Adaptation Versus Retrieval Trade-Off Revisited: an Analysis of Boundary Conditions

Stephen Lee-Urban, Héctor Muñoz-Avila

Department of Computer Science and Engineering, 19 Memorial Drive West,
Lehigh University, Bethlehem, PA 18015
{sml3,hem4}@lehigh.edu

Abstract. In this paper we revisit the trade-off between adaptation and retrieval effort traditionally held as a principle in case-based reasoning. This principle states that the time needed for adaptation reduces with the time spent searching for an adequate case to be retrieved. In particular, if very little time is spent in retrieval, the adaptation effort will be high. Correspondingly, if the retrieval effort is high, the adaption effort is low. We analyzed this principle in two boundary conditions: (1) when very bad and (2) when highly capable adaptation procedures are used. We conclude that in the first boundary condition the adaptation-retrieval trade-off does not necessarily exist. We also claim that the second does not hold for a class of planning domains frequently used in the literature. To validate this claim, we performed experiments on two domains of this type.

Keywords: cased-based reasoning, planning, retrieval, adaptation, trade-offs.

1 Introduction

One of the crucial principles of case-based reasoning is the trade-off between the retrieval time, the time it takes to find a relevant case for a given problem from the case base, and the adaptation time, the time it takes to adapt the retrieved case. The trade-off has been summarized in Fig. 1, taken from [1]. There are three tenets of this principle:

1. If little time is spent on retrieval, then, on average, the adaptation effort involved in using the retrieved cases to solve the given problem will be high. This is basically the result of stopping the retrieval process too early, which results in the retrieval of cases that are not easy to adapt.
2. If too much time is spent on retrieval, then the adaptation effort to solve the given problem will be small. This is basically the result of spending enough time to ensure that a case is retrieved that is easier to adapt. However, any reduction in the adaptation time is counterbalanced by the time spent in retrieval, which may result in a high overall problem-solving time (adding up retrieval and adaptation times).

3. There is an optimal intermediate point at which a case that is good enough to adapt is retrieved and adapting it produces the smallest overall time.

The tenants of this principle have been recurrently discussed over the years. It was in part the motivation for retrieval strategies in case-based planning systems such as CAPLAN/CbC [2], derSNLP+EBL [3], and MRL [4]. It was also observed by studies about the occurrence of the utility problem in case-based reasoning [5]. It continues to be encountered in a number of domains including software design [6], travel domain [7], and process planning [8].
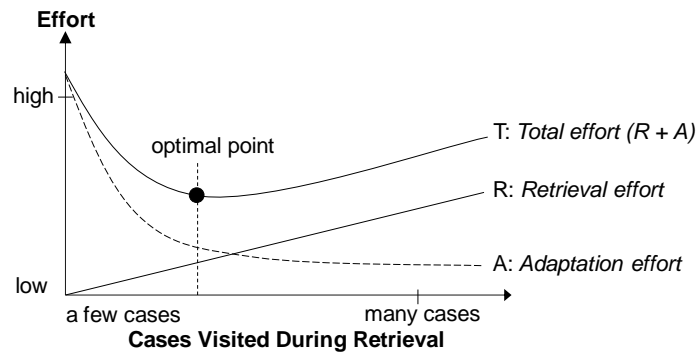

Fig. 1. Adaptation and retrieval trade-off (Veloso, 1994)

Improvements in retrieval procedures (e.g., [9], [10]) can be seen as an effort to reduce the time to find cases "good enough" for fruitful adaptation. Analogously, improvements in adaptation (e.g., [11], [12], [13], [14], [15]) can be seen as indirectly reducing the time needed for retrieval because as adaptation is improved, less time need be spent on finding "good enough" cases to be adapted. Hence both kinds of improvements, in adaptation and retrieval, can be seen as moving the optimal point depicted in Fig. 1, and described in the third tenet, downward, making the overall problem-solving effort less costly.

