends (as defined in Section 2.3.5).

Another use of planning graphs is as preprocessors for other search techniques. The Blackbox system of Kautz and Selman [42] uses a planning graph to generate a SAT encoding of a planning problem. If a SAT encoding can be satisfied, it implies the existence of a plan of a given length. The values of the SAT solution can be used to construct a plan. One of several different SAT solvers is used to solve the SAT encoding.

Blackbox did better empirically than Graphplan on several different planning problems. However, its memory requirements for SAT encodings of many problems are huge, so good performance has tended to be highly domain-dependent. Furthermore, whether instances of planning problems encountered in practice map well to the SAT instances on which SAT solvers do well is still a matter of some debate (c.f. [40]).

3.2 Planning with User-Assisted Search Control

Many planning systems eschew the domain-independent approach in favor of approaches where the user of the system provides extra information to the algorithm to enable the search to be more efficient. Some of these systems enable the user to specify a hierarchical decomposition of the planning problem for the planner to use in its search. Other systems operate by doing a forward-chained search and using user-supplied information to prune aggressively.

3.2.1 Hierarchical Task Networks

The earliest hierarchical task network (HTN) planning system was Sacerdoti's NOAH planner [55]. NOAH introduced the concept of partially-ordered planning; that is, orderings would only be added between operators when a conflict of some kind was detected between them. It also added the concept of a “critic” that could make constructive improvements to a plan being developed. Task-specific information was encoded in “procedural nets”. The nets were networks of nodes. Each node represented an action at a certain level of detail, and had edges to other nodes that refined that detail. Planning proceeded by finding a procedural net that could achieve a goal, and then adding further actions by detailing the action specified by that net. Unfortunately, because NOAH could not backtrack, it would in some situations commit actions that could lead it into dead ends.

The next task network based planner was the NONLIN planner of Austin Tate [57], which was based loosely on NOAH. It extended the idea of “critics” from NOAH into the concept of an “expert”. Experts provided guidance to the nodes in the task network as to how to detail their actions if choices were available. Since backtracking was permitted in order to avoid the incompleteness problems that plagued NOAH, the experts were very important in helping the system to make good choices for action detailing.

The NONLIN project has evolved over time into the O-Plan system [21]. O-Plan is based on the same core ideas, but has numerous extensions to enable the encoding of complex problems. In particular, O-Plan makes use of techniques from the operations research literature to solve timing and resource constraints. The SIPE-2 planner [60] is conceptually similar to O-Plan, but makes use of a different set of heuristics in order to plan efficiently.

The Simple Hierarchical Ordered Planner (SHOP) of Nau et al. [47] grew out of work in using HTN planning ideas in order to play bridge [56]. It operates similarly to most of the other HTN planners, except that task reductions are done in the same order in which the operators will be executed. As task networks are reduced to primitive actions, these actions are added to the plan in a forward-chaining manner. Because this enables the complete world state to be available at all times, much more aggressive pruning techniques can be implemented in SHOP than in partially-ordered HTN planners.

Erol, Hendler, and Nau [32] proved some general computational complexity results for HTN planning. They showed that STRIPS planning is essentially a special case of HTN planning. Unsurprisingly, then, the computational complexity of HTN planning was even worse than that for STRIPS planning. In particular, even without function symbols, if no restrictions are placed on non-primitive tasks, and the HTN can be partially ordered, planning can be undecidable in general. If the HTN is totally ordered and variables are not allowed, HTN planning is provably exponential; if variables are allowed, HTN planning is EXPSPACE-hard.

Restricting the task nodes to have at most one non-primitive task, following all the primitive tasks, HTN planning is PSPACE-Complete if variables are not allowed, EXPSPACE-Complete otherwise. Disallowing non-primitive tasks reduces the complexity to NP-Complete. If only primitive tasks are allowed and the HTN must be totally ordered, then a solution can be found by hill-climbing and polynomial time planning becomes possible.

In spite of these complexity results, claims have often been made in the HTN planning literature that HTN planners are useful for implementing polynomial time planning algorithms. These claims are derived from observing that task networks can be used to encode goal decompositions as described in Section 2.3.3. This use of task networks accounts for much of the good empirical performance of HTN planners in the literature (e.g. [47]).

3.2.2 Planning Using Temporal Logic

The TLPlan planner of Bacchus and Kabanza [7] enables the user to specify search control knowledge using temporal logic formulas. TLPlan uses a forward chaining state-space search, and each state is checked against the search control formula. If the state fails to satisfy the formula, then it is pruned. This state checking is made possible by the use of total-order forward search. TLPlan generates total-order STRIPS plans. (It can also generate plans in the ADL formalism of Pednault [50].)

The search control formulas are expressed using modal temporal logic. This logic extends first-order predicate logic with *always*, *eventually*, *until*, *next*, and *goal* modalities. Each modality except *until* takes one first-order predicate calculus formula as an argument. *Until* takes two first-order formulas as arguments.

The *always* modality is true if its argument is true for all states in a forward chain. *Eventually* is true if its argument is true at some state in a forward chain. *Next* requires its argument to be true in the immediately succeeding state. *Until* is true if the first argument remains true in a forward chain until the second argument becomes true. *Goal* is true if its argument evaluates to true in the goal state.

Using these modalities, search control rules can be specified that enable the implementation of polynomial time planning algorithms for certain domains, such as the blocks world.

TALplanner [44] uses the TLPlan approach as a starting point in combination with a more expressive temporal logic called TAL (Temporal Action Logic) [26]. Their domain encodings had performance superior to TLPlan across a range of domains.

3.3 Plan Repair

The ROGUE system [38] integrates a planner with an execution system for use on-board a robot. When plan failures occur, ROGUE calls the planner to generate new operators to accommodate the failure and bring the plan back on course. The system was designed primarily for dealing with simple failures; if a failure requires any significant amount of search, ROGUE can get tied up for quite some time generating a new plan. Local subplan replacement avoids this problem because it can implement anytime planning.

Plan repair in the O-Plan system [27] utilizes the automated planning system to locate where in the plan problems have occurred. The system then informs the user and makes recommendations for repairs. As O-Plan generates partial-order plans, these recommendations tend to focus on isolating problematic subgraphs for user attention. The user controls what repair is made to the plan. In contrast, using local subplan replacement for plan repair is completely automated, requiring no user intervention.

In the Cypress system [61], replanning occurs asynchronously when a plan failure is detected. Execution is handled by a reactive system. If the reactive system cannot handle a situation, it calls SIPE-2 [60] to replan the component of the plan that has failed. As SIPE-2 generates partial-order plans, these replaceable plan components are subgraphs that are selected with the intention of minimizing the impact of the change on other parts of the plan. This is done by, for example, selecting subgraphs with a minimal number of outgoing postconditions that are preconditions of later operators. The reactive system continues to execute operators that have no interdependencies with the failed subgraph. There is no explicit bound on the time needed to do the replacement. In contrast, local subplan replacement can implement anytime planning.

The CASPER system [20] uses a continual planning approach to plan repair. In continual planning (see [25] for a survey), “planning” and “plan repair” are not distinct from each other. The planner continually updates its plan to reflect changes in the environment that are detected. CASPER is thus an example of an anytime replanning system, as a plan is always available and constantly being repaired and improved.

The most important difference between CASPER and using local subplan replacement is that in CASPER there do not necessarily exist plan steps to achieve all system goals at all times. At each iteration, the planner attempts to resolve conflicts and achieve goals. Special purpose algorithms are used for devising plans for different sorts of goals, and for resolving conflicts between them. Local subplan replacement could, in theory, be one of the algorithms utilized by CASPER for plan updating. Local subplan replacement could similarly be utilized by other continual planning approaches in the literature [25].

Soar (see [53] for an overview) is an architectural framework and programming language for implementing intelligent systems. Action selection in Soar operates as follows ([45], also found in [53]). Soar uses a production system to select actions. Each time an action is to be selected, every rule gets compared against every state in the state set. When the production system is unable to select an action, either through a lack of information, or an inability to decide between two or more rules, a planner is invoked to resolve the problem. An explanation-based learning technique called *chunking* is used to store the result of the planning process in the production rule set.

As planning is done in Soar only to resolve problems as they occur, there is no distinction between “planning” and “replanning” in the Soar environment. As with many other systems, there is no bound on the time used for the planning process.

Again, replanning with local subplan replacement avoids this problem through its use of anytime planning.

3.4 Anytime Planning

The planning algorithm of Elkan [30] finds plans using a search approach inspired by logic programming, using a carefully engineered rule base. A plan is found by trying to unify a binding for a special logic variable P with the assumption that the goal state is true. Search proceeds in a back-chaining fashion. Each rule has a postcondition to be established on its left-hand side and the preconditions that imply that postcondition on its right-hand side. The search begins by finding a rule that has a left-hand side that establishes the goal state. The search proceeds by trying to bind the right-hand side of that rule.

The key difference between Elkan's planning algorithm and Prolog-style search is that Elkan used iterative deepening instead of depth-first search. The depth bound for iterative deepening starts at 1. Each time that a plan cannot be found, the depth bound is increased by 1 and a new search for a plan commences.

The system tries to prove that negated goals (and subgoals) are true by showing that no proof exists for the un-negated version of that goal. If the depth bound cuts off the search for a proof, then it is unknown whether or not that negated goal is true. To make the algorithm sound, the negated goal must be presumed false.

However, by presuming that the negated goal is true when its truth value is unknown, this algorithm becomes an anytime algorithm. The anytime progression is that the algorithm first considers plans where negated goals are assumed to be true, and then, as time permits, attempts to prove those negated goals true. When the proofs fail, new plans are generated to take the negated goals into account. Plans later in the progression have demonstratably more goals and subgoals that have been proven sound, and hence are more likely to be valid plans.

This algorithm is able to find an initial plan in polynomial time. Because negative preconditions can be assumed to be true, only positive predicates need to be achieved. As backtracking is only necessary when a desired positive predicate gets deleted by a negative postcondition, a single depth-first search will succeed in finding the (not necessarily valid) initial plan.

The assumption upon which the anytime progression for replanning using local subplan replacement is fundamentally different in that each plan in the progression is assumed to be a valid plan. The anytime progression for Elkan's algorithm is to gain increased certitude of plan validity by progressively checking negative preconditions.

The Just-In-Case Scheduler of Drummond et al. [29] constructs a schedule, analyzes it for the parts that are most likely to fail, and adds alternatives to those points creating what they call a "multiply contingent" schedule. The anytime progression for this algorithm is the incremental addition of these contingencies as time permits. The Just-in-Case scheduler is specialized for scheduling domains, and hence is not applicable to the more general problem of task planning.

Dean et al. [24] discuss an approach to planning under time constraints in stochastic domains. This approach consists of modeling a subset of the agent's state space (referred to as the *envelope*) and then generating a mapping of states to actions the agent should take in each state (referred to as the *policy*). Transitions between states are modeled probabilistically. That is, the agent performing some action in a given state has a set of probabilities indicating the chance of a transition to each potential successor state occurring. The construction of the envelope and policy is an iterative process. They are initially generated very quickly (or provided *a priori*). Since the envelope only contains a small subset of the total state space of the agent, the agent may enter a state not included in the envelope. In that case, the envelope gets extended and a new policy gets generated to deal with the extension to the envelope. These extensions of the envelope constitute the anytime progression for this algorithm.

The envelope is initially generated by performing a depth-first search through the search space, starting at the initial state and attempting to reach the goal state. Transitions from each state are considered in decreasing order of probability. A policy is then generated using an iterative refinement algorithm. Because each stage of this algorithm produces a strictly improving policy, once a first policy is available, it can be stopped at any time and return a useful policy as a result. The main drawback of this approach is that it requires the modeling of the environment and probabilistic transitions in significant detail. Because of this specialization, it is only applicable to planning problems in which each state transition has an associated probability. Using local subplan replacement for anytime replanning does not require that level of detail in environment modeling, and is applicable to domains where the cost function is something different than probability of success.

The Planning by Rewriting (PbR) algorithm of Ambite and Knoblock [3] assumes a fast algorithm is available for generating a valid plan for a particular problem, and then uses plan-rewriting rules to incrementally improve the quality of the quickly-generated plan. Plans are represented as partially-ordered sets of operators. There are four basic types of transformations that are enabled by the rules:

- A *reorder* rule rearranges operators based on their algebraic properties, such as commutative, associative, and distributive laws.
- A *collapse* rule replaces a subplan with a smaller subplan.
- An *expand* rule replaces a subplan with a larger subplan.

- A *parallelize* rule replaces a subplan with an alternative subplan that requires fewer ordering constraints. This can be very useful in domains where multiple agents can execute different parts of a plan.

These transformations are used to explore the neighborhood around the current plan. Gradient descent techniques are used to explore the neighborhood. These gradient descent techniques serve as the anytime progression for this algorithm. Random walks and restarts are used in order to escape the local minima in which the system can be trapped using gradient descent. Restarts rely upon the generation of alternative initial valid plans. Random walks rely on search spaces where a number of different plans can be found in a search plateau. The search can be interrupted at any time, and a valid plan will be returned, because the rules will only transform a valid plan to another valid plan.

Unlike many other anytime planning algorithms, PbR does not depend upon a particular cost function (such as probability) or a particular sort of problem (such as scheduling or path planning). PbR has the disadvantage that the user must provide a polynomial time algorithm for finding a valid plan, a set of rewriting rules for guiding the improvement search, and a search strategy. The authors have used machine learning techniques to help devise a set of rewriting rules [4]. These techniques, while moderately effective, required very large training sets of data, and the degree of their scalability remains an open question.

Unlike PbR, using local subplan replacement does not require any additional user-provided search control beyond what was necessary to generate the initial plan. Furthermore, the search in PbR is restricted to using local gradient descent techniques, and is not guaranteed to ever find an optimal plan. Local subplan replacement can, in theory, use a completely exhaustive global planner, and hence (given enough time) can escape local minima and provide a plan that is optimal for the given maximum depth. Although local subplan replacement will not likely be used with a totally exhaustive planner in practice, the fact that it can do so in principle allows for the possibility of using planning algorithms that provide good global approximations of the optimal plan.

The main drawback of using local subplan replacement compared with PbR is that there is no analogy to the parallel rewriting rule in PbR. This is because local subplan replacement is limited to improving a total-order plan, and the partial-order plan representation used by PbR permits the discovery of optimizations of this kind.

A number of anytime planning algorithms share an anytime progression in which the plans found early in the anytime progression establish only the most important goals, with later solutions in the progression achieving goals of lesser importance. Local subplan replacement does not prioritize goals; all goals are equally important, and it ensures that they are all achieved at all times. These approaches and local subplan replacement are targeted at fundamentally different types of domains. When finding a valid plan is difficult, prioritizing goals is a useful way to ensure that something of value gets accomplished.

Blythe and Reilly [10] describe modifications to Prodigy [59] to produce an anytime planner. Their variation keeps track of the best state seen so far in the state-space component of Prodigy's bi-directional plan search. The best state is defined in terms of a utility function that assesses the worth of the top-level goals achieved in that state. As the plan corresponding to this best state is returned if the planner gets interrupted, the sequence of "best states" defines the algorithm's anytime progression. A similar system using decision-theoretic planning is described by Haddawy [37].

Briggs and Cook [14] describe an anytime planning system that works by using a hierarchical planner that attempts to achieve the most "critical" goals first, and as time permits attempts to achieve less critical goals. If the planner exhausts its fixed upper bound node limit in the search for a solution, it uses a rule-based system to reactively complete the plan. As the authors were not concerned about finding valid plans (in the sense used in this dissertation, of plans that achieve all specified goals), no guarantees about plan validity were made or asserted, and the authors alluded to situations in which the system would not produce valid plans. Like Planning by Rewriting [3], this algorithm is also independent of both cost function and domain.

