On the role of Disjunctive Representations and Constraint Propagation in Refinement Planning

Subbarao Kambhampati and Xiuping Yang
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287.
Email: { rao, xiuping.yang }@asu.edu
WWW: http://rakaposhi.eas.asu.edu/yochan.html

Abstract

Most existing planners intertwine the refinement of a partial plan with search by pushing the individual refinements of a plan into different search branches. Although this approach reduces the cost of handling partial plans, it also often leads to search space explosion. In this paper, we consider the possibility of handling the refinements of a partial plan together (without splitting them into search space). This is facilitated by disjunctive partial plan representations that can compactly represent large sets of partial plans. Disjunctive representations have hitherto been shunned since they may increase the plan handling costs. We argue that performance improvements can be obtained despite these costs by the use of (a) constraint propagation techniques to simplify the disjunctive plans and (b) CSP/SAT techniques to extract solutions from them. We will support this view by showing that some recent promising refinement planners, such as the GRAPHPLAN algorithm [2], can be seen as deriving their power from disjunctive plan representations. We will also present a new planning algorithm, UCPOP-D, which uses disjunctive representations over UCPOP [19] to improve performance. Finally, we will discuss the issues and tradeoffs involved in planning with disjunctive representations.

1 Introduction

A large part of the work on plan synthesis in artificial intelligence falls under the rubric of refinement planning. Refinement planning [12] involves manipulating sets of partial plans, each of which are shorthand notations for a set of potential solutions for the planning problem (called the candidate set of the plan). The planning process starts with the null plan, corresponding to the set of all action sequences, and consists of two repeated steps. First, called the solution-extraction phase, involves examining current set of partial plans to see if they contain a solution. If this step fails, then a "refinement strategy" is applied to the current set of plans to get a new plan set. Informally, refinement strategies can be understood as operations that narrow the candidate set by eliminating those sequences that cannot be solutions to the problem.

As described above, refinement planning does not need any explicit search. However, most refinement planners described in the literature, including the popular ones like UCPOP [19] and PRODIGY [4] introduce explicit search into the refinement process by considering each of the refinements of a plan in a different search branch. The motivation behind introducing search into refinement planning is to restrict the solution extraction and refinement operations to single plans, thereby making them cheaper. The expense that all these planners pay for this reduction in "per-plan" cost is the increase in search space size. Indeed, it is well known that planners such as UCPOP and PRODIGY generate very large search spaces even for simple problems [7, 2]. The usual solution to this problem is to control the planner's search with the help of search control knowledge acquired from domain experts (e.g. task reduction schemas) or through learning techniques (e.g. explanation-based learning [15], case-based planning [6]).

In this paper, we will consider a more direct solution to the search space explosion problem – that of handling sets of plans without splitting them into the search space. At first glance, this seems to involve a mere exchange of complexity from search space size to solution extraction cost. In particular, handling sets of plans together might lead to unwieldy data structures, as well as a costly solution-extraction process. We will argue that we can nevertheless derive performance improvements by the use of disjunctive partial plan representations, which support compact representation of large sets of partial plans, constraint propagation techniques which simplify the partial plan constraints, and the use of efficient CSP/SAT solvers to help in solution extraction.

The paper is organized as follows. Section 2 provides the background on the planning problem, and syntax and semantics of partial plans, which can be skipped by read-

ers familiar with candidate set semantics for partial plans (c.f. [12]). The next two sections provide a novel view of the notions of refinement strategies and refinement planning which shows the secondary nature of search in refinement planning. Section 5 discusses how disjunctive representations and constraint propagation can be used to efficiently handle sets of plans together, without splitting them into the search space. It also explains the success of GRAPHPLAN algorithm in terms of these ideas. Section 6 shows how these ideas can be used to improve the performance of traditional refinement planners by presenting UCPOP-D algorithm which uses disjunctive representations to improve the performance of UCPOP. This section also describes empirical results demonstrating the potential of UCPOP-D, and the ways in which it can be extended. Section 7 discusses the tradeoffs in planning with disjunctive representations and Section 8 discusses the relations with other recent efforts to scale up AI planning techniques. Section 9 summarizes the contributions of this paper.

2 Preliminaries

A planning problem is defined in terms of the initial and goal states of the world, and a set of actions. The world states are represented in terms of some binary state variables, and the actions transform a given state into another. Actions are described in terms of preconditions (i.e., the specific state variable/value configuration that must hold in the world state for them to be applicable) and effects (i.e., the changes that the action would make to state variable values to give rise to the new world state). A solution to the planning problem is a sequence of actions that when executed from the initial state, results in a state where the state variables mentioned in the goal state have the specified values.

As an example, consider the simple one-way rocket domain, consisting of a single rocket, and two packages, A and B, all of which are initially on earth. The objective is to send both the packages to moon. There are three actions, $Load(x)$ which puts the package x in the rocket, $Unload(x)$ which takes the package out of the rocket (and on to earth or moon, whereever the rocket may be at that time). Finally, there is the $Fly()$ action which transports the rocket, along with its contents, from earth to moon. The preconditions and effects of $Fly()$ action are specified as follows:

Fly()
Preconditions: $At(R, E)$
Effects: $At(R, M)$
$\forall_x In(x) \Rightarrow [At(x, M), \neg At(x, E)]$

The initial state is specified as $At(A, E) \wedge At(B, E) \wedge At(R, E)$ and the goal state is specified by $At(A, M) \wedge At(B, M) \wedge \neg In(A) \wedge \neg In(B)$.

Refinement planners [12] attempt to solve a planning problem by navigating the space of *sets of potential solutions (action sequences)*. The potential solution sets are represented and manipulated in the form of "(partial) plans."

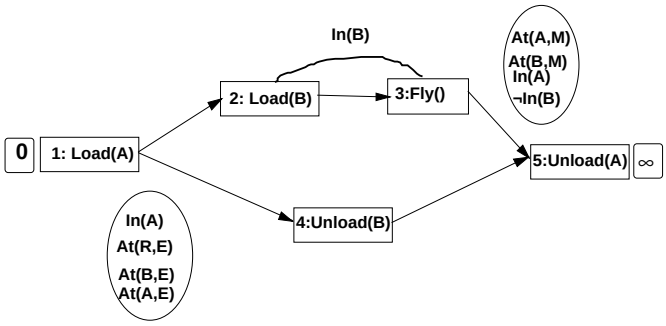

Figure 1: This figure depicts the partial plan from the rocket domain. The ordering constraints between steps are shown by arrows. The interval preservation constraints are shown by arcs. Contiguous steps are shown immediately next to each other. States of the world after the prefix and before the suffix of the partial plan are shown in the ovals beside them.

Syntactically, a partial plan $\mathcal{P}$ can be seen as a set of constraints. Semantically, a partial plan is a shorthand notation for the set of action sequences that are consistent with its constraints. The set of such action sequences is called the set of candidates (or the **candidate set**) of the partial plan $\mathcal{P}$ and is denoted as $\langle\langle \mathcal{P} \rangle\rangle$. Informally, refinements narrow the candidate sets of partial plans by gradually eliminating action sequences that cannot be solutions to the problem. To make this more precise, we need to talk about the syntax and semantics of the partial plans.