The rest of the paper proceeds as follows: In Section 2 presents our analysis of the two boundary conditions. Section 3 provides an illustrative plan adaptation example. In Section 4, we describe domain-configurable plan adaptation by first reviewing partial-order planning (4.1), then by explaining the domain-configurable plan adaptation knowledge (4.2) used by our adaptation procedure (4.3). Next, Section 5 gives an example of domain-configurable plan adaptation. The details of our empirical evaluation are presented in Section 6, followed by concluding remarks.

2   Analyses of Boundary Conditions

An implicit assumption in the adaptation-retrieval trade-off principle is that the adaptation algorithm is only capable of solving problems quickly enough when the retrieved case is reasonably similar to the input problem. If the retrieved case is not sufficiently similar to the new problem then the adaptation effort will take a

significant amount of time. It is worth noting that despite advances in adaptation algorithms, there is still a search process needed in a potentially exponential search space. The relationship between this search space and techniques for its exploration can be loosely summarized as follows: A case sufficiently similar to the input problem is retrieved and adjusted in a sensible way. The output of this process is an "adjusted plan" where the retrieved plan is partially adapted but is an incomplete solution to the input problem. Hence, further search is required to reach a solution node of the new problem starting from the adjusted plan. This search can be performed either by first-principles search as in [11] or by retrieving and adapting another case as in [12] or by composing multiple planning cases as in [16] or by combining further retrieval/adaptation of cases and first-principles search as in [1] and [17].

To make this assumption explicit, we will study two boundary cases: when the adaptation algorithm is capable of solving only those problems for which it has already stored solutions in the case base (naïve) and when a highly capable adaptation procedure (omniscient) is used.

2.1   Analysis of a Naive Adaptation Algorithm.

Fig. 2 (left) illustrates the situation of a naïve adaptation procedure, capable of solving only those problems for which it already has a solution. The dotted segments represent discontinuities in problem-solving time, reflecting those problems for which a solution does not exist in the case base (CB) and therefore no data point can be drawn. The adaptation time for those problems for which a solution is already stored is zero because the solution is taken as-is. Consequently, the linear search of the retrieval procedure (as in Fig. 1, assume a constant time to compute similarity between case and problem, and a sequential search through the cases), yields an overall linear time for problem solving. Basically, the problem-solving time is the time taken for the retrieval procedure to find the solution in the case base, if one is already stored. This is an analogy of the CB working as a sequential database. In this situation there is no trade-off between adaptation and retrieval. The retrieval mechanism must continue looking for a solution until it finds an exact match or it has exhausted the whole case base.
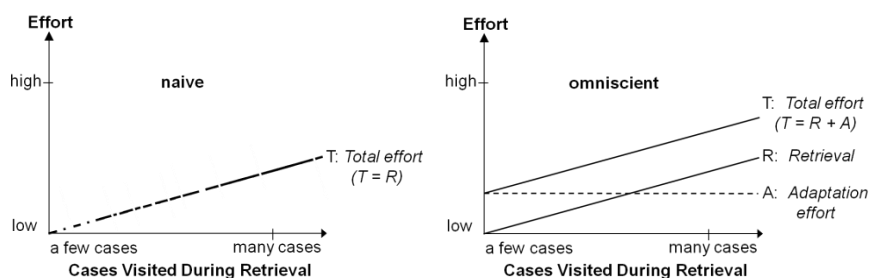

Fig. 2. Naïve (left) and omniscient (right) adaptation algorithms without a trade-off

2.2. Analysis of an Omniscient Adaptation Algorithm.

More interesting and difficult is to analyze the situation where an omniscient adaptation algorithm is given. First, we would like to characterize such an algorithm. An omniscient adaptation procedure is one where it would not need to backtrack to reach the adjusted plan nor backtrack during search to further refine the adjusted plan into a complete solution of the new problem. That is, somehow the algorithm finds the path in the search space to first reach the adjusted plan and then reach the solution plan without needing to explore alternative branches in the search space.