Boddy and Dean [11] [12] and Zilberstein and Russell [63] [64] both describe systems for integrating multiple anytime algorithms for the control of a mobile robot. Both systems determine allocations of available computational resources to the different anytime algorithms based on their *performance profiles*. The performance profile of an anytime algorithm shows the expected quality of the output of the algorithm across a range of levels of available computational resources.

Boddy and Dean [11] present a system that uses two complementary anytime planning algorithms for a robot delivery task. The first algorithm determines the sequence of deliveries to be made; the second algorithm does path planning between delivery destinations. The first algorithm solves a Traveling Salesman Problem. An initial tour is found using a polynomial time approximation algorithm. The anytime progression improves the initial tour using an edge-exchange algorithm.

The path planner uses an A* heuristic to find a path through a grid world environment. Even a partial path is considered a useful result, as further path planning can be performed while the partial path is being executed, and there are no "dead-end states" in the environment they use. For the path planner, then, the anytime progression is simply the degree to which the path plan has been completed.

Zilberstein and Russell [63] [64] discuss using multiple anytime algorithms for controlling a mobile robot. One anytime algorithm is used for the robot's sensing capabilities and another anytime algorithm is used for the robot's path planning. In their implementation, the robot and its environment are simulated, so the tradeoff between the time allocated to the sensing algorithm and the quality of the results of sensing are artificial.

The anytime progression for the sensing algorithm is that its accuracy in identifying obstacles and free space is proportionate to the running time it has available. The path planning algorithm initially generates a very abstract plan very quickly. Its anytime progression is that it progressively refines the level of detail of the path plan by selecting the current “worst” plan segment, dividing it into two smaller segments, and path planning each segment at a lower level of abstraction.

For both Boddy and Dean [12] and Zilberstein and Russell [64], the focus of the research is on integrating the anytime algorithms, not on the algorithms themselves. In contrast, my work is focused on producing an anytime strategy for plan repair for task planning that is not tied to the constraints of any particular domain. It is possible to produce a performance profile for anytime replanning using local subplan replacement so that it could be integrated with other anytime algorithms in a framework along the lines described by [12] and [64]. This possibility is discussed further in Section 5.8.

3.5 Heuristic Search

The Joint and LPA* algorithm of Ratner and Pohl [52] is a polynomial time heuristic search algorithm that relies on an idea akin to subplan replacement. Although their particular application was finding solutions for the 15-puzzle, their algorithm is applicable to any heuristic search problem with an admissible A* heuristic. An initial non-optimal path is generated using an approximation algorithm, and this path gets improved by replacing sections of constant bounded length with the result of an A* search for an optimal replacement. The length bound is set before execution begins and does not depend upon the problem description. The presence of the length bound is what enables the algorithm to complete in polynomial time.

The algorithm proceeds by incrementally attempting to replace subpaths at the start of the path until a shrinkage in length is obtained or the replacement length has reached the maximum. From that point, the remainder of the path is broken into segments of the maximum length, and each segment gets replaced by the result of the A* search. The algorithm completes after one pass through the path. The Joint algorithm then repeats this process, but it centers the replacements on the joints between the replacements from the first part of the algorithm, in an attempt to optimize the "joints" between optimal paths.

These algorithms rely heavily on the presence of an admissible A* heuristic to enable them to prune aggressively so that the length limit can be set relatively high. Local subplan replacement as described in Chapter 4 does not rely on the particular exhaustive search strategy employed. In the presence of a strategy that prunes effectively, local subplan replacement will be able to perform subplan replacements at low levels very quickly, thus enabling it to progress to higher subplan levels promptly. With more exhaustive searches, the low initial limit on search depth provided at low subplan levels should enable the system to find at least some improvements relatively quickly.

Thus, the incremental increasing of the subplan level by local subplan replacement enables the improvement algorithm to adjust to both the quality of the underlying search and the problem-instance-specific time available without any outside interference. The Joint phase of their algorithm is simulated to some extent within subplan replacement by the fact that the replacements with larger maximum search depths will divide the plan into components that include the joints from previous iterations.

Another important distinction is that the end state for the operator subsequence to be replaced in subplan replacement is not a complete world state; rather, it consists only of those state elements necessary as preconditions for future operators and for the establishment of system goals. This enables the consideration of a potentially larger number of new subplans. The Joint and LPA* algorithm does rely upon a complete world state description as its replacement target.

3.6 Summary

In this chapter, I gave an overview of existing planning systems. I described systems that are domain-independent and I also described systems that make use of domain-specific information in order to control and limit the amount of search they conduct. I gave several examples of existing plan repair systems, none of which implement anytime planning in such a way as to guarantee always having available a valid plan. I described existing anytime planning systems, most of which also do not guarantee always having available a valid plan. One system that does make this guarantee, Planning by Rewriting [3], only does so under the assumption that a different system has provided an initial valid plan quickly. Planning by Rewriting also lacks the potential to exhaustively explore a search space given enough time. Local subplan replacement is a plan repair system that implements anytime planning in such a way as to guarantee always having available a valid plan, including the rapid generation of the first plan in the anytime progression. It also can be configured to explore as much of the search space as desired, given enough time.

4 Local Subplan Replacement

In this chapter, I describe the components of the anytime planning system I am evaluating. Section 4.1 briefly describes the components and how they interact. The planning algorithms used by this system are described in Section 4.2. (These algorithms are also used as a baseline for evaluating the system in Chapter 5.) Generating initial valid plans is described in Section 4.3. Progressively improving these plans in an anytime manner using local subplan replacement is described in Section 4.4. It is this technique for progressive plan improvement that is the principal subject of evaluation in this dissertation.

4.1 System Description and Motivation

Anytime planning using local subplan replacement proceeds as follows. Given an original plan that has succumbed to a failure, it begins by quickly repairing the failed plan. Then, it iteratively replaces subsections of the repaired plan in order to improve its quality. To establish a progression necessary for the desired anytime characteristics, the process begins by attempting to improve relatively small subplans, then progresses to consider longer and longer subplans.

In order to detect a plan failure, during plan execution the preconditions of each operator are checked before it is executed. If any of these preconditions is false, then a plan failure has occurred and it becomes necessary to replan.

Once the decision is made to replan, the anytime progression begins by using a plan repair algorithm to generate a valid plan. This plan repair algorithm inserts operators into the plan in order to restore operator preconditions that have become false. It may also insert operators to achieve goals that are no longer being achieved by other operators. It begins by adding operators that will establish the preconditions for the first operator from the original plan that has not yet been executed. Once the first operator has been established, the algorithm inserts operators to establish the preconditions for the second operator. It proceeds forward one operator at a time. If, when preconditions have been established for all operators, there are goals that are still not established, the algorithm inserts operators to achieve each such goal. When the plan repair algorithm is finished, the result is a valid repaired plan. The plan repair algorithm is designed to execute quickly yet yield a valid plan.

Although the repaired plan is valid, it is not likely to be a high-quality plan. The repair process retains a substantial proportion of the operators from the original plan. This research investigates whether these retained operators can be effectively utilized in constructing a high-quality plan. The search process that generated the original plan considered many possible alternatives. The plan found by the original search process was deemed superior to these alternatives. The anytime replanning process will attempt to take advantage of these previously made good decisions.

The means by which the anytime replanning process utilizes material from the original plan is to replace parts of the repaired plan while other operators from the original plan remain. Subplan replacement begins by determining the operator subsequence to be replaced. Once that has been determined, a planner is invoked in order to consider alternatives to the existing subplan. If the planner finds a superior subplan, it will replace the existing subplan.

The locations within the repaired plan where subplan replacements will occur are determined as follows. As mentioned above, during the plan repair process sequences of operators are inserted in order to establish failed preconditions. Subplan replacements are centered at the beginning and end of each of these sequences of inserted operators. These subplan replacement locations are called *anchors*. Centering replacements on these anchors enables the systematic reconsideration of the operators inserted by the plan repair algorithm.

The size of the subplan to be replaced is determined by the system-declared *subplan level*. Each level indicates the fraction of the total number of operators in the plan that gets replaced. When anytime replanning begins, the total number of subplan levels N is specified. Replanning then begins at level 1. At each level L, the size of the subplan replaced around each anchor is $\frac{L}{N}R$, where R is the number of operators in the repaired plan.

The search for each new subplan is limited in depth to the length of the original subplan. At low subplan levels, the depth limit is low, resulting in replacements that can be performed relatively quickly. As the subplan level increases, larger subplans are considered for replacement. The searches at higher subplan levels take more time than those at lower levels, but there is potential for greater improvement due to the larger number of alternative plans considered. Increasing the subplan level is the mechanism by which subplan replacement is used to implement anytime planning.

Both local subplan replacement and plan repair require an underlying planning algorithm in order to add and replace operators. The planner I used for this work, `Find-plan`, is of my own design. It is closely related to total-order, hierarchical task network planners such as SHOP [47] [48] (discussed in Section 3.2.1). It was designed for the purpose of solving planning problems in domains in which valid plans can be found in polynomial time but finding optimal plans is NP-Complete. It performs depth-first forward-chaining to find plans. The depth-first searches are guided by a rule base that indicates what operators to apply in order to achieve certain predicates. The user can specify a maximum number of depth-first searches, a depth limit, and a maximum number of node expansions. The user can also specify two different general search strategies. The first strategy is similar to a traditional depth-first search. The second strategy permutes the ordering of choices in each search in an attempt to "sample" the search space more broadly than the first strategy, given the presence of limits on the number of depth-first searches and nodes.

`Find-plan` is described in detail in Section 4.2. Plan repair is the subject of Section 4.3. Local subplan replacement is described in detail in Section 4.4. The overall integrated anytime planning system including the incremental increase of the subplan level and the selection of anchors is described in Section 4.5.

4.2 Finding Plans

The `Find-plan` algorithm is used both by local subplan replacement and as a baseline for evaluating the effectiveness of local subplan replacement. `Find-plan` uses depth-first searches in order to find plans. In each depth-first search, operator sequences are found for the achievement of one goal at a time. For each goal, a rule set indicates the sequence of subgoals to solve. The rule set also specifies which operators to use in order to solve "primitive" goals and subgoals.

Given a rule base that can guide any depth-first search to a valid plan (provided such a plan exists), depth-first search can be used to implement anytime planning.

The initial solution is the plan found by any single depth-first search, and the anytime progression consists of performing additional depth-first searches in the hope of finding a better plan. Because of this property, the algorithm described in this section can be used as a baseline for evaluating the performance of local subplan replacement, in addition to its use by the plan repair algorithm and local subplan replacement. This is discussed in more detail in Chapter 5.

`Find-plan` constructs a directed acyclic graph containing a node representing each top-level plan goal. Each edge in the graph indicates that the goal corresponding to the originating node has higher precedence than the goal corresponding to the destination node. Each depth-first search follows these precedence relationships when constructing a goal ordering. Goals with no precedence relationship between them are ordered nondeterministically.

This algorithm is very similar to the SHOP algorithm [47] [48]. (The SHOP algorithm was described in Section 3.2.1.) It does not have all of the features of SHOP, but it is easier to configure for my purposes, especially regarding the specification of search limits and search strategies.

4.2.1 Predicate Rules

A rule exists for each predicate in the domain description that can be made true by the application of plan operators. A rule is itself a first-order predicate. If this first-order formula evaluates to `false`, then no plan can be found.

Figure 4.1 gives an example of a simple rule. Each rule is represented in prefix notation. The rule attempts to make the predicate `(robot-at ?machine)` true. If it is already true, the `check` predicate returns true. If it is false, then the rule binds `?robot-l` to the robot's current location and then adds the `travel` operator to the current plan. The rule has no priority over any other rule, so the `priority-over` list is empty.

The rule set contains standard logical operators (`and`, `or`, and `not`), quantifiers (`exists` and `forall`), and several special forms (`check`, `achieve`, `choose`, `goal`, and `default`). It also contains primitive predicate symbols denoting an aspect of the world state. For example, in Figure 4.1, `robot-at` is a primitive predicate relaying information about the world state.

```
((robot-at ?machine)
 (priority-over)
 (or (check (robot-at ?machine))
     (exists (?robot-l) (robot-at ?robot-l)
        (achieve travel ?robot-l ?machine))))
```

Figure 4.1: Rule for Moving a Robot

A rule can be evaluated in two ways. First, it can be evaluated solely in terms of assessing its truth value. In that case, the `check` and `choose` symbols simply return the truth values of their arguments, and `achieve` and `default` return false. The `goal` symbol returns true if its argument would be true in the goal state.

Second, it can be evaluated to generate plan operators to achieve the specified predicate. It will return true if it succeeds and false otherwise. If it succeeds, it will also have constructed the plan it found as a side effect. Evaluation of a primitive predicate triggers the evaluation of its corresponding rule. For example, in the rule shown in Figure 4.2, evaluation of `(robot-at ?store)` causes the `robot-at` rule given in Figure 4.1 to be applied.

The `achieve` symbol adds the specified operator with the specified arguments to the plan, provided that the preconditions of the operator are true in the current state. If it can add the operator it returns true, otherwise it returns false. The `default` symbol returns true. The `check` symbol evaluates its argument solely in terms of assessing its truth value. The logical operators and quantifiers return true or false according to their traditional definitions. With conjunctions (i.e. `and` and `forall`), if any argument or instantiation fails no others will be tried. With disjunctions (i.e. `or` and `exists`), once an argument or instantiation returns success, no others will be tried.

The `choose` symbol is used in order to introduce nondeterminism into the planning algorithm encoded by the rules. The argument to `choose` can be any logical conjunction or disjunction. The planning algorithm will consider all possible orderings of the arguments or instantiations of that logical conjunction or disjunction. Each ordering constructs a distinct plan (if it finds one). The plan with the best quality (according to the metric specified for the domain) is retained by `choose`.

Figure 4.2 shows how `choose` can be used. This rule guides the robot towards obtaining a needed part. It has priority over any goal dictating that the robot be at a particular location, as in order to obtain a part the robot may have to move. The `choose` symbol tells the planning algorithm to consider retrieving a part of the appropriate type from every possible storehouse. The `try-first` symbol tells the planning algorithm to begin by trying to obtain a part from a storehouse at which the robot is currently located.

Each rule set has a depth limit for its depth-first searches. If a depth-first search attempts to generate a plan that exceeds this length, failure is returned. This depth limit can be the same for all depth-first searches in a given domain, or it can depend upon the predicate and numerical values of the initial world state. For example, for

```
((robot-slot-full ?slot ?type)
 (priority-over (robot-at ?a))
 (or (check (robot-slot-full ?slot ?type))
     (choose (exists (?store) (is-storehouse ?store)
        (try-first (robot-at ?store))
        (exists (?store-slot)
           (machine-part-slot-type ?store ?store-slot ?type)
           (and (check (machine-slot-full ?store ?store-slot))
                (robot-at ?store)
                (achieve get-part-from ?store ?type ?slot
                                       ?store-slot)))))))
```

Figure 4.2: Rule for Obtaining a Machine Part

the Repair Robot domain, the depth limit is $2M + 6P$, where M is the number of machines and P is the total number of machine part slots. This depth limit reflects the fact that a simple plan for solving a Repair Robot problem can require up to six operators for installing each part[1] and two operators for activating each machine.[2]

Figure 4.3 gives the top-level algorithm for applying a rule. Each rule is named for the predicate it seeks to achieve. If that predicate is already true, there is no reason to apply the rule, and `Apply-rule` returns true. Each rule is itself a predicate. If it is a primitive predicate, it simply calls itself recursively for the rule for that primitive predicate. If it is the default predicate, it returns true. Otherwise, all special predicate symbols have their own parse rule for evaluating them in rule expressions. The appropriate parser is then called and its return value is returned.