2.1 Partial Plans: Syntax

The following representation of partial plans is a generalization of representations used in several existing planning algorithms, and shows the types of constraints normally used.[1] A *partial plan* consists of a set of *steps*, a set of *ordering constraints* that restrict the order in which steps are to be executed, and a set of *auxiliary constraints* that restrict the value of state variables (describing the states of the world) over particular points or intervals of time. Each step is associated with an action. To distinguish between multiple instances of the same action appearing in a plan, we assign to each step a unique step number i and represent the ith step as the pair $i : A_i$ where A_i is the action associated with the ith step. The step 0 corresponds to a dummy action symbolizing the beginning of the plan, and the step ∞ corresponds to the dummy action symbolizing the end of the plan. The conditions true in the initial state are considered the effects of 0 and the conditions needed in the goal state are considered the preconditions of ∞. Figure 1 shows a partial plan $\mathcal{P}_{eg}$ consisting of seven steps (including 0 and ∞). The plan $\mathcal{P}_{eg}$ is represented as follows:

[1] For a very different partial plan representation, that still has candidate set based semantics, see Ginsberg's paper in these proceedings [5].

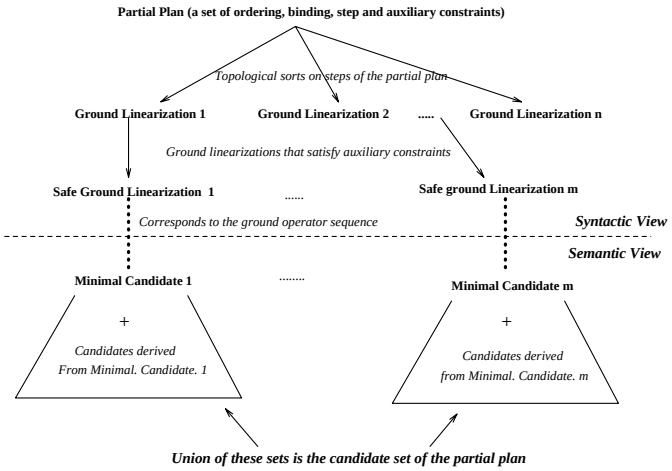

Figure 2: Relations between a partial plan and its candidate set.

$$\left\langle \begin{array}{c} \{(1:Load(A)),(2:Load(B)),(3:Fly()), \\ (4:Unload(B)),(5:Unload(A)),\}, \\ \{(0*1),(1\prec 2),(1\prec 4),(2\prec 3), \\ (3\prec 5),(4\prec 5),(5*\infty)\}, \\ \{(2 \overset{In(B)}{-} 3)\} \end{array} \right\rangle$$

An ordering constraint of the form $(i \prec j)$ indicates that Step i precedes Step j. An ordering constraint of the form $(i * j)$ indicates that Step i is contiguous with Step j, that is Step i precedes Step j and no other steps intervene. When steps are contiguous to 0, we can completely predict the state of the world after those steps. In the example plan, the state of the world after step 1 is shown in the oval below the plan. Similarly, we can predict the minimal requirements on the state of the world before a step that is contiguous to ∞.

An auxiliary constraint of the form $(i \overset{P}{-} j)$ is called an *interval preservation constraint* and indicates that P is to be preserved in the range between Steps i and j (and therefore no operator with postcondition $\neg P$ should occur between Steps i and j). In particular, according to the constraint $(2 \overset{In(B)}{-} 3)$, Step 4 should not occur between Steps 2 and 3. An auxiliary constraint of the form $P@s$ is called the *point truth constraint* (PTC), and requires that the condition P be true in the state in which s is executed. A partial plan containing a step $s:A$ will have PTCs corresponding to all the preconditions of A. In addition, there may be PTCs corresponding to secondary preconditions of A posted to make A either preserve or cause some required condition.

2.2 Partial Plans: Semantics

The semantic status of a partial plan constraint is clarified by specifying when a given action sequence is said to satisfy the constraint. In particular, an action sequence is a candidate of a partial plan if it contains actions corresponding to all the steps of the plan, in an order consistent with the precedence and contiguity constraints, and it satisfies all the interval preservation constraints [12]. Figure 2 shows the schematic relations between a partial plan and its candidate set [12], and we will illustrate these relations with respect to the example plan in Figure 1. Each partial plan corresponds to a set of topological sorts (e.g. $\langle 1,2,3,4,5\rangle$ and $\langle 1,2,4,3,5\rangle$). The subset of these that satisfy the auxiliary constraints of the plan (e.g. $\langle 1,2,3,4,5\rangle$) are said to be the safe-ground linearizations of the plan. ($\langle 1,2,4,3,5\rangle$ is not a safe linearization since $4:Unload(B)$ violates the interval preservation constraint $(2 \overset{In(B)}{-} 3)$). Each safe ground linearization of the plan corresponds to an action sequence which is a minimal candidate of the partial plan (e.g. $\langle Load(A), Load(B), Fly(), Unload(B), Unload(A)\rangle$). An infinite number of additional candidates can be derived from each minimal candidate of the plan by augmenting (padding) it with additional actions without violating the auxiliary constraints (e.g. $Load(A), Unload(A)$,$Load(A)$, $Load(B)$, $Fly()$,$Unload(B)$, $Unload(A)$).

Thus, the candidate set of a partial plan is infinite, but the set of its minimal candidates is finite. The solution extraction involves searching the minimal candidates of the plan to see if any of them are solutions to the planning problem. The process of refinement can be understood as incrementally increasing the size of these minimal candidates so that action sequences of increasing lengths are examined to see if they are solutions to the problem.

3 Refinement Strategies

Refinement strategies narrow the candidate sets of partial plans by adding constraints to eliminate action sequences that cannot be solutions. They differ based on the types of constraints they add (and thus the type of action sequences they eliminate). They are best seen as mapping a set of partial plans to another set of partial plans. Since partial plans represent sets of action sequences, all set theoretic operations – union, intersection etc. – are well defined for them. In particular we will define a **planset** $\widehat{\mathcal{P}}$ to be a set of one or more partial plans $\{\mathcal{P}_1, \mathcal{P}_2 \ldots \mathcal{P}_n\}$. The P_i are called the components of $\widehat{\mathcal{P}}$. The candidate set of $\widehat{\mathcal{P}}$ is defined as the union of the candidate sets of the partial plans in that plan set (a single plan can be seen as a singleton planset). Thus

$$\langle\!\langle \widehat{\mathcal{P}} \rangle\!\rangle \triangleq \langle\!\langle \mathcal{P}_1 \rangle\!\rangle \cup \langle\!\langle \mathcal{P}_2 \rangle\!\rangle \cup \ldots \cup \langle\!\langle \mathcal{P}_n \rangle\!\rangle$$

A planset is said to be *irredundant* if the candidate sets of component plans are all disjoint.[2] We will also denote the set of all action sequences that can be executed from the initial

[2]The differentiation between plans and plansets is an artificial one made for convenience. A planset in one plan language may be a single plan in another plan language. In fact, we shall see below that a set of partial plans can be represented compactly as a single plan containing disjunctive step, ordering and binding constraints.

state I and satisfy the goals G as $\mathcal{L}(I,G)$. The objective of a planning algorithm is to be able to return some desired element of $\mathcal{L}(I,G)$.