Our hypothesis is that there exists a class of planning domains such that the time it takes an omniscient adaptation algorithm to adapt any two cases to solve a new problem is roughly the same, regardless of the similarity of the individual cases to the new problem, and given that the problems solved by the cases and the new problem are of the same size (i.e., they have the same number of objects in their initial states). Once again, combined with the linear search of the retrieval procedure, this hypothesis gives an overall linear time for problem solving. Basically, the problem-solving time is proportional to the time it takes for the retrieval procedure to retrieve a case. We claim that in this situation there is no trade-off between adaptation and retrieval as illustrated in Fig. 2 (right).

This class of planning domains is one where the graph consisting of all world states is a directed, strongly connected graph. In this graph, the vertexes are world states and the directed edges are actions (i.e., an edge from vertex x to vertex y represents the action transforming the state x into state y). We called these **connected domains**. In a connected domain, there always exists a directed path (sequence of actions) between any two vertexes (states) in the graph. The logistics transportation domain [1] is an example of a connected domain, provided that there is at least one truck and one airport in each city, as well as at least one airplane. With these provisions, a package in any location can always be relocated into any other location. Similarly, the blocks world [18] also meets this property: any configuration of blocks can always be reconfigured into any other configuration of these same blocks. An example of a domain that does not meet this property is a logistics domain variant where there are one-way routes between locations [3]. As a result, a package in a certain location may not always be reachable by a truck.

An important question is whether it is possible to construct an omniscient algorithm for connected domains. This is particularly compelling, considering that many experiments designed to demonstrate efficiency gains of new case-based planning techniques use connected domains. Additionally, these domains have an exponential search space and, hence, the question of whether adaptation procedures could be built that somehow adapt the retrieved plans without exploring unnecessary branches in the search space is a good one.

In this paper we will report on an omniscient adaptation algorithm, DCPOP-A (for: domain-configurable partial-order plan adapter). In Section 6, we report the results of experiments that demonstrate, for the connected domains described above, the non-existence of a trade-off between adaptation and retrieval in our omniscient adaptation algorithm as depicted in Fig. 2 (right).

3 A Plan Adaptation Example

Fig. 3 illustrates a snippet of the search space for state-space planning in blocks world. It shows 15 states (five of them labeled *p1*, *state 1*, *state 2*, *state 3* and *final*) out of the 73 possible states with 4 blocks. Lines connecting states represent transitions by the only action in the domain: *Move*(*?x ?y ?z*). This action puts a clear block *?x* (i.e., a block with no other block on top of it) currently on top of a block *?y* (or on the table; *?y* = *table*) on top of another clear block *?z* (or on the table).

Transitions do not have a direction because with a change of parameters and/or actions they can go in either direction. Problems can be simply defined as pairs of states (s, s') where s is the initial state and s' is the goal state. Suppose that two cases are stored in the case base, Case 1 and Case 2. Case 1 solves the problem (*state 1*, *final*) and Case 2 solves the problem (*state 2*, *final*). Their solutions follow the path between these two states denoted by the bold line-connectors (the double-line parts represent common portion of the solutions). Suppose that a new problem (*state 3*, *final*) is given. Of these two cases, Case 2 is a much better choice. In fact if starting from *state 3*, there are only 3 possible transitions, two of which lead directly to the solution plan of Case 2 and the third one (the 4 block pile) which is a dead-end. Case 1 might not be a good choice. For example, from *state 3* a path of length 4 leading to the dead-end representing the 4-block pile [B, A, D, C] can be explored. There is also a length 2 path to p1 which would allow it to use the solution path of Case 1 but a first-principles search might take a significant amount of time before this path is found because it may first explore dead-end paths.
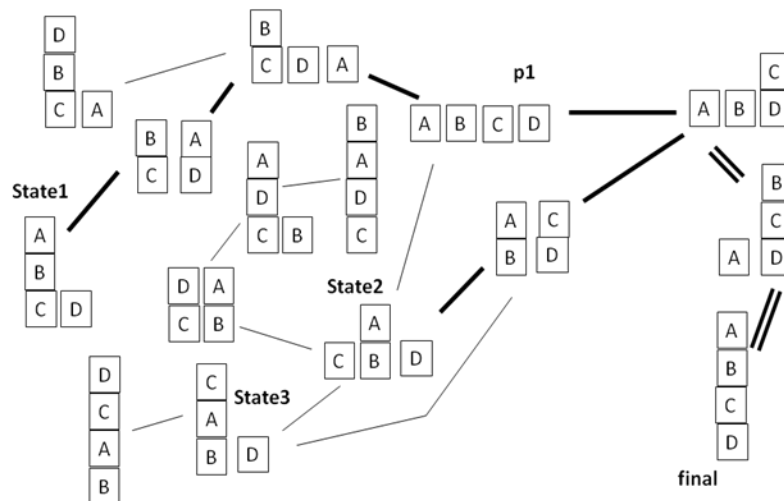