Figure 4.4 shows how to parse the `achieve` predicate. This is at the heart of the planning algorithm because it is the only place where operators are added to the plan. Before adding an operator, it ensures that the operator's preconditions are true, that none of its delete effects eliminates a higher-priority goal, and that the plan depth limit has not been exceeded. It then adds the operator to the plan and updates the world state with all of the add and delete effects of the operator. It then returns true to indicate that it succeeded in adding an operator.

Figures 4.5 and 4.6 show how the `and` and `or` symbols are parsed. `Apply-rule` is called for each of their arguments. In the case of `and`, plans must be found for all of the arguments. For `or`, a plan need only be found for one of the arguments.

Figures 4.9 and 4.10 show how the quantifiers `forall` and `exists` are parsed. The syntax is: `(quantifier (?free-var) (condition ?free-var) (body))`.

[1](1) Travel to a storehouse, (2) unload a part currently held, (3) travel to another storehouse, (4) grab a part, (5) travel to a machine, (6) install the part

[2](1) Travel to the machine, (2) activate it

```
Apply-rule(predicate, currentState, currentPlan, goalState)
  If predicate is true in currentState, return true
  Let rule = find-rule(predicate)
  Let rule-pred be the top-level predicate in rule

  If rule-pred is a primitive predicate:
    Return Apply-rule(rule-pred, currentState, currentPlan,
                      goalState)

  If rule-pred is default, return true

  Otherwise, call the parser for the appropriate non-primitive
  predicate and return its return value.
  Pass rule-pred, currentState, currentPlan, goalState.
  For parse-and, parse-or, parse-exists, and parse-forall,
    pass false for permute
```

Figure 4.3: Apply-rule

```
Parse-achieve(rule-pred, currentState, currentPlan, goalState)
  Let operator be the instantiated plan operator specified by achieve
  If any of these are true:
    Any precondition of operator is false in currentState
    Any delete effect of operator deletes any true goal that is
      higher priority than rule-pred
    Length of currentPlan exceeds depth limit
  Then return false

  Else:
    Add operator to currentPlan
    Update currentState with the effects of operator
    Return true
```

Figure 4.4: Parse-achieve

```
Parse-and(rule-pred, currentState, currentPlan, goalState)
  Let arg-list be a list of all arguments of rule-pred
  Return Apply-conjunction-list(arg-list, currentState,
                                currentPlan, goalState)
```

Figure 4.5: Parse-and

```
Parse-or(rule-pred, currentState, currentPlan, goalState)
  Let arg-list be a list of all arguments of rule-pred
  Return Apply-disjunction-list(arg-list, currentState,
                                currentPlan, goalState)
```

Figure 4.6: Parse-or

```
Apply-conjunction-list(list, currentState, currentPlan, goalState)
  For each predicate in list:
    If Apply-rule(predicate, currentState, currentPlan, goalState)
       is false, return false
  Return true
```

Figure 4.7: Apply-conjunction-list

```
Apply-disjunction-list(list, currentState, currentPlan, goalState)
  For each predicate in list:
    Let planCopy = currentPlan
    Let stateCopy = currentState
    If Apply-rule(predicate, stateCopy, planCopy, goalState) is true
      currentPlan = planCopy
      Return true
  Return false
```

Figure 4.8: Apply-disjunction-list

```
Parse-forall(rule-pred, currentState, currentPlan, goalState)
  Let binding-list = Get-binding-list(get-free-variable(rule-pred),
                                      currentState,
                                      get-position(rule-pred))
  Return Apply-conjunction-list(binding-list, currentState,
                                currentPlan, goalState)
```

Figure 4.9: Parse-forall

```
Parse-exists(rule-pred, currentState, currentPlan, goalState)
  Let binding-list = Get-binding-list(get-free-variable(rule-pred),
                                      currentState,
                                      get-position(rule-pred))
  Return Apply-disjunction-list(binding-list, currentState,
                                currentPlan, goalState)
```

Figure 4.10: Parse-exists

```
Get-binding-list(freeVariable, currentState, condition)
  Let list be an empty list
    For each binding of freeVariable in currentState
      If condition(binding) is true, add binding to list
  Return list
```

Figure 4.11: Get-binding-list

In both cases, all instantiations of the free variable of the quantifier that are true given its `condition` are found. In the case of `forall`, a plan must be found that makes `body` true for all possible instantiations; for `exists`, a plan must be found for any one instantiation. For example, in the quantifier rule `(exists (?store) (is-storehouse ?store) (robot-at ?store))`, a plan must be found to move the robot to any one storehouse. Were the quantifier `forall`, the robot would have to visit all of the storehouses.

```
Parse-choose(rule-pred, currentState, currentPlan, goalState)
  Let choose-pred be the single argument of rule-pred
    (choose-pred must be or, exists)
  If choose-pred is or: arg-list = arguments(choose-pred)
  If choose-pred is exists:
    arg-list = Get-binding-list(get-free-variable(choose-pred),
                                currentState,
                                get-position(choose-pred))

  Return Search-disjunction(arg-list, currentState, currentPlan,
                            goalState)
```

Figure 4.12: Parse-choose

The `choose` symbol is handled as shown in Figure 4.12. The code for this symbol calls `Search-disjunction` to perform a separate depth-first search for each instantiation of `exists` or each argument of `or`. `Search-disjunction` calls `Find-best-plan`, described below in Figure 4.18, in order to solve the next goal once it has added

```
Search-disjunction(list, currentState, currentPlan, goalState)
  Let bestPlan be an empty plan
  For each predicate in list
    Let planCopy = currentPlan
    Let stateCopy = currentState
    If Apply-rule(predicate, stateCopy, planCopy, goalState) is true
      Find-best-plan(planCopy, stateCopy, goalState, goalList,
                     goalGraph)
      If cost of planCopy is better than the cost of bestPlan
      bestPlan = planCopy
  If bestPlan is empty
     Return false
  Return true
```

Figure 4.13: Search-disjunction

operators to achieve `predicate`.

```
Parse-check(rule-pred, currentState, currentPlan, goalState)
  If rule-pred's argument is true in currentState, return true
  Else return false
  Note: achieve and default are considered to always be false
    when checked by check
```

Figure 4.14: Parse-check

Figure 4.14 describes how the `check` symbol is parsed. This symbol is used when it is desired for a predicate to simply have its truth value checked against the current state without invoking a rule for any of the predicates.

```
Parse-goal(rule-pred, currentState, currentPlan, goalState)
  If argument of rule-pred is true in goalState, return true
  Else return false
```

Figure 4.15: Parse-goal

Figure 4.15 shows how the goal symbol is parsed. The goal symbol indicates whether its argument is true in the goal state. It is used in practice in order to determine whether the specified predicate argument is a goal. No plan actions will be added to achieve this; it is purely informational.

4.2.2 Finding a Plan

The `Find-plan` algorithm (shown in Figure 4.16) begins by creating a directed acyclic graph that determines a partial ordering of the goals based on their priorities. It then creates a goal list containing all the goals from the graph with no incoming edges. It also creates an empty plan.

At this point, `Find-best-plan` (see Figure 4.18) is called. Each goal in the goal list is selected as the starting point of a depth-first search. The search commences for each goal by calling `Find-plan-for-goal` (Figure 4.19), which calls `Apply-rule` (Figure 4.3). When `Apply-rule` has completed, the newly achieved goal is removed from the goal graph and all nodes that have no incoming edges that were not already in the goal list are added to the goal list. The depth-first search continues by starting a depth-first search for each goal as before. Each depth-first search completes when the goal list is empty or a call to `Apply-rule` returns false.

```
Find-plan(initialState, goalState)
  Let goalGraph = make-goal-graph(goalState)
  Let goalList be a list of all nodes in goalGraph with no
    incoming edges
  Let plan be an empty plan
  Let best-plan be an empty plan
  Let currentState be a copy of initialState
  Return Find-best-plan(plan, currentState, goalState, goalList,
                        goalGraph)
```

Figure 4.16: Pseudocode for `Find-plan`

```
Make-goal-graph(goalState)
  Let goalGraph be an empty graph
  For each goal g in goalState
    Create a node for g in goalGraph
  For each goal g in goalState
    For every other goal h in goalState
      If the rule for g indicates priority over h
        Add a directed edge from g to h
  Return goalGraph
```

Figure 4.17: Pseudocode for `Make-goal-graph`

4.2.3 Search Control

The amount of search performed by `Find-plan` can be limited by setting maximum values for the total number of depth-first searches and the total number of world state expansions.

If these maximum values limit the search to a small subset of the search space, it is possible for the search to be concentrated in too narrow an area of the search space. A particular danger is for most of the depth-first searches attempted within the maximum to have in common all but a few operators towards the end of the plan.

In order to address this problem, when either maximum is set, the depth-first searches can be divided into equally-sized groups. In each group, a different goal from the goal list is the first goal to be solved. This way, each goal gets a chance to be the first goal solved. Subsequent goal selections and `choose` points are also offset

```
Find-best-plan(plan, currentState, goalState, goalList, goalGraph)
  If goalList is empty return true
  Let bestPlan be an empty plan
  While goalList is not empty
    For each goal in goalList
      Let plan-copy  = plan
      Let state-copy = currentState
      Let graph-copy = goalGraph
      Let list-copy  = goalList
      Remove goal from list-copy
        Let success =
          Find-plan-for-goal(goal, plan-copy, state-copy, goalState,
                             list-copy, graph-copy)
      If success is true and bestPlan is empty or
        the cost of plan-copy is better than the cost of bestPlan
        bestPlan = plan-copy
  Return bestPlan
```

Figure 4.18: Pseudocode for `Find-best-plan`

```
Find-plan-for-goal(goal, plan, currentState, goalState, goalList,
                   goalGraph)
  Let success = Apply-rule(goal, currentState, plan, goalState)
  If success is false, return false
  Delete goal from goalGraph
  Add to goalList all goals in goalGraph with no incoming edges
    that are not already in goalList
  Let success =
    Find-best-plan(plan, currentState, goalState, goalList, goalGraph)
  Return success
```

Figure 4.19: Pseudocode for `Find-plan-for-goal`

differently in each group. This helps make the search more heterogeneous.

4.2.4 Example Rule Set

A full formal description of the rule set used for the Repair Robot domain can be found in Appendix A. This section gives a brief overview of the algorithm it encodes.

The top-level system goals are for each machine to be operational. For each machine, the algorithm checks to see if all of its parts are in place. If any are not, then the robot travels to a storehouse to retrieve all the parts the robot needs. It may need to travel to multiple storehouses in order to do this. The storehouse to visit is chosen nondeterministically.

Once the robot has reached a storehouse and retrieved all parts from that storehouse that could help in repairing the current machine, it nondeterministically chooses whether to fill each empty robot slot with a part for another damaged machine. It makes a nondeterministic choice for every empty machine part slot belonging to another broken machine. It may choose to retrieve the part or ignore that machine entirely.

At this point, the robot returns to the machine it was originally trying to repair, and replaces as many broken parts as possible. It continues to retrieve parts for the machine and install them until all broken parts have been replaced. At that point, the robot activates the machine, enabling it to resume production.

4.3 Valid Plan Repair in Polynomial Time

In this section, I present an algorithm for repairing failed plans. Because this algorithm returns the first plan in the anytime progression, it is important that it execute quickly. In general, repairing a failed plan can be as computationally complex as generating a new plan from scratch [49]. Hence, this plan repair algorithm requires, as described in Section 2.3.2, that a valid plan can be found in polynomial time.

The objective of the plan repair algorithm (see Figure 4.20 and Figure 4.21 for pseudocode) is to transform the failed plan into a valid plan without regard to the cost of the resulting plan. It accomplishes this by examining each of the operators in the failed plan (starting with the first operator) and determining if any of its preconditions are false. If so it invokes `Find-plan` in order to generate a plan with the failed operator's preconditions as its goal. If it is unable to find a plan to meet the failed operator's preconditions, it deletes the failed operator entirely and moves to the next operator. If the plan repair algorithm has checked all the plan operators and ensured that their preconditions are true, and there remain goal conditions that are not true, it calls `Find-plan` one more time to add operators to the end of the plan that will achieve the outstanding unachieved goals. It saves the locations of the inserted operators in the array `newOpNums` in order to enable local subplan replacement to select anchor points.

The parameters of `Repair-plan` are:

- `currentState`: The world state following the plan failure.
- `goalSet`: The top-level plan goals.
- `failedPlan`: A plan containing all the operators of the original plan that have not yet been applied.

4.3.1 Correctness

The algorithm `Repair-plan` is correct if it always returns a valid plan (as defined in Section 2.2.1). The key elements of a valid plan are:

```
Repair-plan(currentState, goalSet, failedPlan)

  Let repairedPlan be a plan with no operators
  Let newOpNums be an empty list

  For each operator currentOp in failedPlan
    Repair-operator(currentState, currentOp, repairedPlan)

  If any goals are false in currentState
    Let planTail = Find-plan(currentState, goalSet)
    If a plan is found
      For each operator tailOp in planTail
        Add tailOp to repairedPlan
          Add the position of tailOp in repairedPlan to newOpNums
        Apply tailOp to currentState
    Else
      Return failure

  Return repairedPlan and newOpNums
```

Figure 4.20: Plan Repair

```
Repair-operator(currentState, currentOp, repairedPlan)

  If all preconditions of currentOp are true
    Add currentOp to repairedPlan
    Update currentState with effects of currentOp
  Else
    Let patchGoals = all preconditions for currentOp
    Let patchPlan = Find-plan(currentState, patchGoals)
    If a plan is found
      For each operator patchOp in patchPlan
        Add patchOp to repairedPlan
        Add the position of patchOp in repairedPlan to newOpNums
        Apply patchOp to currentState
      Add currentOp to repairedPlan
      Apply currentOp to currentState
    Else
      Return
```

Figure 4.21: Plan Operator Repair

1. For each operator, all of its preconditions are true in the state in which the operator is to be applied.

2. After all operators have been applied, all goal predicates are true in the resulting state.

For a plan to be invalid, there need only exist one false precondition for one operator, or one false goal predicate.

For the purposes of demonstrating the correctness of the above algorithm, I make the following assumptions:

1. `Find-plan` produces a valid plan if a valid plan exists

2. `Find-plan` will not produce an invalid plan

3. It is not possible for an operator to transition to a dead-end state (defined in Section 2.3.5). This assumption may hold true either because the operator preconditions disallow the application of the operator to a state in which its postconditions could lead to a dead-end state, or because its postconditions are simply incapable of transforming any world state into a dead-end state.

The correctness of `Repair-plan` follows directly from the above assumptions. If `Repair-plan` does find a plan, each operator in the plan will have all of its preconditions true because:

- The operator was present in the original plan, and any false preconditions it had were handled by adding operators using `Find-plan`, or

- The operator was added by `Find-plan`, which produces valid plans.

It also follows that every goal will be established by the plan, because either the goal was established by operators present from the original plan, or `Find-plan` added operators to establish the goal.

4.3.2 Computational Complexity

Let o be the number of operators in the failed plan. Let p be the maximum number of preconditions of any operator (or the total number of goals if it is greater), and e be the maximum number of effects of any operator. Let $P(S, g)$ be the computational complexity of invoking the planner to find a plan $q(S, g)$ for g goals and initial state S, let $L(q(S, g))$ be the length of $q(S, g)$, and let $L_{max}(q(S, g))$ be the length of the longest plan $q(S, g)$ that $P(S, g)$ can generate. Let S be the number of elements in the initial world state (stored as a red-black tree). Let S_o be the number of elements in the world state following the application of operator o.

The maximum number of elements that can be contained in any world state is $S_{max} = S + oeL_{max}(q(g))$. This is derived from observing that for each operator in the failed plan, $L_{max}(q(g))$ additional operators may be inserted. Each plan operator can add up to e elements to a world state.