A refinement strategy $\mathcal{R}$ is an operation which takes a planset $\widehat{\mathcal{P}}$ and returns a new planset $\widehat{\mathcal{P}}_{new}$ such that $\langle\!\langle \widehat{\mathcal{P}}_{new} \rangle\!\rangle$ is a *subset* of $\langle\!\langle \widehat{\mathcal{P}} \rangle\!\rangle$. $\mathcal{R}$ is said to be a **progressive** refinement if $\langle\!\langle \widehat{\mathcal{P}}_{new} \rangle\!\rangle$ is a proper subset of $\langle\!\langle \widehat{\mathcal{P}} \rangle\!\rangle$. $\mathcal{R}$ is said to be a **complete** refinement if all solutions of $\widehat{\mathcal{P}}$ are also solutions of $\widehat{\mathcal{P}}_{new}$ (i.e., $\langle\!\langle \widehat{\mathcal{P}} \rangle\!\rangle \cap \mathcal{L}(I,G) \equiv \langle\!\langle \widehat{\mathcal{P}}_{new} \rangle\!\rangle \cap \mathcal{L}(I,G)$). $\mathcal{R}$ is said to be **systematic** if given an irredundant planset, it produces another irredundant planset.

Intuitively, a complete and progressive refinement strategy narrows down the candidate set of a planset by winnowing out action sequences that cannot be solutions to the problem. Thus, when we refine the null plan $\mathcal{P}_\emptyset$ repeatedly, we get a sequence of plan sets $\widehat{\mathcal{P}}_i$ which satisfy:[3]

$$\mathcal{U} \equiv \langle\!\langle \mathcal{P}_\emptyset \rangle\!\rangle \supset \langle\!\langle \widehat{\mathcal{P}}_2 \rangle\!\rangle \supset \langle\!\langle \widehat{\mathcal{P}}_2 \rangle\!\rangle \ldots \langle\!\langle \widehat{\mathcal{P}}_n \rangle\!\rangle \ldots \supset \mathcal{L}(I,G)$$

Examples of Refinement Strategies: As shown in [13], existing planners use three types of complete and progressive refinement strategies – forward state space refinement (FSR), backward state space refinement (BSR) and plan space refinement (PSR). Informally, FSR can be understood as eliminating action sequences that have unexecutable prefixes, and BSR can be understood as eliminating action sequences that have infeasible suffixes (in that not all goals can hold at the end of the suffix). Finally, PSR can be understood as eliminating action sequences that do not contain relevant actions. All of them introduce new action constraints onto the plansets and thus increase the length of their minimal candidates. In addition, FSR and BSR add contiguity constraints between actions – thus fixing their relative positions, and giving state information, while PSR adds precedence constraints, fixing only the relative ordering between steps.

In the rocket domain, The FSR refinement takes the singleton planset containing the null plan $\mathcal{P}_\emptyset : \langle 0 \prec \infty \rangle$ and gives the planset:

$$\left\{ \begin{array}{c} \langle 0 * load(A) \prec \infty \rangle, \\ \langle 0 * load(B) \prec \infty \rangle, \\ \langle 0 * Fly() \prec \infty \rangle \end{array} \right\}$$

since only the actions $load(A)$, $load(B)$ and $Fly()$ are applicable in the initial state. It is easy to see that FSR refinement strategy is complete, since every solution to the problem must start with one of the applicable actions. The refinement is progressive since all action sequences belonging to the candidate set of $\mathcal{P}_\emptyset$ which start with unexecutable actions are eliminated after the refinement. Finally, the refinement is systematic since the candidates of the different plansets start with different initial actions. BSR is similar to FSR, except that it extends the suffix of the plan by considering all actions that are applicable in the backward direction.

Finally, PSR refines a planset by "establishing" preconditions of steps in the component plans. Specifically, it picks a precondition C of some step s of a component plan P_i in the given planset $\widehat{\mathcal{P}}$. It then returns a new planset $\widehat{\mathcal{P}}_{new}$ in which P_i is replaced by a set of plans each corresponding to a different way of "establishing" the condition C. In the rocket domain, if we apply PSR refinement to the null plan $\mathcal{P}_\emptyset$, to support the top level goal $At(A,M)$, we generate the singleton planset:

$$\left\{ \left\langle \begin{array}{c} 0 \prec 1 : Fly() \prec \infty, \\ In(A)@1, \\ (1 \overset{At(A,M)}{-} \infty), (1 \overset{\neg At(A,M)}{-} \infty), \end{array} \right\rangle \right\}$$

The point truth constraint $In(A)@1$ is added to force the $Fly()$ action to give the effect $At(A,M)$. The two optional interval preservation constraints are added to ensure that this establishment will not be violated, or repeated, by the actions introduced by later refinements. Once again, it is easy to see that PSR refinement is progressive in that we have eliminated all action sequences that do not contain $Fly()$ action, and complete in that every solution to the problem must satisfy the constraints of the plan shown here.

4 Refinement Planning

The operation of a refinement planner can be understood, broadly, as starting with the null plan $\mathcal{P}_\emptyset$ and repeatedly narrowing down the candidate set by the application of refinement strategies. Figure 3 shows a general refinement planning procedure, Refine($\mathcal{P}$). We will now discuss the development of this procedure.

The simplest case of refinement planning algorithm consists of the procedure in Figure 3, sans the optional steps. This algorithm first checks to see if the current planset is consistent. The second step checks for termination by examining the minimal candidates of the planset to see if any of them correspond to solutions to the problem. Since there are at most exponential number of minimal candidates (corresponding to the safe ground linearizations) for each component of the planset, and since we can check if an action sequence is a solution in linear time, the solution extraction process can be cast as a combinatoric search problem, such as CSP or SAT [9]. The length of the minimal candidates of a plan increase as refinements are applied to it, thus allowing for an incremental exploration of the candidates.

The third step involves refining the planset to generate a new planset. As long as the refinements are complete and progressive, termination is ensured for any solvable problem

[3] Thus, complete and progressive refinements can be seen as computing increasingly finer upper bounds on $L(I,G)$. In [5], Ginsberg presents an novel refinement strategy that simultaneously computes increasingly finer *lower bounds* on $L(I,G)$. In such cases, as soon as the lower bound is non-empty, we can terminate with any of its minimal candidates.

Refine($\widehat{\mathcal{P}}$:Planset)

0. Consistency Check: If $\widehat{\mathcal{P}}$ has no minimal candidates (i.e., safe linearizations), fail.

1. Solution Extraction:
If an action sequence $\langle A_1, A_2, \ldots, A_n \rangle$ is a minimal candidate of $\widehat{\mathcal{P}}$ and also solves the planning problem, terminate and return the action sequence. (Can be cast as a CSP/SAT problem).

2. Refinement: Select refinement R and apply it to $\widehat{\mathcal{P}}$ to get a refined planset $\widehat{\mathcal{P}}_{new}$.

3. Planset splitting (search): (optional) Select a number $k \geq 1$ such that k is less than or equal to the number of components in $\widehat{\mathcal{P}}_{new}$. Split $\widehat{\mathcal{P}}_{new}$ into k plansets $\widehat{\mathcal{P}}_1, \widehat{\mathcal{P}}_2 \ldots \widehat{\mathcal{P}}_k$. *Nondeterministically* select a planset $\widehat{\mathcal{P}}_i$ and set it to $\widehat{\mathcal{P}}_{new}$.