Fig. 3. Snippet of the search space with 4 blocks

4   Domain-Configurable Plan Adaptation

Our approach for plan adaptation is motivated by existing research in domain-configurable planning. In this form of planning, domain-specific knowledge enhancing the action schemas is given. This knowledge is used to guide the planning process, which like first-principles planning generates a plan from scratch. Domain-configurable planners have been shown to solve problems more quickly and to scale much better with problem size than first-principles planners. Because of their scalability, their increasing number of applications, and their ability to drop classical planning assumptions, domain-configurable approaches are believed to be closing the gap between academic research in AI planning and real-world applications [19].

We developed DCPOP-A, a domain-configurable plan adaptation algorithm, in order to investigate the adaptation-retrieval trade-off in a system capable of performing "omniscient" search with ideal inputs. This new problem-solving paradigm for plan adaptation uses domain-specific knowledge to guide a domain-independent plan adaptation process. The domain-independent property allows the semantics of the resulting planning algorithms to be clear. Domain-specific knowledge allows problem-solving to scale well with problem size. This, in addition to previous analyses of the search space, illustrates the potential for substantial speed-up gains in the plan adaptation process, thus providing a suitable framework in which to re-evaluate the adaptation-retrieval trade-off.

We used partial-order planning (POP) as the underlying planning formalism to conduct our research. POP was the dominant planning paradigm some 15 years ago because of its ability to flexibly interleave actions, rather than totally order them, while solving problems. POP drops the classical requirement for actions to be totally ordered, which is particularly useful for plan adaptation (e.g., [11], [12]). However, interest in POP waned when other paradigms such as analysis of planning graphs and more recently planning with heuristics, demonstrated significant gains in planning speed and solvable problem size. More recently, there has been a revival of POP as heuristic methods have been developed that perform comparably to other state-of-the-art first-principles planners. Researchers have pointed out the importance of POP planning for real-world domains because in many real world situations actions can be performed in parallel and the planner should not commit to step orderings unless necessary (e.g., [20], [21], [22], [23]).

4.1   Partial-Order Planning

Partial-order planning begins with an input action schema and a symbolic initial and goal state specification of the problem. ***Actions*** have a name, zero or more parameters, preconditions, and effects. Next an ***initial plan*** is created, consisting of two special steps. The first of these steps has as effects those atoms appearing in the problem's initial state; the second has as preconditions those atoms appearing in the goal state. Partial-order planning refines this initial plan by adding constraints and plan steps, ordered between the two initial steps, until a complete partial-order plan is obtained (complete plans are defined below). A ***partial-order plan*** is defined as a 4-tuple $(S, \rightarrow, \rightarrow_{CL}, B)$ of sets of ***POP plan elements.*** S is the set of plan ***steps***, which