Determining if any preconditions are false for a single operator a is $O(p \log S_{max})$. For each precondition, we could have to do a red-black tree lookup ($O(\log S_{max})$). Finding a plan for the precondition set for that operator is $O(P(S_a, p))$. Adding any operator to the new plan and updating the state accordingly is $O(e \log S_{max})$.

We can assume that $P(S_{max}, p)$ is polynomial in S_{max} and p by setting a constant upper bound on the number of plans `Find-plan` is allowed to consider.

The overall complexity of the algorithm is:

$$O(o(p \log S_{max} + P(S_{max}, p) + L(p)e \log S_{max}))$$

4.4 Using Local Subplan Replacement to Improve Plans

The purpose of replacing subplans of a repaired plan is to attempt to improve selected operator subsequences while retaining the remainder of the existing plan, thus seeking to exploit some of the good decisions that were made in the original process of plan generation. Local subplan replacement consists of selecting subplans in the repaired plan and constructing new plans for connecting the precondition state of that subplan and its postcondition state. The plan resulting from this replacement is substituted for the current repaired plan in the event that it signifies a cost improvement.

The locations within the repaired plan where subplan replacements will occur is determined as follows. During the plan repair process (see Section 4.3) sequences of operators are inserted in order to establish failed preconditions. Subplan replacements are centered at the beginning and end of each of these sequences of inserted operators. These subplan replacement locations are called *anchors*. Centering replacements on these anchors enables the systematic reconsideration of the operators inserted by `Repair-plan`.

4.4.1 Replacing Subplans

Figure 4.22 contains pseudocode for local subplan replacement. The first parameter, `plan`, is the plan to be modified. The second parameter, `startState`, is the world state after the occurrence of the plan failure. The third parameter, `goalSet`, contains all the goals for `plan`. The fourth and fifth parameters, `subplanStartOp` and `subplanEndOp`, are the operator numbers of the first and last operators of the subplan to be considered for replacement.

`Replace-subplan` begins by computing `subStartState`, the initial state for the subplan. It does this by applying the postconditions of all operators that precede the subplan in `plan` to `startState`. Next, it generates `subEndState`, the state resulting from applying the operators in the current subplan. Then, it determines what predicates from `subEndState` are relevant to achieving goals and preconditions by calling `Find-subplan-goals` (described below in Section 4.4.2). Finally, it creates a new plan by replacing the subplan with the result of a call to `Find-plan`. If the resulting new plan is superior in cost to `plan`, the new plan is returned.

The `Op-before` and `Op-after` functions referred to in Figure 4.22 return the immediately preceding or immediately following operators from the plan, respectively.

The `Replace-subplan` algorithm has the following properties:

- Any plan returned will be a valid plan, given a valid plan as input.
- The plan returned will be equal to or lower in cost compared to the input plan.
- Its worst-case execution time is the execution time of `Find-plan` plus the execution time for `Find-subplan-goals`.

The property that the cost will be lower follows immediately from the final `if` statement.

Generating `subStartState` and `subEndState` in the worst case requires checking and changing each state predicate for every operator. This is bounded above by $O(ns)$, where n is the number of operators in the original plan and s is the maximum number of predicates in a world state. The computational complexity of `Find-subplan-goals` has a higher upper bound than $O(ns)$, dominating the complexity of generating `subStartState` and `subEndState`.

There are two possible cases to consider regarding whether `Replace-subplan` will always return a valid plan. In the first case, the subplan discovered is equal or greater

```
Replace-subplan(plan, startState, goalSet, subplanStartOp,
                subplanEndOp)

  let subStartState = Apply-subplan(plan, startState,
                                    plan.firstPlanOp,
                                    Op-before(subplanStartOp))
  let subEndState = Apply-subplan(plan, subStartState,
                                  subplanStartOp,
                                  subplanEndOp)
  let subGoalSet = Find-subplan-goals(plan, goalSet, subEndState,
                                      Op-after(subplanEndOp),
                                      plan.finalPlanOp)
  let newPlan = append(plan.get-subplan(plan.firstPlanOp,
                                        Op-before(subplanStartOp)),
                       Find-plan(subStartState, subGoalSet)
                       plan.get-subplan(Op-after(subplanEndOp),
                                        plan.finalPlanOp))

  if newPlan.plan-cost < plan.plan-cost
     return newPlan
  else
     return plan
```

Figure 4.22: Local Subplan Replacement

in cost compared to the subplan it seeks to replace. In that case, the parameter `plan`, already a valid plan, is the plan returned.

In the second case, the subplan discovered is inserted into `plan`. To determine whether a valid plan is returned, we must determine whether appending the three components results in a valid plan. The `Find-subplan-goals` function (see Section 4.4.2 for details) is guaranteed to return a list of all the goals the subplan must accomplish in order to meet the operator preconditions of the operators following the subplan as well as any goals achieved by the original subplan.

The parts of the original plan outside the subplan are unchanged; hence, any goals they achieve are still achieved, and any preconditions they achieve are still achieved. `Find-plan` returns a plan that achieves goals that suffice to compensate for all useful things done by the original subplan. Concatenating these three subplans together, then, results in a valid plan.

The `Replace-subplan` algorithm would still work if, instead of determining a goal set for the subplan based on the preconditions of the operators that follow, the state resulting from applying the original subplan was used as the goal set instead. Using the preconditions as goals instead of the complete resulting state allows a larger space of potential subplans to be considered.

4.4.2 Finding the Goal Set for a Subplan

Figure 4.23 contains pseudocode for an algorithm to determine the goal set for a subplan. The first parameter, `plan`, is the plan for which a subplan is being replaced. The second parameter, `goalSet`, contains all the top-level goals for the plan. The third parameter, `postSubplanState`, is the world state description after the application of the postconditions of the subplan that is to be replaced. The fourth and fifth parameters, `startOp` and `endOp`, are the operator numbers of the first and last operators of the subplan to be replaced.

All predicates in `postSubplanState` might be goals for the subplan. Some of them, however, may be extraneous, in that they might not establish any goals or preconditions of operators in the remainder of the plan. `Find-subplan-goals` seeks to eliminate these extraneous predicates while preserving all of the essential ones.

For each operator, the call to `Update-subplan-goals` adds to `subplanGoals` any predicates from `subplanGoalProspects` that achieve a precondition of the current operator. The final call to `Update-subplan-goals` after the loop does the same thing for any remaining top-level goals. `Update-subplan-goals` is described below in Section 4.4.3.

The `Find-subplan-goals` algorithm has the following properties:

- If all predicates returned in `subplanGoals` are true prior to the application of `startOp`, then all preconditions for operators starting at `startOp` will be established before the application of each relevant operator.
- When all operators have been applied, all goals will be true.
- `subplanGoals` does not contain any predicates that do not either establish preconditions of operators or goals.
- Worst-case execution time is $O(n(e \log s + s(P_c p + \log s)))$, where n is the number of operators, e is the maximum number of effects for a precondition, s is the number of states, p is the maximum number of preconditions for an operator, and P_c is the worst-case complexity for determining if a predicate satisfies a precondition.

To determine the worst-case execution time, we multiply the sum of the cost of `apply-operator` and `update-subplan-goals` by the number of operators in the plan.

```
Find-subplan-goals(plan, goalSet, postSubplanState, startOp, endOp)

  let subplanGoalProspects = postSubplanState
  let currentState = postSubplanState
  let subplanGoals = {empty set}

  for op = startOp to endOp
    update-subplan-goals(currentState, subplanGoals,
                         subplanGoalProspects, preconditions(op))
    let currentState = apply-operator(op, currentState)

  update-subplan-goals(currentState, subplanGoals,
                       subplanGoalProspects, goalSet)

  return subplanGoals
```

Figure 4.23: Finding Subplan Goals

Applying an operator requires $O(\log s)$ computation for each effect of the operator, as it requires updating a red-black tree (which is how the world state is stored). The worst-case execution time for updating the subplan goals is described in Section 4.4.3.

`SubplanGoalProspects` contains all of the predicates established by the original subplan. Each predicate must be classified as either establishing a precondition or goal, or superfluous. Demonstrating the properties above requires demonstrating that `Find-subplan-goals` properly classifies all predicates from `subplanGoalProspects`. Hence we must demonstrate that all predicates that are not superfluous have been added to `subplanGoals` by the time `Find-subplan-goals` terminates. We must also demonstrate that no superfluous predicates get added to `subplanGoals`.

We will demonstrate these properties by induction. In the base case, there are no operators that follow, the loop does not execute, and no predicates get added to `subplanGoals`. At that point, `Update-subplan-goals` is called and all goals present in `currentState` get added to `subplanGoals` (as described and demonstrated in Section 4.4.3). No superfluous predicates have been added, as `Update-subplan-goals` will only add predicates from `goalSet`.

In the inductive step, we assume that the preconditions for all operators already examined have been added to `subplanGoals`. `Update-subplan-goals` has two effects (as described and demonstrated in Section 4.4.3). First, all preconditions for the current operator present in `subplanGoalProspects` get added to `subplanGoals`. Second, all predicates in `subplanGoalProspects` that are no longer true in `currentState` get removed from `subplanGoalProspects`. Any precondition that has been deleted cannot be expected to satisfy a precondition for a future operator; it will have to be added subsequently by a later operator if it is needed.

In summary, predicates are only added to `subplanGoals` if they establish a precondition of a future operator or a goal, and predicates that are negated before satisfying a precondition or goal can never be added.

4.4.3 Classifying Subplan Goals

Figure 4.24 contains pseudocode for an algorithm to determine what prospective goal predicates are relevant to achieving any of a set of specified preconditions. The first parameter, `currentState`, is the world state resulting from applying all previous operator postconditions. The second parameter, `subplanGoals`, will contain all of the preconditions of future operators that need to be achieved by this subplan. The third parameter, `subplanGoalProspects`, contains all of the candidate goal predicates. The fourth parameter, `preconditions`, contains the operator preconditions (or goals) that must be accomplished for this operator.

For each potential goal, `Update-subplan-goals` first checks to see if it is false. If so, if it is important, it must be established by a later precondition, so it is removed from the candidate goal set. The potential goal is then checked against every predicate in preconditions. If it establishes any of them, it is added to the goal set.

`Update-subplan-goals` has the following properties:

- All predicates from `subplanGoalProspects` that are false in `currentState` are removed from `subplanGoalProspects`. This property is important in order to prevent "false positives" from being added to `subplanGoals`.

- All predicates from `subplanGoalProspects` that are in `preconditions` get added to `subplanGoals`

- Worst-case execution time is $O(s(P_c p + \log s))$, where s is the number of states, p is the maximum number of preconditions for an operator, and P_c is the worst-case complexity for determining if a predicate satisfies a precondition.

```
Update-subplan-goals(currentState, subplanGoals
                     subplanGoalProspects, preconditions)

  for each predicate in subplanGoalProspects
    if predicate is false in currentState
      remove predicate from subplanGoalProspects
    else
      for each precondition in preconditions
        if predicate establishes precondition
          add predicate to subplanGoals
          remove predicate from subplanGoalProspects
```

Figure 4.24: Classifying Subplan Goals

The first two properties follow immediately from the above pseudocode. It is important to note that it is first determined whether a predicate is superfluous before it is checked against the preconditions; in the other order, superfluous predicates could get added.

The worst-case execution time follows from multiplying the number of iterations of the outer loop by the execution time of the interior. Checking to see if a predicate is false is $O(\log s)$ because the states are stored in a red-black tree. The complexity of checking to see if a predicate establishes a precondition depends upon the complexity of the precondition itself. If the precondition is simply a predicate, it is constant time. If the precondition contains a quantifier, it can require checking every possible instantiation of the quantifier (up to $O(s)$ checks for each quantifier).

4.5 Selecting Subplans to Replace

The center of each subplan replacement is called an *anchor*. There is an anchor for the start and end of each sequence of operators inserted by `Repair-plan`. Replacements proceed in order of increasing size of the replaced subplan.

The number of subplan levels specified determines the granularity of this anytime planning algorithm. Let L be the current subplan level, N be the total number of subplan levels, and R be the number of operators in the plan returned by the call to `Repair-plan`. The size of the subplan replaced at level L is $\frac{L}{N}R$. That is, the current level specifies the fraction of the plan that is to be replaced at each anchor. Figure 4.25 gives pseudocode for the `Local-replan` function. `Local-replan` iteratively calls `Replace-subplan` for each anchor at each subplan level. It incorporates the above formula for determining subplan size based upon subplan level.

`Local-replan` has five parameters. `Plan` is the plan that failed. `StartState` represents the current world state after the plan failure that triggered this plan repair episode. `GoalSet` contains the current system goals. `NumLevels` is the total number of subplan levels (that is, the variable N used above). `MaxLevel` is the maximum subplan level for which a replacement will actually occur on this run of `Local-replan`. By calling `Local-replan` with varying values for `max-level`, we can simulate different interruption times.

The `Local-replan` function has the following properties:

- `fixedPlan` is always a valid plan.
- The cost of `fixedPlan` never decreases.

Both of the above properties follow immediately from the same properties of `Replace-subplan` (see Section 4.4). The quality of plans produced by this algorithm is determined solely as the result of experimentation, as described in Chapter 5.

```
Local-replan(plan, startState, goalSet, numLevels, maxLevel)

  let (fixedPlan, newOperatorNums)
    = Repair-plan(plan, startState, goalSet)
  let subplanAnchors = Find-subplan-anchors(newOperatorNums)
  for level = 1 to maxLevel
    let subplanSize = fixedPlan.num-operators * level / numLevels
    for each subplanAnchor in subplanAnchors
      let (subplanStartOp, subplanEndOp)
        = Compute-subplan-start-end(subplanAnchor, subplanSize,
                                    fixedPlan)
      let oldPlanSize = fixedPlan.num-operators
      fixedPlan = Replace-subplan(fixedPlan, startState, goalSet,
                                  subplanStartOp, subplanEndOp)
      Update-subplan-anchors(subplanAnchors, subplanAnchor,
                             subplanEndOp,
                             oldPlanSize - fixedPlan.num-operators)
  return fixedPlan
```

Figure 4.25: Anytime Replanning

Three important helper functions used by `Local-replan` are:

- `Find-subplan-anchors`
- `Compute-subplan-start-end`
- `Update-subplan-anchors`

`Find-subplan-anchors` determines the anchor points for the subplan replacement. For each operator that was inserted by `Repair-plan`, if it is the start or end of a sequence of inserted operators, it gets added to the list of subplan anchors.

`Compute-subplan-start-end` is described in Figure 4.26. It determines the operators where the subplan starts and ends. The `subplanAnchor` parameter indicates where the replacement will be anchored. The `subplanSize` parameter gives the size of the replacement, and `plan` is the plan upon which the replacement will be performed. The function determines the start and end operators based on anchoring a subplan of `subplanSize` around `subplanAnchor`.

The first line of code ensures that the first subplan operator is not lower than the index of the first operator of the plan. The last section of code ensures that the last subplan operator is not greater than the index of the last operator in the plan. In either case, a subplan of the full size is replaced; however, the anchor is effectively moved forward or backward accordingly.

Figure 4.27 shows how anchors are gradually eliminated. An anchor gets removed when it falls within the radius of another anchor.