4. Planset simplification (constraint propagation)
(optional) Simplify $\widehat{\mathcal{P}}_{new}$ by enforcing local consistency among constraints. If the simplification shows an inconsistency, then eliminate the plan from further consideration.

5. Recursive invocation: Call $Refine(\widehat{\mathcal{P}}_{new})$.

Figure 3: General Template for Refinement Planning. With k set to the number of components in the planset, we have refinement planning with complete search, as done in most existing planners. With k set to 1, we have refinement planning without search, as is done in GRAPHPLAN, SATPLAN etc. UCPOP-D described in this paper corresponds to a value of k between 1 and the number of planset components.

(in that we will ultimately produce a planset one of whose minimal candidates correspond to solutions).

Introducing search into refinement planning: Most refinement planners augment the pure refinement planning procedure by introducing explicit search into the process. This is done by splitting the components of the plansets into the search space, so that individual components can be refined one by one. In general, splitting the planset components into the search space trades off the search space size increase against the reduction in the solution extraction process. This splitting operation is so prevalent in refinement planners that many previous accounts of refinement planning (including our own [12]) considered the splitting to be a requirement of the refinement planning. As the foregoing discussion shows, this is not necessary.

Controlled search through controlled splitting: Although introducing search in refinement planning reduces the cost of solution extraction function, it does so at the expense of increased search space size. In a way, splitting all the components of a planset into different search branches is an extreme approach for taming the cost of solution extraction. A better solution is to be more deliberate about the splitting.

The algorithm template in Figure 3 supports this by allowing any arbitrary amount of splitting. Specifically, the plan set after refinement, $\widehat{\mathcal{P}}_{new}$, can be split such that some subset of its components are considered together in each search branch.

Handling plansets without splitting: In the foregoing, we have seen that search is introduced into refinement planning by splitting the plansets, and it can be eliminated by handling plansets together. Keeping plansets together and searching for solutions among their minimal candidates also supports a clean separation of planning and scheduling – with refinement strategies doing the bulk of action selection and the CSP/SAT techniques doing the bulk of action sequencing.

There are of course several concerns regarding handling plansets without splitting. The first is that this can lead to very unwieldy data structures. This concern can be alleviated by the use of *disjunctive* plan representations, which allow us to represent a planset containing multiple components by a single partial plan with disjunctive step, ordering, and auxiliary constraints. This representational transformation is best understood as converting external disjunction (among the components of a planset) into internal disjunction (among the constraints of a plan).

The second concern is that avoiding splitting of plansets may just transfer complexity from search space size to plan-handling cost, and thus may not lead to overall improvements in performance in the worst case. We may still hope to win on average for two reasons. First, the CSP and SAT algorithms, which can be used to extract solutions from plansets, seem to scale up much better in practice than general state space search [18, 3], thus encouraging the idea of pushing the complexity into solution extraction phase. Second, and perhaps more important, we can reduce the plan handling costs in disjunctive plan representations by the use of constraint propagation techniques that enforce local consistency among the disjunctive plan constraints. This in turn reduces the number of component plans generated by later refinement strategies.

Thus, although we cannot expect a real performance improvement just by handling plansets without splitting (in that this merely exchanges complexity from search space size to solution extraction cost), disjunctive representations, constraint propagation, and SAT algorithms can in practice tilt the balance in its favor. In the next section, we elaborate the use of disjunctive representations and constraint propagation techniques.

5 Refining Disjunctive Plans and Constraint propagation

The general idea of disjunctive representations is to allow disjunctive step, ordering, and auxiliary constraints into the partial plan representation. Figure 4 shows two examples of

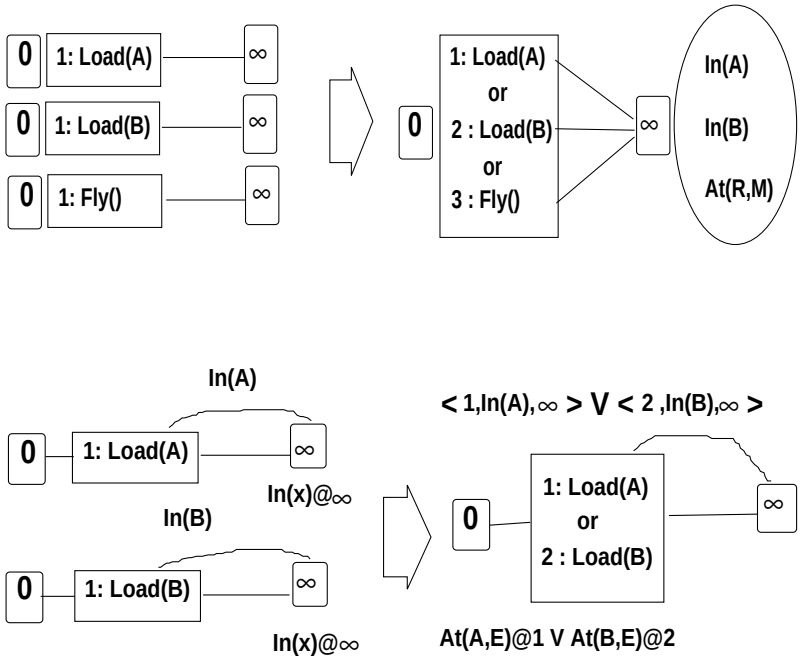

Figure 4: Converting plansets into disjunctive plans. On the top right, there are three plans that may be generated by a forward state space refinement, and on the top left is a single disjunctive plan that is equivalent to the three plans Similarly, on the bottom left we have two partial plans that may be generated by a plan-space refinement. On the top right is a single disjunctive plan that is equivalent to these two plans. .

converting sets of plans into single disjunctive plans. The three plans on the top left can be converted into a single plan on the right, with the disjunctive step and contiguity constraints. Similarly, the two plans on the bottom left can be converted into a single plan on the bottom right with disjunctive step, ordering and auxiliary constraints.

Candidate set semantics for disjunctive plans develop from the simple observation that the set of action sequences that satisfy the disjunctive constraint $c_1 \vee c_2$ is just the union of the set of action sequences that satisfy c_1 and c_2 respectively. In particular, the disjunctive plan on top left in Figure 4 admits into its candidate set any action sequence which starts with $Load(A)$, $Load(B)$ or $Fly()$ actions.

Disjunctive representations clearly lead to a significant increase in the cost of plan handling. For example, in the disjunctive plan on top right side, we do not know which of the steps will be coming next to step 0 and thus we do not quite know what the state of the world will be after the disjunctive step. So, how are we going to apply the FSS refinement? Similarly, in the disjunctive plan on the top right corner in Figure 4, we do not know whether steps 1 or 2 or both will be present in the eventual plan. Thus we do not know whether we should work on $At(A, E)$ precondition or the $AT(B, E)$ precondition.

At first glance, this might look hopeless as the only reasonable way of refining the disjunctive plans will be to split disjunction into the search space again, and refine the components separately. . However, it turns out that we are underestimating the power of what we do know, and how that knowledge can be used to constrain further refinements.