4.6 Summary

In this chapter, I have described an anytime replanning algorithm, `Local-replan`. It utilizes `Plan-repair`, an algorithm that can find a valid repaired plan in polynomial

```
Compute-subplan-start-end(subplanAnchor, subplanSize, plan)

  subplanStartOp = maximum(subplanAnchor - (subplanSize / 2),
                           plan.firstPlanOp)
  subplanEndOp = subplanStart + subplanSize - 1

  if subplanEndOp > plan.lastPlanOp
    subplanStartOp -= (subplanEndOp - plan.lastPlanOp)
    subplanEndOp = plan.lastPlanOp
```

Figure 4.26: Finding Subplan Boundaries

```
Update-subplan-anchors(subplanAnchors, currentAnchor, subplanEndOp,
                       numOpsRemoved)

  subtract half of numOpsRemoved from currentAnchor
  for each subplanAnchor in subplanAnchors later than currentAnchor
    if subplanAnchor <= subplanEndOp
      remove subplanAnchor from subplanAnchors
    else
      subtract numOpsRemoved from subplanAnchor
```

Figure 4.27: Updating Anchors

time, and `Replace-subplan`, an algorithm that can improve the quality of an existing valid plan by replacing a subplan. I have shown that the running time of these algorithms is dominated by the running time of the planning algorithm used for selecting operators (i.e. `Find-plan`). That is, these algorithms introduce no extra worst-case complexity beyond that of `Find-plan`. I did not show any provable guarantees about the magnitude of plan quality improvements using these algorithms. I have only been able to obtain that information through experimentation, as described in Chapter 5. I also described a new anytime planning algorithm, `Find-plan`, that uses depth-first search. This algorithm is both called by `Plan-repair` and `Replace-subplan`, and is also used as a baseline for evaluating the performance of `Local-replan`. `Find-plan` is the first planner to take advantage of the assumption that a single depth-first search can find a valid plan in order to implement anytime planning.

5

Experimental Evaluation

The goal of this research is to determine whether local subplan replacement provides an effective means for organizing an anytime search for a plan in the context of repairing a preexisting plan that has encountered a failure. Local subplan replacement is intended to utilize good decisions made during the search for the preexisting plan. By first inserting operators into this plan to restore it to validity, and then progressively improving it by replacing subplans, the hope is that good operator sequences found for the original plan will be preserved, contributing to a high-quality repaired plan.

My experiments utilized both the basic version of `Find-plan` as described in Section 4.2 and the modified version from Section 4.2.3. Each served both as a baseline and for generating new subplans, yielding four total algorithms:

- `LocalSub-1` is local subplan replacement using unmodified `Find-plan`
- `Baseline-1` is unmodified `Find-plan` serving as a baseline
- `LocalSub-2` is local subplan replacement using modified `Find-plan`
- `Baseline-2` is modified `Find-plan` serving as a baseline

The `Baseline` algorithms here were organized as anytime planning algorithms (as discussed in Section 4.2). For each run of a `LocalSub` algorithm, the corresponding `Baseline` algorithm is also executed. The `Baseline` run is given a search limit equal to the number of nodes expanded by the `LocalSub` run. This enables the `Baseline` algorithm to provide a frame of reference for the ability of the `LocalSub` algorithms to organize an anytime search for a plan.

In each experiment, an initial plan for repairing broken machines in the Repair Robot domain (described in Section 2.5) experiences a sequence of failures as it is being executed. After each failure, the plan currently being executed is first repaired; it is then improved using local subplan replacement. The corresponding baseline made no use of the original plan; it generated a new plan with the world state subsequent to the plan failure serving as its initial state.

As described in Section 4.5, the `LocalSub` algorithms replace subplans of increasing size depending upon the amount of time available. The progression of increasing subplan size is expressed in terms of *subplan levels*. Each level indicates the fraction of the total number of operators in the plan that gets replaced. Performance was measured after each subplan level in order to investigate the dynamics of how `LocalSub` improves plans given varying amounts of search.

In order to understand the dynamics of `LocalSub` in different situations, experiments were varied along several different axes including failure severity and the size of the initial plan. Failure severity is assessed because different types of failures can disrupt the original plan to different degrees, and it is important to understand how different amounts of disruption affect the performance of local subplan replacement. Plan length is important because local subplan replacement may have a greater advantage with more material available from an original plan to utilize.

In summary, I decided to evaluate the ability of local subplan replacement to organize an effective anytime search by answering the following research questions:

1. How does the overall performance of local subplan replacement compare to the baseline algorithms?
2. To what extent does local subplan replacement effectively utilize the decisions made by the original search process?
3. How does the length of the initial plan influence local subplan replacement?
4. How do variations in failure severity influence the plans returned by local subplan replacement?

Here are some summary answers to the research questions:

- Regarding question 1, the overall average performance of `LocalSub-1` was superior to that of `Baseline-1`, while `LocalSub-2` and `Baseline-2` were approximately equivalent. At levels 1 and 2, `LocalSub-2` had a higher average cost than `Baseline-2`; at higher levels, `LocalSub-2` had a lower cost.[1] Because `LocalSub-1` so strongly dominated the performance of `Baseline-1`, most of the remaining analysis performed focused on `LocalSub-2` and `Baseline-2`.
- Regarding question 2, I measured the utilization based on the percentage of operators in each new plan that could be found in the original plan. I used this same metric to determine the resemblance between the plans discovered by each baseline and the original plan. `LocalSub-1` and `LocalSub-2` both had a consistently higher correspondence to the original plan, indicating that the original plan does indeed have a strong influence on the result of the search.

[1]Recall that in the Repair Robot domain, plan cost refers to the total amount of *lost* production. Hence, a plan with lower cost is preferable.

- Regarding question 3, the length of the initial plan had a very significant impact on relative performance, with longer initial plans being utilized more effectively than shorter ones by `LocalSub-2`.

- Regarding question 4, plans generated by `Baseline-2` for handling the more severe failures had a lower average cost than those generated by `LocalSub-2`, while the reverse was the case with the less severe failures. Plans that handled severe failures well tended to utilize fewer operators from the original plan in comparison to plans that handled mild failures well.

Section 5.1 describes the details of the experimental infrastructure. The sections that follow describe and analyze results from the experiments. All graphs containing averages also include 95% confidence intervals. In many cases, the confidence intervals are so small as to be invisible, but they are present in all graphs nevertheless. All graphs containing data comparing a `LocalSub` algorithm with its `Baseline` have `Baseline` to the right in the darker-shaded bar.

5.1 Experiment Design

In each experiment, `LocalSub` is run in order to improve a failed (and subsequently repaired) plan. All plans are situated in the Repair Robot domain (described in Section 2.5). In this domain, a robot travels around a factory repairing broken manufacturing equipment. The cost of a plan is the total amount of lost productivity that occurs before the robot has fixed the last broken machine. Good plans minimize the lost production, and hence have low values for cost. The cost is the means by which I have quantitatively analyzed the performance of the planning algorithms. In the experiments, planning is assumed to be instantaneous; the time spent planning does not affect the cost computed for that experiment. The comparison is still sound, as both planning algorithms run for the same period of time.

Each experiment is given the following:

- Initial plan
- Initial world state in which the initial plan will be applied
- Total subplan levels
- Maximum subplan level to actually use
- A failure script

A *failure script* contains a list of failures that will occur as the plan is executed. Each failure has two components. The first component indicates when the failure occurs and the second component indicates what changes to the world state result from the occurrence of the failure. The timing of the failure is specified in terms of the occurrence of a particular operator. For example, a failure can happen before the second time an `activate-machine` operator gets applied. The changes to the world state are given in terms of add and delete lists. The predicates added and deleted are specified in terms of parameters shared with the operator used to specify failure timing. In these experiments, failures consisted of the destruction of parts in a machine immediately before the robot attempts to use the `activate-machine` operator to restart production.

A group of initial plans was created prior to running any of the experiments. The same initial plan could be used with different failure scripts in different experiments.

Each individual experiment proceeded as follows:

1. Set the current plan to be the initial plan, the current world state to be the initial world state, the total cost to 0, and the current failure number to be 0.

2. Apply each operator of the current plan, updating the current world state and accumulating cost values, until a failure is indicated in the failure script or all operators have been applied. Go to step 7 upon the application of the final operator.

3. Increment the failure number by 1.

4. Apply the failure from the failure script to the current world state.

5. Call `Local-replan` (described in Section 4.5). The result is the new current plan. Record the number of nodes expanded in an array, `nodes-searched`, indexed by the failure number.

6. Go back to step 2.

7. Repeat steps 1 through 6 using `Find-plan` (described in Section 4.2) instead of `Local-replan` in step 5. Set the node limit for `Find-plan` to be the element of `nodes-searched` indexed by `failure-number`.

Variations in map layout and machine part configuration were included in order to formulate a large enough number of distinct problem instances in combination with the other experimental variations so as to have statistically significant results. A range of values for the subplan level was included in order to address research question 1, variations in the number and severity of failures were included in order to investigate research question 4, and variations in the sizes of the initial plans were included in order to investigate research question 3.

Here are the details of these experimental variations:

- As discussed earlier, every experiment was run with one local subplan replacement algorithm and one baseline algorithm. The first pair is `LocalSub-1` with `Baseline-1`; the second pair is `LocalSub-2` with `Baseline-2`.

- Three different maps were used. Each map represents a different layout of machines in the factory. All maps have ten machines and three storehouses.

- Machines could have three or six parts each. In all experiments, each machine had the same number of parts. With three parts, each machine had two parts of one type and one part of a second type. With six parts, machines had various combinations of one, two or three part types.

- Each combination of map and machine parts (6 total combinations) has two initial plans: one plan that responds to a situation in which seven out of the ten machines break, and a second plan that responds to a situation in which all ten machines break. In addition, more parts per machine break in the scenario in which all ten machines break. This variation was included so as to have experiments in which relatively short and long initial plans were modified.

- Each first initial plan was subjected to three distinct failure sequences; each second initial plan was subjected to four.

- For each map, 17 different specific failure patterns were given for each failure sequence. Seven of the patterns applied to machines with three parts each; the other ten patterns applied to machines with six parts each. For each sequence, there were six patterns in which one part failed per machine, six patterns with two parts failing, four patterns with three parts failing, and one pattern with six parts failing.

- In all experiments, the robot has a carrying capacity of three spare parts.

- For all experiments, the total number of subplan levels is six. Each experiment was run with each possible maximum subplan level from one to six.

In summary, there are two planning algorithms, seven failure sequences, three maps, 17 failure patterns, and six subplan levels, totaling 4284 experiments. More details about machine configurations, initial states, and failure scripts are given in Appendix B.

5.1.1 Planning Algorithm Parameters

The planning algorithms were configured as follows. Each call to `Find-plan` made by `Repair-plan` was allowed one depth-first search. The depth limit for these searches was 200 for the experiments where each machine had three part slots, and 380 for the experiments where each machine had six part slots.[2]

The initial plans were generated by calling `Baseline-2` with 100,000 depth-first searches. The initial plans generated averaged 51 operators for the plans to repair seven machines and 83 operators for the plans to repair 10 machines.

Calls to `Find-plan` from `LocalSub` were allowed 1000 depth-first searches given a depth maximum of 20. Given other depth limits, the allowable number of depth-first searches was varied proportionately. With a depth maximum of 10, 500 depth-first searches were allowed; with a depth maximum of 40, 2000 depth-first searches were allowed.

[2]The depth limits were determined by observing that a simple Repair Robot plan requires six operators per part and two operators per machine. For each part, the robot might have to (1) travel to a storehouse, (2) unload a part currently held, (3) travel to another storehouse, (4) grab a part, (5) travel to a machine, and then (6) install the part. For each machine, the robot has to (1) travel to the machine and (2) activate it

5.2 Overall Results

An overall comparison of `LocalSub-1`, `Baseline-1`, `LocalSub-2`, and `Baseline-2` is given in Figure 5.1. Using `LocalSub-2` results in a statistically significant lower average cost than using `LocalSub-1`. Both `LocalSub-1` and `LocalSub-2` compare favorably against `Baseline-1`. `LocalSub-2` is very slightly better in performance compared against `Baseline-2`.

The fact that `Baseline-2` performs better than `Baseline-1` was not unexpected, but I did not have an intuition as to the degree of difference in cost until viewing the results of these experiments. The modifications to `Baseline-2` described in Section 4.2.3 were designed to prevent `Find-plan` from spending too much time searching a narrow part of the space. I hypothesized that forcing a broader search of the space would improve the resulting plan, and these experiments verify that hypothesis.

The fact that the differences between `LocalSub-1` and `LocalSub-2` were relatively small was also not surprising, as only a relatively small part of each plan was generated by the underlying algorithm. We will see in Section 5.2.1 that when examined in terms of the individual subplan levels, `LocalSub-2` provides a much more significant advantage at the higher subplan levels in which larger subplans get replaced.

5.2.1 Subplan Levels

As described in Chapter 4, the higher the available subplan level, the larger the space that is searched. I hypothesized that the increase in search space would imply a corresponding increase in solution quality. It is not necessarily the case that performing more search will lead to a higher quality plan. Any incomplete search heuristic will fail to examine part of the search space. As both `LocalSub` and `Baseline` are incomplete search heuristics, the degree of improvement achieved by additional search had

Figure 5.1: Global Replanning Variations vs. Local Subplan Replacement Variations (2142 runs/algorithm)

to be measured empirically.

In order to test this hypothesis, I ran experiments in which local subplan replacement was given 1, 2, 3, 4, 5, and 6 levels, with the maximum level set at 6 in all cases. In each experiment, $\frac{L}{6}$ of the plan operators were replaced at each replacement location for each subplan level L. Each `Baseline` was run with a node limit equal to the number of nodes expanded by `LocalSub` at that level.

Figure 5.2 aggregates the average costs of `LocalSub-1` and `LocalSub-2` given in Figure 5.1 and then decomposes these aggregated costs according to subplan levels. The same aggregation and decomposition is done with regard to `Baseline-1` and `Baseline-2`. Figures 5.3 and 5.4 decompose the datasets given in Figure 5.2 by algorithms 1 and 2, respectively.

The data given in Figure 5.2 verifies the hypothesis that `LocalSub` produces higher quality plans when using higher subplan levels. However, the average performance of `Baseline` tends not to vary much with the subplan level. The same relative results for `Baseline-1` and `Baseline-2` can also be seen in Figures 5.3 and 5.4.

Because of the restrictions placed on the number of nodes `Baseline-1` can expand, it never expands most of its top-level child nodes. Hence, it considers only one or a few possibilities for its first choice of goal to solve. Due to the nature of the cost function, a poor choice for the first goal to achieve can result in a plan that involves a lot of travel before the first machine gets repaired. This travel can result in a large amount of lost production. Instead, `Baseline-1` spends most of its search effort exploring nodes at high depths, effectively considering plans that differ primarily in the last few operators.

`Baseline-2` is designed to expand all of its top-level child nodes by subdividing its search space into separate groups of depth-first search (as described in Section 4.2.3). Each group begins its search with a different top-level node, thus avoiding the problem mentioned above regarding `Baseline-1`. Still, within each group the search performed behaves similarly to `Baseline-1`, spending most of its effort exploring nodes at high depths.

In addition to seeing the average decrease in plan cost corresponding to each increase in subplan level, it is also useful to know how often the application of a subplan level actually results in finding a replacement plan of lower cost. The frequency of change was measured by counting, for each subplan level for each plan repair, whether an improvement occurred as a result of a subplan replacement at any anchor. This count was divided by the total number of plan repairs. In Figures 5.5, 5.6, and 5.7, we can see that the measured frequency of change decreases with an increase in subplan level. This decrease is most strongly marked in Figure 5.6, which is consistent with the fact that Figure 5.3 shows that `LocalSub-1` does not improve much at all after the first three levels. The more moderate decrease shown for `LocalSub-2` in Figure 5.7 is consistent with the more consistent improvements in cost seen in Figure 5.4.