For example, in the plan on top right in Figure 4, knowing that only $Load(A)$, $Load(B)$ or $Fly()$ could have occurred in the first time step lets us realize that any eventual state of the world after the first step will contain only some subset of the conditions $In(A)$, $In(B)$ and $At(R, M)$, plus the conditions true in the initial state. This list of "feasible conditions" is best seen as the "union" of the states after the first time step. It is clear that any action whose preconditions are not a subset of this set cannot be executed in the second time step, *no matter which action we execute in the first time step*. This allows us to do a version of FSR that considers only those actions whose preconditions hold in the current list of feasible propositions at the current time step.

We can do even better in tightening the possible refinements. The knowledge that both $Load(A)$ and $Fly()$ actions cannot occur together in the first time step (since their preconditions and effects interfere), tells us that the second state may either have $In(A)$ or $At(R, M)$ but not both. So, any action whose preconditions include these conditions can also be ignored. This is an instance of propagation of mutual exclusion constraints and can be used to reduce the number of actions considered in the next step by the forward state space refinement. Specifically, the actions that require both $In(A)$ and $AT(R, M)$ can be ignored. This type of constraint prop-

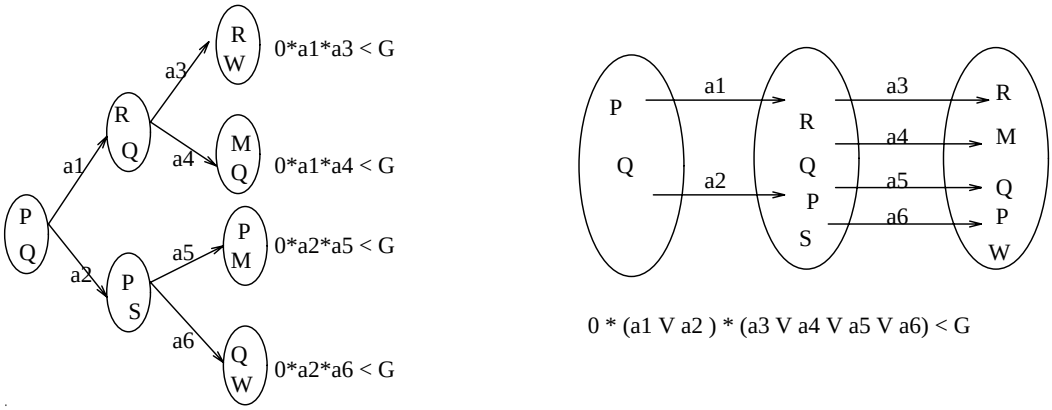

Figure 5: Interpreting GRAPHPLAN in terms of disjunctive representations. To the left is the search space generated by a forward state space refinement. To the right is the partial plan representation, called plan-graph, used in GRAPHPLAN Each candidate plan of the plan graph must have some subset of the actions in i^{th} level coming immediately before some subset of actions in the $i+1^{th}$ level (for all i). The minimal candidates corresponding to all plans generated by the forward state space planner are compactly represented by a single partial plan (plan graph) in GRAPHPLAN.

agation forms the backbone of the GRAPHPLAN algorithm [2] planning algorithm, that we shall discuss in more detail below. Of course, unless the propagation is very strong, we will not be able to weed out all infeasible actions from being considered. In summary, we can do refinements on the disjunctive representations directly, if we do not mind admitting some action sequences that would not have been the candidates of a planset produced by the same refinement operating on non-disjunctive plans. Soundness can still be maintained since the solution extraction process checks to make sure that a minimal candidate is a solution, before terminating.

Similar ideas apply to the disjunctive plan on the bottom right in Figure 4. For example, knowing that either 1 or 2 must precede the last step and give the condition tells us that if 1 does not, then 2 must. This is an instance of constraint propagation on orderings and reduces the number of establishment possibilities that plan-space refinement has to consider at the next iteration. We will see an example of this in UCPOP-D system described in Section 6. It is worth noting that the advantages of constraint propagation depend critically on the disjunctive representations. If we represented the plansets in terms of their components, the constraints on the different plan components would have been in terms of different step names and would thus not have interacted.

5.1 Case Study: GRAPHPLAN

GRAPHPLAN [2] is a recent planning system, that can be understood as using a partial plan representation that corresponds to the disjunction of the refinements produced by a forward state space planner (see Figure 5)[4]. Specifically, GRAPHPLAN's planning process involves two phases that are alternated until a plan is found. In the first phase, a compact structure called "plan-graph" is constructed. A plan-graph corresponds to the disjunction of all the refinements produced by a forward state space planner [16]. Thus the GRAPHPLAN refinement process does not introduce any branching into the search space. All the complexity is transferred to the solution extraction process which has to search the plan graph structure for minimal candidates that are solutions. A plan-graph specifies sets of operators at different time steps, such that each candidate solution must contain a subset of the actions at each time step contiguous to each other. As illustrated in Figure 5, the plan-graph can be seen as a disjunctive representation of the plansets generated by forward state space refinements. Empirical results show that GRAPHPLAN scales-up significantly better than non-disjunctive planners on a large number of benchmark domains.

GRAPHPLAN depends crucially on constraint propagation routines both in the plan-graph construction and solution extraction phases. Constraint propagation is done in terms of mutual exclusion relations between actions, stating that the presence of one action might necessitate the absence of another action, or vice versa. Two actions are mutually exclusive if their preconditions and effects are mutually conflicting. The mutual exclusion of actions at one time step can be propagated to make otherwise independent actions at a latter time step mutually exclusive. Extension of k-level plan-graph to a $(k+1)$-level plan-graph roughly corresponds to doing forward state-space refinement on all the components of the k^{th} level planset.[5] Empirical results demon-

[4]There are some minor further differences between the search space of normal forward state space refinements and the plan-graph representation of GRAPHPLAN. Specifically, plan-graph construction is better understood in terms of a forward state space planner which allows "noop" actions, and projects sets of independent operators simultaneously from the current state See [16] for a full reconstruction of GRAPHPLAN from forward projection.

[5]Strictly speaking, the candidate set of the k-level plangraph is a *superset* of the candidate set of the corresponding k^{th} level planset [16]. This is because GRAPHPLAN may introduce actions into level $k+1$ which may not be applicable in any physical state af-

strate that shifting the complexity entirely from search space size to solution extraction this way does lead to significant improvements in performance. Kautz and Selman [9] show that GRAPHPLAN's performance can be further improved by casting its solution extraction process as a SAT problem and solving it using local search methods.

6 UCPOP-D: Disjunction over Plan-space refinements

In the foregoing, we have argued in favor of disjunctive refinements and constraint propagation, and suggested that GRAPHPLAN algorithm can be understood in these terms. The success of GRAPHPLAN shows that there is a lot to be gained by considering other disjunctive partial plan representations. Since GRAPHPLAN concentrates on disjunctive representations of a forward state space planner, one interesting direction is to explore disjunctive representations over other refinements. In this section, we will describe our experience with disjunctive representations over plan-space refinements in the context of UCPOP [19], a popular partial-order planner.

This section has multiple aims. We want to demonstrate that the ideas of disjunctive representations can be folded naturally into existing refinement planners. We also want to explore the middle-ground in terms of splitting plansets. UCPOP splits every component of the planset resulting from a refinement into the search space. GRAPHPLAN keeps all components together with disjunctive representations. The UCPOP-D algorithm we discuss here disjoins some of the components of the planset produced by a refinement into a single plan, while keeping the other components separately.

Since plan-space refinements concern themselves with different ways of establishing a specific condition in the plan, disjunction here will deal with multiple establishment possibilities together. Let us motivate the utility of such a disjunction with an example. Consider the plan shown to the left in Figure 6, which contains a step s that requires a precondition p. There are two steps s_1 and s_2 in the plan such that both of them are capable of providing the condition p to s. When planners using plan-space refinement consider the precondition $p@s$ for establishment, they typically make several refinements, two of them corresponding to simple establishment with s_1 and s_2 respectively. The resulting refinements will contain IPCs $(s_1 \stackrel{p}{-} s)$ and $(s_2 \stackrel{p}{-} s)$ respectively. One way of cutting down the branching in this process is to combine these two plans into a single refinement, and ensure that either s_1 or s_2 will give p to s.[6] The search space schematic on the right of Figure 6 illustrates how the branching factor is reduced by such disjunctive causal commitment constraints. Specifically, the simple establishment options are all bundled into a single plan.

To support such disjunctive causal commitments, we have to ensure that (a) either s_1 or s_2 will precede s and (b) for every step s_t that deletes p, either either s_t comes after s or it comes before either s_1 or s_2, with s_1 or s_2 preceding s at the same time. More generally, if we want to use any of n steps $s_1, s_2 \cdots s_n$ to support some condition at step s, we need to impose the following disjunctive ordering constraints:

$$(s_1 \prec s) \vee \ldots \vee (s_n \prec s).$$

For every step s_t that can threaten the establishment, we need:

$$(s \prec s_t) \vee [(s_t \prec s_1) \wedge (s_1 \prec s)] \vee \ldots \vee [(s_t \prec s_n) \wedge (s_n \prec s)].$$

6.1 Handling Disjunctive Orderings via constraint propagation

In the above, we noticed that maintaining disjunctive causal structures ultimately boils down to handling disjunctive orderings (in the case of propositional planning). This can be done efficiently with the help of constraint propagation techniques [20]. The basic idea is that whenever an atomic ordering constraint is added to the plan, it can be propagated through the disjunctive constraints, simplifying them. The simplification may give rise to more atomic orderings, which in turn cause further simplifications. In our implementation, we do two types of direct simplifications or local consistency enforcement: (1) A disjunctive ordering constraint $O_1 \vee O_2 \ldots \vee O_n$ (where O_i are atomic orderin constraints) simplifies to $O_1 \vee O_2 \vee \ldots O_{i-1} \vee O_{i+1} \ldots \vee O_n$ if an ordering O_i^{-1} (where the inverse of an ordering relation $s_1 \prec s_2$ is $s_2 \prec s_1$) is propagated through it. It also simplifies to $True$ when an ordering O_i is propagated through it.[7] This type of propagation often simplifies the disjunctions completely. If disjunctive orderings remain in the plan by the time all other open conditions are established, we can solve the CSP corresponding to the ordering constraints to check if there exists a ground linearization of the plan that is consistent. Rather than use a separate CSP, in our current implementation, we simply split the disjunction into the UCPOP search space. Since we only do this splitting for those disjunctive orderings that remain unsimplified, the search space size is likely to be much smaller than that for normal UCPOP.

ter the components of the k^{th} level planset. Some, but not all, of these inapplicable actions are weeded out by the fact that GRAPHPLAN propagates mutual exclusion relations among state literals and avoids introducing actions whose preconditions are mutually exclusive. This is the price we pay for the simplicity of disjunctive representation.

[6] The idea of maintaining multiple causal contributors has been around, and our previous work provides a formalization of them [11], and uses them to revoke prior causal commitments. However, this is the first time that disjunctive causal commitments are used in their full generality, involving disjunctive ordering constraints and constraint propagation, to control search space explosion

[7] It is of course possible to enforce stronger constraint propagation – for example, resolving two disjunctive ordering constraints $O_1 \vee O_2$ and $O_1' \vee O_2'$ to $O_1 \vee O_2'$ when $O_2 = \neg O_1'$. But, our current experience is that such stronger propagation strategies do not improve performance [20].

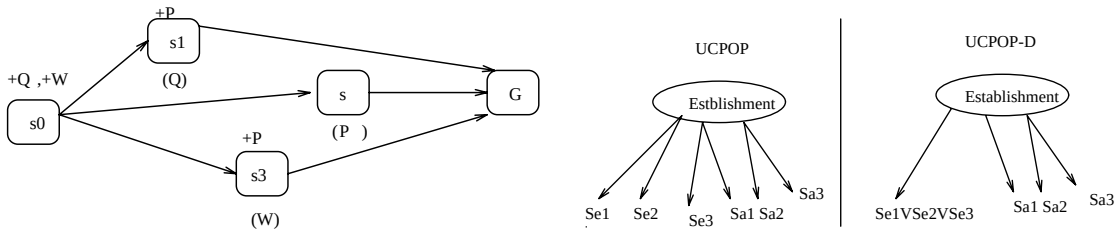

Figure 6: Combining establishment branches in plan-space refinement using disjunction

6.2 Implementation and Empirical Evaluation

We implemented these ideas about disjunctive causal constraints on top of the standard UCPOP system from University of Washington [19]. Our initial implementation made minimal changes to UCPOP – for example, we depend on the standard termination criterion of UCPOP, rather than the minimal candidate based termination. For convenience, we will call this variant of UCPOP, UCPOP-D. Since UCPOP is already optimized to handle consistency with atomic ordering constraints, the easiest way to make it handle disjunctive orderings was to keep a separate field in the plan structure for disjunctive orderings. Whenever propagation techniques derive new atomic orderings from the existing disjunctive orderings, they are added to the normal orderings list of UCPOP. As described above, the changes to the algorithm come in two places – first is in establishing new conditions through existing steps, and the second is in handling of conflicts to causal commitments. In both cases, additional disjunctive orderings are added to the plan representation.

The disjunctive orderings are simplified whenever the planner adds non-disjunctive orderings to the plan. The latter are added by the planner either as part of handling unsafe links when only one contributor or one type of conflict resolution is possible. In either case, the ordering is *propagated* through the disjunctive orderings of the plan.

In most cases, propagation simplifies disjunctive orderings of the plan so that by the time all open conditions and unsafe links are handled, the plan will not have any disjunctive orderings left. If disjunctive orderings are left unsimplified however, the normal termination criteria of UCPOP will not apply and unsound plans can be returned. In the current implementation, we avoid this by explicitly splitting the disjunction into the search tree in such cases.