5.3 Analysis Metrics

5.3.1 Utilization of Original Search

In order to understand the extent to which `LocalSub` benefits from the decisions made in the search for the preexisting plan, I measured the percentage of operators from the preexisting plan that could be found in various plans returned by the `LocalSub`. I used this same metric to determine the resemblance between the plans discovered by each `Baseline` and the original plan. While the baselines do not use the original plan to guide their search, they could produce operator sequences that resemble sections of the original plan. Measuring this duplication provides a baseline of assessing the

Figure 5.2: Plan Cost vs. Subplan Level (714 runs/level)

Figure 5.3: Plan Cost vs. Subplan Level: LocalSub-1 and Baseline-1 (357 runs/level)

Figure 5.4: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2 (357 runs/level)

Figure 5.5: Plan Change Percentage vs. Subplan Level: LocalSub-1 and Baseline-1 (357 runs/level)

Figure 5.6: Plan Change Percentage vs. Subplan Level: LocalSub-1 and Baseline-1 (357 runs/level)

Figure 5.7: Plan Change Percentage vs. Subplan Level: LocalSub-2 and Baseline-2 (357 runs/level)

significance of utilization in the corresponding `LocalSub` experiments. In addition, comparing the utilization of `Baseline-1` and `Baseline-2` can indicate different kinds of underlying dynamics between these two algorithms.

Figure 5.8 contains the percentages for `LocalSub-1` and `Baseline-1` averaged over all experiments. Figure 5.9 contains the percentages for `LocalSub-2` and `Baseline-2`. The `LocalSub` planners have a consistently higher percentage of original plan operators than the `Baseline` planners. This demonstrates that the search conducted by `LocalSub` is being guided by decisions made in the search for the original plan.

Comparing Figure 5.8 with Figure 5.3, we can see that the 50% utilization rate of `Baseline-1` consistently produces plans of relatively high cost. The utilization levels of 80% and above for `LocalSub-1` suggest that the low cost of `LocalSub-1` relative to `Baseline-1` can be credited to those operators.

Comparing Figure 5.9 with Figure 5.4, we can see that at levels 1 and 2, utilization rates above 75% result in plans with higher cost than their corresponding baselines, which have utilization rates around 65%. This suggests that it is beneficial for `LocalSub` to eliminate some of the operators from the original plan. At levels 3, 4, 5, and 6, this is exactly what happens. At those levels, the cost of `LocalSub-2` drops below that of `Baseline-2`. The utilization rate of `LocalSub-2` also declines, although it converges around 70%, still higher than the peak utilization rate for `Baseline-2`. This suggests that `LocalSub-2` has discovered a useful subset of the operators from the original plan for the purpose of the new plan, and that this is a capability `Baseline-2` does not demonstrate.

5.3.2 Repair Penalty

The *repair penalty* can serve as an abstract guide for determining the performance of `LocalSub-2` given different situations. In particular, it will be used for comparing the

Figure 5.8: Original Plan Operator Utilization vs. Subplan Level: LocalSub-1 and Baseline-1 (357 runs/level)

Figure 5.9: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2 (357 runs/level)

results for failure severity (Section 5.4) and plan length (Section 5.5). It represents a greedy distance measure in plan space from the original failed plan to a repaired valid plan.

The repair penalty is calculated as follows. Assume the original plan would have cost c if no failure had occurred. Cost c can be partitioned into d, the cost prior to the failure point, and t, the cost after (still assuming that the failure did not occur). Note that $t = c - d$. Once the failure occurs, an initial repaired plan is generated. This plan has cost r. Inserting extra operators results in the net addition of $r - t$ to the total cost, compared with the failure not having occurred. The repair penalty is the percentage of the cost of the repaired plan that results from these extra operators; in other words, $100 \times \frac{r-t}{r}$.

5.4 Failure Severity

Failure severity was measured based upon the number of parts that failed in a single machine. Failures of 1, 2, 3, and 6 parts were considered. Increasing the number of part failures resulted in both `LocalSub` and `Baseline` producing higher overall cost plans. In Figure 5.10, we can see that the overall aggregated results for `LocalSub` were superior to the aggregated results for `Baseline`.

However, the superior performance of `LocalSub` depends heavily upon the baseline of comparison and the underlying planning algorithm. Figure 5.11 shows the results for `LocalSub-1` vs. `Baseline-1`. `LocalSub-1` does very well in comparison. In contrast, in the case of `LocalSub-2`, Figure 5.12 shows that in the cases of 3 and 6 part failures, `Baseline-2` does better.

Because of the ambiguous advantages of `Baseline-2`, I further analyzed that data in terms of the subplan levels as well as the number of part failures. The data for

Figure 5.10: Plan Cost vs. Number of Failed Parts (1 or 2 parts: 1512 runs; 3 parts: 1008 runs; 6 parts: 252 runs)

Figure 5.11: Plan Cost vs. Number of Failed Parts, LocalSub-1 and Baseline-1 (1 or 2 parts: 756 runs; 3 parts: 504 runs; 6 parts: 126 runs)

Figure 5.12: Plan Cost vs. Number of Failed Parts, LocalSub-2 and Baseline-2 (1 or 2 parts: 756 runs; 3 parts: 504 runs; 6 parts: 126 runs)

1-part failures is given in Figure 5.13, for 2-part failures in Figure 5.14, and for 3-part failures in Figure 5.15.

`LocalSub-2` performs worse than `Baseline-2` at the first subplan level in all cases, even the otherwise favorable 1-part failure data. `LocalSub-2` is at least statistically tied with `Baseline-2` by the 4th subplan level in all three cases.

In order to further analyze the behavior of `LocalSub-2` across these different severities of failure, we will now examine the impact of failure severity on how `LocalSub-2` utilizes operators from the original plan. With 1 part failure, `LocalSub-2` performs better than `Baseline-2` starting at subplan level 2. In Figure 5.16, we see that at subplan level 2, the utilization is around 80% for `LocalSub-2`. Those operators from the original plan have helped guide `LocalSub-2` to a better plan than could be achieved by `Baseline-2` with the same amount of search.

With 2 part failures, `LocalSub-2` does not perform better than `Baseline-2` until subplan level 3. At that point, the utilization for `LocalSub-2` is around 75%; about 5% lower than for the 1 part failure scenario (see Figure 5.17). With 3 part failures, `LocalSub-2` does not perform better than `Baseline-2` until subplan level 5, where according to Figure 5.18 utilization is approximately 70%.

These examples demonstrate that depending upon failure severity, there are different levels of helpful utilization of operators from the original plan. Retaining too many operators can have negative consequences. However, retaining the right proportion of operators enables `LocalSub-2` to find better plans than `Baseline-2` given the same amount of search.

Figure 5.19 shows the average repair penalty for each level of failure severity, averaged across all experiments. The repair penalty closely tracks severity. This shows a relationship between the average repair penalty and the increased difficulty of finding a low-cost plan relative to `Baseline-2`.

Figure 5.13: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1 Part Failure (126 runs/level)

Figure 5.14: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 2 Part Failure (126 runs/level)

Figure 5.15: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 3 Part Failure (84 runs/level)

Figure 5.16: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2, 1 Part Failure (357 runs/level)

Figure 5.17: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2, 2 Part Failure (357 runs/level)

Figure 5.18: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2, 3 Part Failure (357 runs/level)

One part failure problems average 29% for repair penalty, and from Figure 5.13, we see that with one part failure, `LocalSub-2` drops below `Baseline-2` at subplan level 2. Two part failure problems average 34% for repair penalty, with `LocalSub-2` dropping below `Baseline-2` at subplan level 3 (Figure 5.14). Three part failures average a repair penalty of 41%, with the drop at subplan level 5 (Figure 5.15).

5.5 Initial Plan Length and Failure Spacing

I ran experiments using seven different failure spacings. In the failure scripts for each experiment, each failure was specified as the *nth* instance of the `activate-machine` operator since the previous failure. For example, a failure spacing of 1-2-2 indicates that a failure will occur at the first attempt to activate a machine, another failure after the second attempt following replanning, and a final failure after the second attempt following the second replanning episode. In each failure, only the machine the robot attempts to activate fails. The number of parts that fail in that machine vary as described in Section 5.4.

In all maps, ten total machines are present. The spacings 1-2-2, 1-3-3, and 1-4-1 were used with initial plans that fix seven machines. The spacings 1-2-2-2-2, 1-5-3, 1-4-4, and 1-6 were used with initial plans that fix all ten machines. These spacings were selected in order to investigate the effect of permitting the repaired and improved plans to run for varying amounts of time before experiencing new failures.

The results presented in this section focus on local subplan replacement using `LocalSub-2` compared to `Baseline-2`. This focus was selected because local subplan replacement completely dominates `Baseline-1` in all cases, whereas in comparison with `Baseline-2` things are more interesting.

Figure 5.20 shows, for each subplan level, the average cost of the overall runs that

Figure 5.19: Repair Penalty vs. Failure Severity

use an initial plan that repairs seven machines. It was not until subplan level 6 that `LocalSub-2` performed better than `Baseline-2`. Figure 5.21 shows the runs using an initial plan that repairs ten machines and indicates that `LocalSub-2` performs better starting at subplan level 3.

On average, the plans generated by `LocalSub-2` using a larger initial plan performed significantly better in comparison to `Baseline-2` than those using a smaller initial plan. Figure 5.22 shows the average repair penalty for each initial plan length category. Short plans have much higher average repair penalties than long plans.

In Figure 5.20, we see that with the shorter initial plans, the average level at which the drop occurs is level 6, in contrast to the long initial plans, where Figure 5.21 indicates the drop occurring at level 2. The average repair penalties are 32% and 43% respectively. These values fall in line with the values given above in Section 5.4 for failure severity, demonstrating the utility of the repair penalty as a metric for domain difficulty for `LocalSub-2`.

The plans created using longer initial plans have a more consistent advantage over their baselines than those created using shorter initial plans. The shorter initial plans are on average statistically tied with their baselines until the operators at the end of each plan are reached. As it happens, the opening is never reached; the average operator failure happens at operator 20, well before the split after operator 30. The application of only the first group of operators from the new plan also helps to explain why in Figures 5.27, 5.28, and 5.29, overall performance is very similar.

Figure 5.23 shows the average cost by operator for the plans generated in response to the first failure for all runs with the shorter initial plans. Figure 5.24 shows the same information for all the scenarios starting with the larger initial plan. These costs were calculated by keeping track of the accumulated cost after applying each operator in the course of each experiment.

Figure 5.20: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, Initial Plan Repairs 7 Machines (153 runs/level)

Figure 5.21: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, Initial Plan Repairs 10 Machines (204 runs/level)

Figure 5.22: Repair Penalty vs. Initial Plan Length

Figure 5.23: Smaller Initial Plan, by Operator

Figure 5.24: Larger Initial Plan, by Operator

Although Figure 5.23 shows how `LocalSub-2` is at a disadvantage with a short initial plan (in comparison to a long initial plan), prior to the failure they are still statistically tied in performance. In Figure 5.25, the decreasing performance of `LocalSub-2` after each failure is demonstrated. Figure 5.26 shows how the repair penalty increases after each failure.[3] This occurs as the remaining plans become smaller, offering further evidence of the relationship between the repair penalty and expected utility of `LocalSub-2`.

Figure 5.31 shows spacing 1-4-4 and Figure 5.33 shows spacing 1-6. In comparing these two spacings with 1-2-2-2-2 (Figure 5.30), we see that the presence of a larger number of failures results in a higher overall cost. Because each failure occurs immediately prior to activating a machine, scenarios with more failures will have longer operator sequences between activations in comparison to scenarios with fewer failures. These longer sequences add overhead to the cost of the plan in comparison to a plan that experienced fewer failures.

The results for operator utilization given initial plan length are shown in Figure 5.34 for short initial plans and Figure 5.35 for long initial plans.

In Figure 5.20, we see that with the shorter initial plans, the average level at which the drop occurs is level 6, in contrast to the long initial plans, where Figure 5.21 indicates the drop occurring at level 2. The operator utilization percentages at the drop points are 73% and 78% respectively.

The utilization percentage for the long initial plans is comparable to that for the failure severity levels with comparable repair penalties. However, the utilization for the short initial plans is high compared to its counterpart in failure severity. As many different plans of different costs can be associated with the same utilization value, this is not necessarily surprising. Still, a general trend that an increased repair penalty

[3]The data contained in Figures 5.25 and 5.26 was averaged across all runs with short initial plans.

Figure 5.25: Plan Cost vs. Failure: Short Initial Plan: LocalSub-2 and Baseline-2

Figure 5.26: Repair Penalty vs. Failure: Short Initial Plan

Figure 5.27: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-2-2 (51 runs/level)

Figure 5.28: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-3-3 (51 runs/level)

Figure 5.29: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-4-1 (51 runs/level)

Figure 5.30: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-2-2-2-2 (51 runs/level)

Figure 5.31: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-4-4 (51 runs/level)

Figure 5.32: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-5-3 (51 runs/level)

Figure 5.33: Plan Length vs. Subplan Level: LocalSub-2 and Baseline-2, 1-5-3 (51 runs/level)

Figure 5.33: Plan Cost vs. Subplan Level: LocalSub-2 and Baseline-2, 1-6 (51 runs/level)

implies a lower beneficial level of utilization does seem to hold.

5.6 An Incomplete Baseline Run

In a small number of the subplan level 1 runs, local subplan replacement used so few search nodes that the baseline algorithm did not have enough nodes available to it to enable it to find even a single plan. An example is given in Figure 5.36.

The plan in Figure 5.36 is the result of plan repair with no subplan replacements. The cost of this plan is 630 and it has 11 operators. As there are six total subplan levels, the subplan replacement size formula indicates that at level 1, $\frac{1}{6}$ of the operators are to be replaced at each replacement point. With 11 operators total, the system rounds down by default, and the subplan size for replacement is just 1 operator. Based on the size of the plan, the system dictates that 50 plans be attempted for each replacement point. (The parameter configuration resulting in this is discussed in Section 5.1.1.) There are two such points in the above plan, resulting in 100 depth 1 searches for replacement plans. 100 total nodes get expanded. Unsurprisingly, the plan remains unchanged.

The baseline algorithm is then given 100 nodes for its search. In this case, the modified `Find-plan` algorithm is being used. For this particular problem instance, modified `Find-plan` partitions its search space into ten subspaces, each of which is given a maximum of ten nodes for its search. Ten nodes being insufficient for finding a plan, none of the subsearches find a plan.

This behavior was observed in seven out of 2142 total experiments that used the modified `Find-plan` algorithm. It did not occur in any experiments using unmodified `Find-plan`. The subdivision of the state space used in the modified `Find-plan` algorithm was essential for this problem to occur in each instance. It only ever happens at

Figure 5.34: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2, Initial Plan Repairs 7 Machines (153 runs/level)

Figure 5.35: Original Plan Operator Utilization vs. Subplan Level: LocalSub-2 and Baseline-2, Initial Plan Repairs 10 Machines (204 runs/level)

```
 1 (travel 1 11)
 2 (get-part-from 11 1 0 2)
 3 (get-part-from 11 1 1 3)
 4 (travel 11 1)
 5 (place-part-at 1 1 0 0)
 6 (place-part-at 1 1 1 2)
 7 (travel 1 12)
 8 (get-part-from 12 2 0 7)
 9 (travel 12 1)
10 (place-part-at 1 2 0 4)
11 (activate-machine 1)
```

Figure 5.36: Repaired Plan

subplan level 1; at higher levels, given my test cases, there are always enough nodes available to ensure that the baseline generates at least one plan.

5.7 Computational Resources

Figure 5.37 shows the average number of nodes expanded for each subplan level. These numbers are cumulative; for example, the count for level 3 includes all the nodes expanded at levels 1 and 2. The number of nodes expanded increases significantly from each subplan level to the next level.

Determining the time required by the plan repair phase along with the average number of nodes expanded per second given a particular implementation and computing hardware can indicate the amount of wall-clock time needed by local subplan replacement.

On a one gigahertz Pentium III with one gigabyte of RAM, the repair phase takes an average of 1.31 seconds with a 95% confidence interval of 0.03 seconds. On the same machine, an average of 453 nodes were expanded per second, with a 95% confidence interval of 9 nodes per second. Based on the data from Figure 5.37, with my implementation on those machines, a replanning episode at level 1 took approximately one minute, at level 2 three minutes, at level 3 six and a half minutes, at level 4 ten and a half minutes, at level 5 15 minutes, and at level 6 an episode took about 22 minutes.