6.3 Results

To see whether or not the use of disjunctive constraints and propagation helps improve the planner efficiency, we conducted empirical studies in a variety of domains. To provide a baseline for comparison, we used three different versions of UCPOP - one with simple causal constraints, but with constraint propagation used to handle conflict resolution (CP+NMCL), the second with disjunctive causal commitments and constraint propagation over disjunctive orderings (CP+MCL) and finally the standard UCPOP implementation, which uses single contributor causal links and non-disjunctive representations. The plots in Figure 7 show the results of these experiments. The first set of plots show the comparative performance in ART-MD-RD domain, which was used in [11] to illustrate the utility of multi-contributor causal links. The second set of plots show the results in Link-Chain domain, which was used by Veloso et. al [21] to highlight the inadequacies of the single-contributor causal structures. Finally, the third set of plots show the results in the prodigy blocks world domain (with $pickup, putdown, stack$ and $unstack$ actions). The plots show that in all cases, disjunctive representations and constraint propagation outperform standard UCPOP. In particular, the disjunctive causal commitments improve performance significantly.

6.4 Extending UCPOP-D

UCPOP-D can be extended in several ways. The first obvious step would be to extend it to non-propositional cases, and this can be done by applying constraint propagation to variable codesignation and non-codesignation constraints. Allowing disjunction over step-addition establishments, and/or handling actions with conditional effects will lead us to disjunctive open conditions and will necessitate more extensive changes to partial-order planning algorithms. The main changes come in the form of generalizing plan-space refinement to handle fully disjunctive partial plans. For example, suppose we are considering two contributors s_1 and s_2 as possible contributors of the condition p to the step s. Suppose s_1 has a conditional effect $r \Rightarrow p$ and s_2 has the conditional effect $q \Rightarrow p$. Since $r@s_1$ will be a secondary precondition if s_1 gives p and $q@s_2$ will be a precondition if s_2 gives p, the disjunctive causal commitment thus leads to disjunctive open condition $r@s_1 \vee q@s_2$, only one of which need to be achieved. Even here, it is possible to handle them in terms of steps currently existing in the plan. Suppose the steps s_3 and s_4 are currently in the plan and they give conditions r and q respectively, we can take care of the disjunctive open condition $r@s_1 \vee q@s_2$, with the help of the disjunctive causal constraint $(s_3 \overset{r}{-} s_2) \vee (s_4 \overset{q}{-} s_3)$, and handling it in the usual way. The problem comes when we are trying to make the disjunctive open condition true by adding new steps. From least commitment point of view, it is best to introduce all the new steps capable of establishing any of the

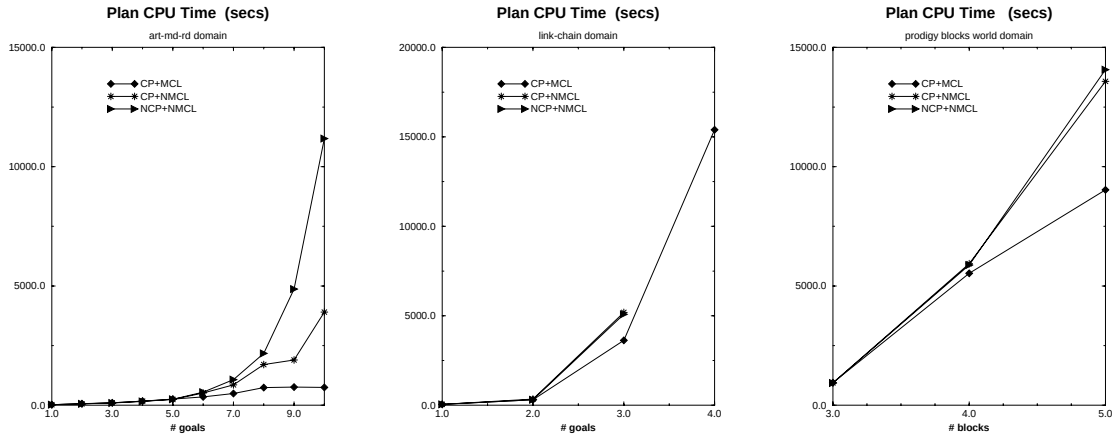

Figure 7: Plots showing the utility of disjunctive causal link representations in multiple domains

disjuncts of the disjunctive open condition simultaneously, and consider the steps to be "tentative" in that only one of those steps need be present in the final plan. The tentative steps will then give rise to tentative open conditions which need to be handled. While applying establishment refinement to all tentative conditions is one possibility – and this is essentially what is done in the causal "SNLP" encodings used in [10] – we are hopeful that there may be ways of more tightly constraining the refinements (as is done in GRAPHPLAN using mutual exclusion constraints, see Section 5).

7 Discussion

In this section, we shall discuss several broad issues raised by planning with disjunctive representations:

Amount of splitting: Research in the constraint satisfaction literature shows that propagation and splitting can have synergistic interactions in improving planner performance. Specifically, splitting leads to commitment, which in turns supports local consistency enforcement. As an example, in the 8-queens problem, committing the position of one of the queens leads to improved local consistency enforcement through arc-consistency algorithms. This leads to the possibility that the best way to do planning may involve being deliberate about splitting the plansets into the search space. Complete splitting leads to search space explosions as typified in UCPOP type planners, while avoiding splitting all together may inhibit constraint propagation.

Table 1 describes a variety of planners and the amount of splitting/search they employ. The question of how much splitting is to be done is a hard one. We believe that to a large extent this depends on common substructure between the components of the planset. In general, it may be best to combine components with shared structure into a single disjunctive plan so that propagation is facilitated among its constraints. UCPOP-D provides a preliminary example of this strategy.

Relative support provided by different refinements: Another important open issue is which refinements should we be using to get the maximum mileage out of disjunctive representations. Clearly, all refinements can support planning with disjunctive representations. We have already seen that GRAPHPLAN can be understood in terms of disjunction over state space refinements, while UCPOP-D and Descartes [8] can be understood in terms of disjunction over plan-space refinements. An important issue is the relative tradeoffs offered by disjunction over different types of refinements. The previous analyses of relative tradeoffs concentrated on the level of commitment enforced by a refinement strategy in its planset components and on the subgoal interactions that arise due to this. This analyses is of little use in planners using disjunctive representations since they do not uniformly split planset components into the search space. Instead, we need to concentrate on issues such as: (a) the efficiency of extraction of solutions from the plansets and (b) the support for constraint propagation provided by the plansets. As an example of such a tradeoff, recent work by Kautz et. al. [10] indicates that SAT "encodings" based on disjunctive state-space refinements are larger than the encodings based on plan-space refinements, but that the latter do not seem to be as easy to handle during solution extraction process.

8 Related Work

As we mentioned, our description of refinement planning is general enough to include both traditional refinement planners and some new planning algorithms. Table 1 characterizes several of these planners in terms of our framework. We believe that the understanding of the role of disjunctive refinements and constraint propagation that we developed in this paper facilitate a clearer appreciation of the connections between the many CSP-based planning algorithms and traditional refinement planning. We have already shown how GRAPHPLAN can be understood in terms of re-

Planner	Level of splitting	Type of refinement
UCPOP [19], SNLP [17]	Full	PSR
TOPI [1]	Full	BSR
GRAPHPLAN [2]	No	FSR
SATPLAN [9]	No	FSR/PSR
Descartes [8]	Some	PSR
UCPOP-D	Some	PSR

Table 1: Different planners as instantiations of **Refine** algorithm template

finement planning with disjunctive representations and constraint propagation. Another successful strand of work in plan generation is related to SATPLAN, which considers planning as a satisfiability problem [9, 10]. SATPLAN starts with a SAT encoding such that all k-step solutions to the planning problem correspond to the satisfying assignments of the encoding. The idea is to produce SAT encodings corresponding to different solution lengths, and solve them until a satisfying assignment is found to one of the instances. Much of the work here is aimed at generating "compact" encodings that are easy to solve using existing systematic or local satisfiability checking algorithms.

Described this way, there seems to be little connection between this strand of work and the traditional refinement planning algorithms. However, general refinement planning framework presented in this paper does clarify some of the tradeoffs in posing planning as satisfiability. For example, there is a straightforward connection between the SAT encodings for k-step solutions, and the plansets produced by refinement planners. In particular, consider a planset $\widehat{\mathcal{P}}$ produced by any refinement planner using a series of complete and progressive refinements, such that all components of $\widehat{\mathcal{P}}$ have exactly k steps. It can be shown that each k-step solution for the planning problem must correspond to one of the minimal candidates of $\widehat{\mathcal{P}}$.[8] The problem of finding a minimal candidate of $\widehat{\mathcal{P}}$ that corresponds to a solution can be posed as a SAT problem.

This relationship brings up several points to the fore: The exact representation of the planset $\widehat{\mathcal{P}}$, and consequently the size of its SAT encoding, depend on the types of refinements used (recall that we can use any sequence of state space or plan space refinements since they are all progressive and complete). This relates the syntactic notion of the SAT encoding size to the well-established idea of refinement strategies. Different sequences of refinements will correspond to different encodings. There may thus be a strong connection between the theories of refinement selection and the theories of encoding generation.

The current implementation of SATPLAN does not explicitly consider refinements, but rather attempts to come up with a necessary and sufficient set of propositional constraints that must be satisfied by all k-step solutions. It would be interesting to consider generating the encodings using refinements on disjunctive plans. Kautz and Selman [9] do this implicitly when they convert GRAPHPLAN's plan-graph into a SAT instance. This raises the possibility that the research on refinement selection, such as "goal order heuristics" in partial order planning [7], or planning by interleaving multiple refinements [13] may have an impact on generating efficient SAT encodings.

It is also worth understanding the relation between least commitment, task reduction ideas in traditional refinement planning, and the idea of planning with disjunctive representations. Most existing refinement planners use plan representations that comprise of conjunctions of atomic constraints: steps, orderings between steps etc. The usual way of increasing the candidate set size of partial plans (and thus reduced commitment), is to use weaker atomic constraints (e.g., partial ordering instead of contiguity ordering relations between steps). Disjunctive representations provide an alternate way of achieving least commitment and search space reduction. HTN planning [14] can be seen as an idea closely related to disjunctive refinements. In particular, HTN planners introduce non-primitive actions into the partial plans, and gradually reduce them with the help of user-supplied task reduction schemas into primitive actions. Thus, the presence of non-primitive action in a partial plan can be interpreted as the presence of disjunction of all the plan fragments that the action can be eventually reduced to. One important difference is that HTN planners never explicitly deal with this disjunction, but push it gradually into the search space (by considering each reduction of the action in a different search branch). Thus, they miss out on the advantages of handling plansets together, such as constraint propagation. In a way, both traditional least commitment and task reduction ideas can be seen as instances of the general heuristic of converting the search space to have lower branching at the top levels. Disjunctive representations give this ability, but they also support handling plansets together without ever pushing them back into the search space.

9 Conclusion

By teasing apart the hitherto closely intertwined notions of "refinement" and "search", we were able present a model of refinement planning that not only includes the traditional planners, but also supports a large variety of planners that transfer search to solution extraction phase by handling sets

[8] Not all minimal candidates may be k-step solutions, however.

of partial plans together. We have argued that efficient handling of large plansets requires the use of disjunctive plan representations and constraint propagation techniques.

We have used our model of refinement planning to explain the operation of several newer planning approaches such as GRAPHPLAN and SATPLAN. We have also shown how the ideas of disjunctive representations and constraint propagation can be incorporated into traditional planners by presenting the UCPOP-D algorithm that supports disjunctive handling of plans differing only in their causal structures, and demonstrating that UCPOP-D outperforms UCPOP in several domains. Finally, we discussed several broad issues raised by this "inclusive" picture of refinement planning.

Acknowledgements:

We thank Biplav Srivastava for his many helpful critiques of the ideas in this paper, and his help in preparing some figures. This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory planning initiative grants F30602-93-C-0039 and F30602-95-C-0247, and the ARPA ASERT award DAAH-04-96-1-0247.

References

[1] A. Barrett and D. Weld. Partial Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence*, Vol. 67, No. 1, 1994.

[2] A. Blum and M. Furst. Fast planning throug planning graph analysis. In *Proc. IJCAI-95*, 1995.

[3] J.D. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. AAAI-93*, 1993.

[4] E. Fink and M. Veloso. Formalizing the Prodigy Planning Algorithm. CMU CS Tech. Report, Fall 1994.

[5] M. Ginsberg. A new algorithm for generative planning. In *Proc. KRR-96*, 1996

[6] L. Ihrig and S. Kambhampati. Design and Implementation of a Replay Framework based on a Partial order Planner. In *Proc. AAAI-96*, 1996.

[7] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial order planning. *Proceedings of AAAI-94*, 1994.

[8] D. Joslin and M. Pollack. Passive and active decision postponement in plan generation. In *Proc. 3rd European Workshop on Planning*, 1995.

[9] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search In *Proc. AAAI-96*, 1996.

[10] H. Kautz, D. McAllester and B. Selman. Encoding plans in propositional logic In *Proc. KRR-96*, 1996.

[11] S. Kambhampati. Multi-Contributor Causal Structures for Planning: A Formalization and Evaluation. *Artificial Intelligence*, Vol. 69, 1994. pp. 235-278.

[12] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A Unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence* special issue on Planning and Scheduling. Vol. 76. 1995.

[13] S. Kambhampati and B. Srivastava. Universal Classical Planner: An algorithm for unifying state space and plan space approaches. In *Proc European Planning Workshop*, 1995.

[14] S. Kambhampati and B. Srivastava. Unifying classical planning approaches. ASU-CSE-TR 96-006. Submitted for journal publication.

[15] S. Kambhampati, S. Katukam, Y. Qu. Failure Driven Dynamic Search Control for Partial Order Planners: An explanation-based approach *Artificial Intelligence* (To appear in Fall 1996)

[16] S. Kambhampati. Reconstructing GRAPHPLAN Algorithm from Forward Projection. In Planning Methods in AI (Notes from ASU Planning Seminar). ASU CSE TR 96-004. *http://rakaposhi.eas.asu.edu:8001/yochan.html*

[17] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proc. 9th AAAI*, 1991.

[18] B. Selman, H.J. Levesque and D.G. Mitchell. GSAT: A new method for solving hard satisfiability problems. In *Proc. AAAI-92*, 1992.

[19] J.S. Penberthy and D. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proc. KR-92*, 1992.

[20] E. Tsang. *Foundations of Constraint Satisfaction*. (Academic Press, San Diego, California, 1993).

[21] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. *Proceedings of AIPS-94*, 1994.

[22] Q. Yang. A theory of conflict resolution in planning. *Artificial Intelligence*, 58:361-392, 1992.