5.8 Discussion

My research has sought to investigate the idea of local subplan replacement as a means of taking advantage of the search effort previously expended to generate the

Figure 5.37: Nodes Expanded vs. Subplan Level (716 runs/level)

preexisting plan it is repairing. Research question 1 sought to investigate the overall performance of local subplan replacement in comparison to global replanning. In comparison to `Baseline-1`, `LocalSub-1` performed exceptionally well; compared to `Baseline-2`, performance of `LocalSub-2` was just slightly better on average.

`Baseline-1` fared poorly, as it focused too much of its search energy on a relatively narrow part of the search space. The broader exploration of `Baseline-2` was demonstrated to be helpful. As the amount of search space available increased, the performance of neither baseline improved much. `LocalSub-1` and `LocalSub-2` both demonstrated significant improvement given additional search, especially `LocalSub-2`.

An important aspect of this research is demonstrating the influence of the original plan on the search for a new plan (research question 2). I measured this influence based on the percentage of operators in each new plan that could be found in the original plan. I used this same metric to determine the resemblance between the plans discovered by each baseline and the original plan. Local subplan replacement has a consistently higher correspondence to the original plan, indicating that the original plan does indeed have a strong influence on the result of the search. My results also show that baselines that perform well have at least a 60% correspondence with the original plan.

Focusing on the severity of individual failures (in response to part of research question 4), more severe failures were problematic for `LocalSub-2` in comparison with `Baseline-2`. More severe failures are often handled best by plans that make less use of the original plan. Consequently, a great deal of search was required in some cases for `LocalSub-2` to find a plan with a cost competitive with `Baseline-2`.

Regarding research question 3, the length of the initial plan was a significant contributor to the success or failure of `LocalSub-2`. In particular, when the initial plan was relatively long, the plans generated by `LocalSub-2` were consistently superior to those of `Baseline-2` measured on an operator-by-operator basis. With short initial plans, any advantage held by `LocalSub-2` was not manifest until towards the end of the plan. In my experiments, those operators never got applied because subsequent failures would interrupt the plan prior to reaching that point.

The performance of `LocalSub-2` across different initial plan lengths and severity levels was summarized using the repair penalty. The repair penalty provides a greedy measure of the distance between the original failed plan and a repaired valid plan. A high penalty implies a larger greedy distance in plan space, and consequently greater difficulty for local subplan replacement to find a good plan.

6

Conclusions and Future Work

6.1 Conclusions

The purpose of this research was to determine whether local subplan replacement provides an effective means for organizing an anytime algorithm for planning in the context of repairing a preexisting plan that has encountered a failure. Of particular interest was evaluating the extent to which utilizing aspects of the search that generated the preexisting plan was helpful in organizing an anytime search for a plan.

The contributions of this research are as follows:

- Local subplan replacement is the first plan repair algorithm to guarantee the availability of a valid plan at all times. This valid plan is available on average within two seconds in my experiments. This contribution is achieved in that context by restricting the problem space to those for which a valid plan could be found in polynomial time. No previous work in plan repair has studied how that restriction could be effectively exploited.

- My results demonstrate empirically that local subplan replacement generates plans that do a better job of maximizing factory production than those generated by a global replanning algorithm under certain conditions.

- Limitations of local subplan replacement are demonstrated empirically. High values for the repair penalty, a greedy measure of the distance in plan space between the original failed plan and a valid repaired plan, are shown to correspond with local subplan replacement requiring large amounts of search to generate a plan of lower cost than the corresponding baseline. With a repair penalty of no more than 34%, local subplan replacement produced plans of lower cost than the baseline when subplans of at least one-third the overall plan size could be replaced. With higher values for the repair penalty, nearly all of the original plan needs to be replaced in order to generate a plan with lower cost than the baseline.

- The effect of using the original plan to guide the search is quantified and measured empirically. The plans generated by local subplan replacement contain significantly more operators from the original plan than those generated by the baseline on average. In many cases, especially when the repair penalty is relatively low, this utilization corresponds to a lower plan cost than the baseline, empirically demonstrating the benefit of utilizing material from the original plan in such situations.

Secondary contributions of this research include:

- A new anytime planning algorithm employing depth-first search is described. It utilizes the assumption that a valid plan can be found with a single depth-first search in order to implement its anytime progression. Unlike existing systems, it will always have available a valid plan, and it takes responsibility for generating the first plan in its anytime progression.

- An analytical demonstration of the relationship between different categories of planning problems for which valid plans can be found in polynomial time is given. I identify a particular type of delete effect that implements resource consumption as a major contributor to the intractability of finding plans in polynomial time.

6.2 Future Work

6.2.1 New Domains

Obtaining experimental results for domains beyond the Repair Robot domain would be of interest, with a particular focus on domains with different types of cost functions. Verifying the utility of the repair penalty with different cost functions could demonstrate its general applicability as a measure of the difficulty a particular problem instance will present to local subplan replacement.

Exploring other domains could test whether certain domain assumptions from the Repair Robot problem are significant. In particular, evaluating a multi-robot variation of the Repair Robot problem, in which there are more machines to be repaired than robots available, would be able to demonstrate whether the assumption of only one robot had any impact. Evaluating a package delivery problem (see Section 2.3.7.1) could provide a similar comparison while evaluating a different type of failure, namely, vehicles becoming disabled and unable to deliver packages.

6.2.2 Determining Time Allocation

Another important aspect of investigating local subplan replacement involves integrating it into an anytime planning system that allocates it time for computation based on system needs. Local subplan replacement could be used in conjunction with the deliberation scheduler of Dean et al. [12] or the related system of Zilberstein and Russell [64]. Both systems require a performance profile for each anytime algorithm they incorporate. The experimental results given in Chapter 5 are examples of the kinds of performance profiles that can be generated for local subplan replacement. It would again be very important to calibrate the available subplan levels to the timing requirements and capabilities of the target system. In particular, the deliberation scheduler could take into account the initial plan size and failure frequency in determining the amount of system resources it ought to dedicate to local subplan replacement. A properly configured deliberation scheduler could utilize local subplan replacement primarily when it is most likely to be useful.

6.2.3 Reorganizing Subplan Replacement

The issue of failure severity discussed in Section 5.4 demonstrated that in many situations, local subplan replacement stays too close to the original plan. Still, even in those situations, the utilization rate for a good plan was still above 60%. A method to enable larger jumps in the search space at low subplan levels might be able to help in finding these good plans earlier in the anytime progression. An example of such a method could be to perform multiple searches utilizing different components of the original plan.

A variation on this idea would be to divide the original plan into subplans that are all small enough to be executed before the next expected failure. Plan repair could be applied to each of these subplans in turn to produce a valid plan.

This idea could be very helpful in the Repair Robot domain, in which a plan consists of distinct tasks that can be performed in any order. For a domain with stronger ordering constraints, such as the Blocks World, this idea might be less effective, although in a Blocks World domain with a large number of small towers to be built it could still be helpful to try.

Along these same lines, further research could address the meta-level control problem for anytime algorithms. One particular focus could be for a control algorithm to use machine learning techniques to predict the timing of the next failure. This would be the source of the input to a modified local subplan replacement algorithm as described above.

6.2.4 Tractable Algorithms for Valid Planning

As I mentioned above, the restriction of local subplan replacement to domains for which valid plans can be found in polynomial time is a significant limitation. Further research on refining the categories of planning problems for which this is the case would be most helpful in broadening the applicability of local subplan replacement. In particular, the observation that it is easy to classify a planning problem in such a way as to make it seem a more difficult problem than it really is would seem to indicate that such category refinement is a distinct possibility. (An example of such an easily misclassified problem is the famous Blocks World problem; see Section 2.3.2 for more details about this.)

Although planning theory has shown that delete effects in plan operators are extremely harmful to the prospects of finding valid plans quickly, the practical utility the `Find-plan` algorithm described in Section 4.2 and systems that use hierarchical task networks indicates that delete effects do not intrinsically interfere with finding valid plans tractably across domains. The current state-of-the-art is to customize the search depending on very specific domain characteristics.

My belief is that delete effects are much more powerful constructs than most domain implementors actually need for encoding their domains. Determining what delete effects are actually used for, and designing algorithms with these issues in mind, could eventually lead to improved planning algorithms that are reasonably general (albeit less general than full STRIPS planning) but require less configuration than algorithms like `Find-plan` from Section 4.2 and hierarchical task networks.

Here are three common uses for delete effects in planning domains:

- Moving objects around
- Resource depletion
- Disassembly of assembled objects

Moving objects around is the centerpiece of logistics problems such as the Repair Robot problem and other related problems, many of which have been demonstrated to be tractable to solve for finding valid plans. Resource depletion can be handled if there is a reliable means of ensuring replenishment occurs. It is possible that object disassembly could also be handled efficiently in ways similar to moving objects around, but that is speculation on my part at this time.

Part of the intuition here is that the precondition, add effect, delete effect specification and states as predicates is analogous to the use of assembly language. What is needed is a higher level planning language where common tasks are easy. In my own experience, I have often written miniature compilers to generate states as encoded as predicates based on higher level specifications. Determining how generally useful such specifications could be, and what kinds of good algorithms could be used with them is a promising subject for further investigation.

6.2.5 Modified Depth-first Search

The modified DFS algorithm described in Section 4.2 was devised in order to have available a competitive benchmark global replanning anytime algorithm for evaluating local subplan replacement. In some situations, it performed very well, much better than traditional DFS. The goal was to achieve some kind of "statistical sampling" of the state space explored by DFS.

There already exist search algorithms that attempt to do similar things. Simulated annealing is one popular example. Simulated annealing is a variation of hill climbing in which disadvantageous moves are made in the search space with a probability that diminishes as the algorithm progresses. Comparing the ability of modified DFS and simulated annealing to explore different search spaces could give some useful results about what kinds of algorithms are most effective in practice at sampling the solutions in a search space.

A

Rules for the Repair Robot Domain

```
((machine-works ?machine)
 (priority-over (machine-slot-full ?a ?b) (machine-slot-empty ?a ?b))
 (or (check (machine-works ?machine))
     (and (forall (?mach-slot)
                  (is-machine-part-slot ?machine ?mach-slot)
                  (machine-slot-full ?machine ?mach-slot))
          (robot-at ?machine)
          (achieve activate-machine ?machine))))
```

Figure A.1: Rule for `machine-works`

```
((robot-at ?machine)
 (priority-over)
 (or (check (robot-at ?machine))
     (exists (?robot-l) (robot-at ?robot-l)
        (achieve travel ?robot-l ?machine))))
```

Figure A.2: Rule for `robot-at`

```
((machine-slot-full ?machine ?slot)
 (priority-over (robot-slot-full ?a ?b) (robot-slot-empty ?a))
 (or (check (machine-slot-full ?machine ?slot))
     (or (and (check (is-storehouse ?machine))
              (get-part-for ?machine ?slot)
              (install-one-part ?machine ?slot))
         (and (check (not (is-storehouse ?machine)))
              (get-part-for ?machine ?slot)
              (get-extra-parts-for ?machine ?slot)
              (get-other-machine-parts ?machine)
              (install-all-parts ?machine)))))
```

Figure A.3: Rule for `machine-slot-full`

```
((machine-slot-empty ?machine ?slot)
 (priority-over (robot-slot-full ?a ?b) (robot-slot-empty ?a))
 (or (check (machine-slot-empty ?machine ?slot))
     (exists (?type) (machine-part-slot-type ?machine ?slot ?type)
        (choose (exists (?robot-slot)
                  (is-robot-part-slot ?robot-slot)
                  (try-first (robot-slot-empty ?robot-slot))
                  (and (robot-slot-empty ?robot-slot)
                       (robot-at ?machine)
                       (achieve get-part-from ?machine ?type
                                ?robot-slot ?slot)))))))
```

Figure A.4: Rule for `machine-slot-empty`

```
((robot-slot-full ?slot ?type)
 (priority-over (robot-at ?a))
 (or (check (robot-slot-full ?slot ?type))
     (choose (exists (?store) (is-storehouse ?store)
        (try-first (robot-at ?store))
        (exists (?store-slot)
           (machine-part-slot-type ?store ?store-slot ?type)
           (and (check (machine-slot-full ?store ?store-slot))
                (robot-at ?store)
                (achieve get-part-from ?store ?type ?slot
                                      ?store-slot)))))))
```

Figure A.5: Rule for `robot-slot-full`

```
((robot-slot-empty ?slot)
 (priority-over (robot-at ?a))
 (or (check (robot-slot-empty ?slot))
     (exists (?type) (robot-slot-full ?slot ?type)
        (or (choose (exists (?machine) (not-machine-works ?machine)
               (and (check (not (is-storehouse ?machine)))
                    (choose (exists (?mach-slot)
                       (machine-part-slot-type ?machine ?mach-slot
                                               ?type)
                       (and (check
                             (and (machine-slot-empty ?machine
                                                      ?mach-slot)
                                  (goal (machine-slot-full ?machine
                                                  ?mach-slot))))
                            (robot-at ?machine)
                            (achieve place-part-at ?machine ?type
                                     ?slot ?mach-slot)))))))
            (choose (exists (?store) (is-storehouse ?store)
               (exists (?store-slot)
                  (machine-part-slot-type ?store ?store-slot ?type)
                  (and (check (machine-slot-empty ?store ?store-slot))
                       (robot-at ?store)
                       (achieve place-part-at ?store ?type ?slot
                                ?store-slot)))))))))
```

Figure A.6: Rule for `robot-slot-empty`

```
((not-machine-works ?machine)
 (priority-over)
 (check (not-machine-works ?machine)))
```

Figure A.7: Rule for `not-machine-works`

```
((can-activate-machine ?machine)
 (priority-over (robot-slot-full ?a ?b) (robot-slot-empty ?a))
 (forall (?mach-slot) (is-machine-part-slot ?machine ?mach-slot)
             (machine-slot-full ?machine ?mach-slot)))
```

Figure A.8: Rule for `can-activate-machine`

```
((get-part-for ?machine ?slot)
 (priority-over)
 (or (tag-added-for ?machine ?slot)
     (choose
        (exists (?robot-slot) (is-robot-part-slot ?robot-slot)
           (try-first (robot-slot-empty ?robot-slot))
           (and (robot-slot-empty ?robot-slot)
                (choose (exists (?store) (is-storehouse ?store)
                   (part-for-slot-gotten ?machine ?slot
                                 ?store ?robot-slot))))))))
```

Figure A.9: Rule for `get-part-for`

```
((get-extra-parts-for ?machine ?slot)
 (priority-over)
 (or (check (is-storehouse ?machine))
     (forall (?other-slot) (machine-slot-empty ?machine ?other-slot)
        (or (check (= ?other-slot ?slot))
            (check (exists (?robot-slot)
                      (robot-part-tagged ?robot-slot ?machine
                                         ?other-slot)))
            (check (goal (machine-slot-empty ?machine ?other-slot)))
            (tag-added-for ?machine ?other-slot)
            (exists (?robot-slot) (robot-slot-empty ?robot-slot)
               (exists (?store) (robot-at ?store)
                  (and (check (is-storehouse ?store))
                       (part-for-slot-gotten ?machine ?other-slot
                                             ?store ?robot-slot))))
            (default)))))
```

Figure A.10: Rule for `get-extra-parts-for`

```
((get-other-machine-parts ?machine)
 (priority-over)
 (forall (?other-mach) (not-machine-works ?other-mach)
    (or (check (= ?other-mach ?machine))
        (forall (?other-slot)
            (machine-slot-empty ?other-mach ?other-slot)
            (or (check
                  (goal (machine-slot-empty ?other-mach ?other-slot)))
                (check (and (is-storehouse ?other-mach)
                            (not (goal (machine-slot-full
                                        ?other-mach ?other-slot)))))
                (tag-added-for ?other-mach ?other-slot)
                (choose
                   (or (default)
                       (exists (?robot-slot)
                          (robot-slot-empty ?robot-slot)
                          (exists (?store) (robot-at ?store)
                             (and (check (is-storehouse ?store))
                                  (part-for-slot-gotten ?other-mach
                                        ?other-slot ?store
                                        ?robot-slot))))))
                (default))))))
```

Figure A.11: Rule for `get-other-machine-parts`

```
((install-one-part ?machine ?mach-slot)
 (priority-over)
 (and (robot-at ?machine)
      (exists (?type)
         (machine-part-slot-type ?machine ?mach-slot ?type)
         (or (exists (?robot-slot) (robot-slot-full ?robot-slot ?type)
                (and (achieve place-part-at ?machine ?type ?robot-slot
                                ?mach-slot)
                     (del-from-state
                         (robot-part-tagged ?robot-slot ?machine
                                ?mach-slot))))
             (default)))))
```

Figure A.12: Rule for `install-one-part`

```
((install-all-parts ?machine)
 (priority-over)
 (and (robot-at ?machine)
      (forall (?mach-slot) (machine-slot-empty ?machine ?mach-slot)
         (install-one-part ?machine ?mach-slot))))
```

Figure A.13: Rule for `install-all-parts`

```
((part-for-slot-gotten ?machine ?slot ?store ?robot-slot)
 (priority-over)
 (exists (?type) (machine-part-slot-type ?machine ?slot ?type)
    (exists (?store-slot)
       (machine-part-slot-type ?store ?store-slot ?type)
       (and (check (machine-slot-full ?store ?store-slot))
            (robot-at ?store)
            (achieve get-part-from ?store ?type ?robot-slot
                     ?store-slot)
            (add-to-state (robot-part-tagged ?robot-slot ?machine
                           ?slot))))))
```

Figure A.14: Rule for `part-for-slot-gotten`

```
((tag-added-for ?machine ?slot)
 (priority-over)
 (exists (?type) (machine-part-slot-type ?machine ?slot ?type)
   (exists (?robot-slot) (robot-slot-full ?robot-slot ?type)
      (and (check
              (not (exists (?other-mach) (is-machine ?other-mach)
                      (and (not (= ?other-mach ?machine))
                           (exists (?other-slot)
                              (robot-part-tagged ?robot-slot
                                 ?other-mach ?other-slot))))))
           (or (check (robot-part-tagged ?robot-slot ?machine ?slot))
               (and (check
                       (not (exists (?other-slot)
                               (machine-slot-empty ?machine
                                                   ?other-slot)
                               (and (not (= ?other-slot ?slot))
                                    (robot-part-tagged ?robot-slot
                                       ?machine ?other-slot)))))
                   (add-to-state (robot-part-tagged ?robot-slot
                                    ?machine ?slot))))))))
```

Figure A.15: Rule for `tag-added-for`

B

Maps, World States, and Failures

This appendix describes the maps used, the initial states for each of those maps, and the failure sequences used with each map.

B.1 Map Layout

The geometric configuration of each map is given in the corresponding figure. In each map, circles are machines and squares are storehouses. Figure B.1 shows the layout of Map 1. The layout of machines and storehouses in Map 1 is a regular polygon. Figure B.2 shows Map 2, and Figure B.3 shows Map 3. The coordinates of each location in Map 2 and Map 3 were generated pseudo-randomly.

Experiments were performed using two versions of each map. In the first version, each machine required two machine parts of type 1 and one part of type 2. Storehouse 11 contained 20 type 1 parts, storehouse 12 contained 10 type 2 parts, and storehouse 13 contained 10 type 1 parts and 5 type 2 parts. In all of the first version experiments, the machines produced the following numbers of widgets per time step:

1. 2

Figure B.1: Map 1

Figure B.2: Map 2

Figure B.3: Map 3

2. 6
3. 10
4. 2
5. 6
6. 10
7. 2
8. 6
9. 10
10. 2

In the second version, each machine required six partsand there were three total types of parts. Different machines had different part type requirements. Here are the requirements for each machine/storehouse:

1. 4 type 1, 2 type 2
2. 2 type 1, 4 type 2
3. 1 type 1, 1 type 2, 4 type 3
4. 2 type 1, 4 type 3
5. 3 type 2, 3 type 3
6. 2 type 1, 1 type 2, 3 type 3
7. 2 type 1, 4 type 2

8. 2 type 1, 1 type 2, 3 type 3
9. 2 type 1, 3 type 2, 1 type 3
10. 2 type 1, 1 type 2, 3 type 3
11. 20 type 1, 12 type 3
12. 14 type 2, 12 type 3
13. 10 type 1, 7 type 2

In the second version, each machine produced the following number of widgets per time step:

1. 6
2. 10
3. 2
4. 6
5. 10
6. 2
7. 6
8. 10
9. 2
10. 6

B.2 Initial States

For each number of part types, there were two initial states from which original plans were generated. The variation in initial state regarded what parts were lacking from what machines. The storehouses were always completely full. Here are the initial numbers of parts per machine, and whether the machine was functioning:

Seven machines broken, 2 part types:

1. 1 type 1, 1 type 2, not functioning
2. 2 type 1, 1 type 2, functioning
3. No parts, not functioning
4. 2 type 1, 1 type 2, functioning
5. No parts, not functioning
6. 2 type 1, 1 type 2, functioning
7. No parts, not functioning
8. 1 type 2, not functioning
9. 1 type 1, 1 type 2, not functioning
10. 1 type 1, not functioning

Seven machines broken, 3 part types:

1. 3 type 1, 2 type 2, not functioning
2. 2 type 1, 4 type 2, functioning

3. 3 type 3, not functioning
4. 2 type 1, 4 type 3, functioning
5. 1 type 2, 2 type 3, not functioning
6. 2 type 1, 1 type 2, 3 type 3, functioning
7. 3 type 2, not functioning
8. 1 type 1, 1 type 2, 2 type 3, not functioning
9. 2 type 1, 3 type 2, not functioning
10. 1 type 1, 1 type 2, 2 type 3, not functioning

For these cases, no machines were functioning initially; just the number of working parts of each type per machine is listed.

Ten machines broken, 2 part types:

1. 1 type 1, 1 type 2
2. 1 type 2
3. No functioning parts
4. 1 type 1
5. No functioning parts
6. No functioning parts
7. No functioning parts
8. 1 type 2

9. 1 type 1, 1 type 2
10. 1 type 1

Ten machines broken, 3 part types:

1. 1 type 1
2. 1 type 1
3. 1 type 3
4. No functioning parts
5. 1 type 2
6. 1 type 1
7. 2 type 2
8. 1 type 1, 1 type 2, 1 type 3
9. 1 type 2
10. 1 type 1, 1 type 2, 1 type 3

As each of the four initial machine configurations was applied to each of the three maps, twelve total initial plans were created.

B.3 Failure Scripts

A failure script contains a list of failures that will occur as the plan is executed. Each failure has two components. The first component indicates when the failure occurs and the second component indicates what changes to the world state result from the occurrence of the failure. The timing of the failure is specified in terms of the occurrence of a particular operator. For example, a failure can happen before the second time an `activate-machine` operator gets applied. The changes to the world state are given in terms of add and delete lists. The predicates added and deleted are specified in terms of parameters shared with the operator used to specify failure timing. In my experiments, failures consisted of the destruction of parts in a machine immediately before the robot attempts to use the `activate-machine` operator to restart production.

Figure B.4 contains an example of how a failure sequence was specified. Each `Before` clause indicates a failure to occur before the application of the specified operator. The number refers to the *nth* application of that operator since the last failure. In this example, the first failure would be applied before the first machine activation in the initial plan. The second failure would be before the third machine activation in the plan generated in order to handle the first failure.

Each initial plan generated to fix seven machines was subjected to three distinct failure sequences; each initial plan generated to fix all of the machines was subjected to four. Each sequence was specified using a failure script similar to the one depicted in Figure B.4. The sequences for the short initial plans were:

- 1, 2, 2
- 1, 3, 3
- 1, 4, 1

The sequences for the long initial plans were:

- 1, 2, 2, 2, 2

```
((Before 1 (activate-machine ?mach)
  ((add ((machine-slot-empty ?mach 0) (machine-slot-empty ?mach 1)
         (not-machine-works  ?mach)))
   (del ((machine-slot-full  ?mach 0) (machine-slot-full  ?mach 1)
         (machine-works ?mach)))))
 (Before 3 (activate-machine ?mach)
  ((add ((machine-slot-empty ?mach 1) (machine-slot-empty ?mach 2)
         (not-machine-works  ?mach)))
   (del ((machine-slot-full  ?mach 1) (machine-slot-full  ?mach 2)
         (machine-works ?mach)))))
 (Before 3 (activate-machine ?mach)
  ((add ((machine-slot-empty ?mach 2) (machine-slot-empty ?mach 0)
         (not-machine-works  ?mach)))
   (del ((machine-slot-full  ?mach 2) (machine-slot-full  ?mach 0)
         (machine-works ?mach))))))
```

Figure B.4: A Failure Script

- 1, 4, 4
- 1, 5, 3
- 1, 6

For each map, 17 different specific failure patterns were given for each failure sequence. Seven of the patterns applied to machines with three parts each; the other ten patterns applied to machines with six parts each. For each sequence, there were six patterns in which one part failed per machine, six patterns with two parts failing, four patterns with three parts failing, and one pattern with six parts failing.

Bibliography

[1] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning.* Morgan Kaufman, 1990.

[2] Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence*, 118:115–161, 2000.

[3] Jose Luis Ambite and Craig A. Knoblock. Planning by rewriting. *Journal of Artificial Intelligence Research*, 15:207–261, 2001.

[4] Jose Luis Ambite, Craig A. Knoblock, and Steven Minton. Learning plan rewriting rules. In *Proceedings of the Fifth Artificial Intelligence and Planning Symposium (AIPS-2000)*, Breckenridge, Colorado, 2000.

[5] Ronald C. Arkin and Tucker Balch. AuRA: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2), 1997.

[6] Fahiem Bacchus. Aips 2000 planning competition, 2000. http://www.cs.toronto.edu/aips2000/.

[7] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in Planning.* IOS Press, 1996.

[8] Christer Backstrom and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4), 1995.

[9] A. L. Blum and M Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[10] Jim Blythe and W. Scott Reilly. Integrating reactive and deliberative planning for agents. Technical Report CMU-CS-93-155, Carnegie Mellon University, School of Computer Science, May 1993.

[11] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of IJCAI-1989*, pages 979–984, 1989.

[12] Mark Boddy and Thomas L. Dean. Deliberation scheduling for problem solving in time-constrainted environments. *Artificial Intelligence*, 67:245–285, 1994.

[13] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2), 1997.

[14] Will Briggs and Diane Cook. Anytime planning for optimal tradeoff between deliberative and reactive planning. In *Proceedings of the 1999 Florida AI Research Symposium (FLAIRS-99)*, 1999.

[15] Frank Z. Brill, Glenn S. Wasson, Gabriel J. Ferrer, and Worthy N. Martin. The effective field of view paradigm: Adding representation to a reactive system. *Engineering Applications of Artificial Intelligence*, 11:189–201, 1998.

[16] Rodney A. Brooks. A robost layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[17] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[18] David Chapman. Planning for conjuncitve goals. *Artificial Intelligence*, 32:333–377, 1987.

[19] S. Chien, A. Barrett, T. Estlin, and G. Rabideau. A comparison of coordinated planning methods for cooperating rovers. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000)*, Barcelona, Spain, June 2000.

[20] Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, Breckenridge, Colorado, April 2000.

[21] Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[22] Thomas Dean, James Allen, and Yiannis Aloimonos. *Artificial Intelligence: Theory and Practice*. Addison-Wesley Publishing Company, 1995.

[23] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of AAAI-1988*, pages 49–54, 1988.

[24] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

[25] Marie E. desJardins, Edmund H. Durfee, Jr. Charles L. Ortiz, and Michael J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 21(4), Winter 2000.

[26] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnstrom. (TAL) Temporal Action Logics: Language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2(3-4):273–306, 1998.

[27] Brian Drabble, Jeff Dalton, and Austin Tate. Repairing plans on-the-fly. In *Proceedings of the NASA Workshop on Planning and Scheduling for Space*, Oxnard, California, USA, October 1997.

[28] Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of AAAI-90*, July-August 1990.

[29] Mark Drummond, John Bresina, and Keith Swanson. Just-in-case scheduling. In *Proceedings of AAAI-1994*, Seattle, WA, 1994.

[30] Charles Elkan. Incremental, approximate planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-1990)*, pages 145–150, 1990.

[31] Chris Elsaesser and Richard MacMillan. Representation and algorithms for multiagent adversarial planning. Technical Report MTR-91W000207, The MITRE Corporation, December 1991.

[32] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of AAAI-94*, July 1994.

[33] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability, and undecidability results for domain-independent planning. *Artificial Intelligence*, 76:75–88, 1995.

[34] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 189–208, Imperial College, London, England, September 1971.

[35] R. J. Firby, P. Prokopowicz, and M. Swain. Plan representations for picking up trash. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, Montreal, Canada, 1995.

[36] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, August 1992.

[37] Peter Haddawy. Focusing attention in anytime decision-theoretic planning. *SIGART Bulletin*, 7(2), Summer 1996.

[38] Karen Zita Haigh and Manuela M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In *Proceedings of the First International Conference on Autonomous Agents*, pages 363–370, February 1997.

[39] Jorg Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, Washington, August 2001.

[40] Jorg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, May 2001.

[41] Ian Horswill. Polly: A vision-based artificial agent. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993.

[42] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 318–325, Stockholm, Sweden, July/August 1999.

[43] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[44] Jonas Kvarnstrom, Patrick Doherty, and Patrik Haslum. Extending TALplanner with concurrency and resources. In *ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence*, Amsterdam, 2000.

[45] J. E. Laird and P. S. Rosenbloom. Integrating execution, planning, and learning in soar for external environments. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1022–1029, 1990.

[46] Drew McDermott. Aips 1998 planning competition, 1998. ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html.

[47] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 968–973, Stockholm, Sweden, July/August 1999.

[48] Dana Nau, Hector Munoz-Avila, Yue Cau, Amnon Lotem, and Steven Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, Seattle, Washington, August 2001.

[49] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76:427–454, 1995.

[50] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First Conference on Principles of Knowledge Representation and Reasoning (KR 89)*, pages 324–332. Morgan Kaufman Publisher Inc., 1989.

[51] Gregg Rabideau, Russell Knight, Steve Chien, Alex Fukunaga, and Anita Govindjee. Iterative repair planning for spacecraft operations using the aspen system. In *Proceedings of the International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)*, Noordwijk, The Netherlands, June 1999.

[52] Daniel Ratner and Ira Pohl. Joint and LPA*: Combination of approximation and search. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-1986)*, pages 173–177, 1986.

[53] Paul S. Rosenbloom, John E. Laird, and Allen Newell, editors. *The Soar Papers: Research on Integrated Intelligence*. MIT Press, 1993.

[54] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[55] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.

[56] S.J. Smith, Dana Nau, and T. Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[57] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[58] Austin Tate, James Hendler, and Mark Drummond. A review of AI planning techniques. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 26–49, 1990.

[59] Manuela Veloso, Jaime Corbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

[60] David E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.

[61] David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):197–227, 1995.

[62] Qiang Yang, Dana S. Nau, and James Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(2):648–676, February 1992.

[63] Shlomo Zilberstein and Stuart J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of IJCAI-1993*, pages 1402–1407, 1993.

[64] Shlomo Zilberstein and Stuart J. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.