represent the application of actions in the plan. The set $\rightarrow$ contains the ***ordering constraints*** between plan steps, which take the form $s \rightarrow s'$, indicating that step s must be executed before step s'. The set $\rightarrow_{CL}$ contains ***causal links***, $s \rightarrow_p s'$, indicating that the precondition p needed by the action in step s' is produced as an effect of the action in step s. Step s might be an existing step in the plan or a new one added to satisfy p. The set B indicates variable ***binding constrains***, $?x \neq ?y$ or $?x = ?y$, indicating that whenever variable $?x$ occurs in the plan it must take a different (respectively the same) value as the variable $?y$ ($?x$ represents that x is a variable symbol). Set B is empty when planning without variables (i.e. "grounded"). A partial-order plan is ***complete*** if it has no flaws. There are two kinds of ***flaws*** in POP: open preconditions and causal threats. An ***open precondition*** occurs when a step s' in the plan has a precondition p, written $p@\, s'$, for which no causal link $s \rightarrow_p s'$ exists. A ***threat*** occurs when a causal link $s \rightarrow_p s'$ and a step s'' exist such that s'' has as effect the negation of p (i.e., $\neg p$), written $s'' \rightarrow_{\neg p}$, and s'' can occur between s and s', written $s'' \,||\, (s \rightarrow_p s')$, in a linearization of the plan. A ***linearization*** of a plan is a sequencing of all steps in a manner consistent with the ordering constraints such that, for every two steps s and s', s will always be listed before s' if either $s \rightarrow_p s'$ or $s \rightarrow s'$ hold. The objective of the POP planning process is to refine the initial plan into a complete partially-ordered plan. Any linearization of this complete partially-ordered plan is a solution to the planning problem. There are four possible ***POP plan refinements***: adding ordering constraints, adding steps, adding causal links, and adding binding constraints. Ordering constraints and binding constraints are added to solve causal threats. Steps and causal links are added to satisfy open preconditions.

Fig. 4 shows an example of a partial-order plan in the Blocks World domain. The arrows represent causal links and ordering constraints. The meaning of the "x" beside some of the plan steps will be explained in a later discussion and may be ignored for now. If nothing is underneath a block, this means that the block is on the table (not shown), For example, in the initial state block C is on the table and in the goals block B is on the table. This plan unstacks all blocks on the table and then stacks them to form the configuration indicated by the goals.
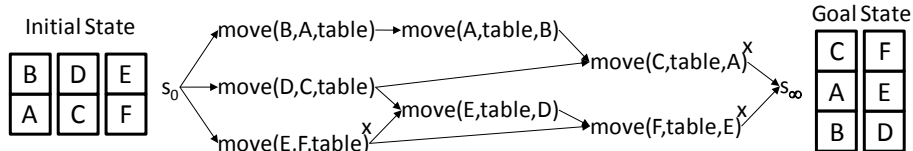

Fig. 4. Example of a partial-order plan

4.2 Domain-Configurable Partial-Order Plan Adaptation Knowledge

Domain-configurable partial-order knowledge in DCPOP-A is encoded as rules of the following form:

if (+/–) <POP plan element> [,(+/–) <POP plan element>]
then (**do:**/**undo:**) <POP plan refinement> [, (**do:**/**undo:**) <POP plan refinement>]

The conditional part of the rule is a conjunction of one or more POP plan elements as defined in the previous section. These POP plan elements are preceded by either a

plus or a minus. The consequent part is a sequence of POP plan elements, preceded by do or undo symbols. The semantics of a rule are as follows. The rule is satisfied if each of the POP plan elements preceded by a plus sign occurs in the current plan and none of the POP plan elements preceded by the minus sign occur in the current plan. The consequent part indicates each of the POP plan refinements to add, if it is preceded by a *do*, or to delete, if it is preceded by an *undo*. When a plan step s is deleted (i.e., by an undo), any ordering constraint or causal link connecting to/from s is also removed. When an ordering constraint, a causal link, or a binding constraint is deleted, no other plan element is removed. The POP domain-configurable rules, which henceforth we refer to as POP rules, are a natural extension of POP refinements and in fact all POP refinements can be expressed using these rules. We also allow as conditions *same($?x,?y$)* and *different($?x,?y$)* indicating that two variables take the same (or different) value. We write instead $?x = ?y$ (or $?x \neq ?y$) for readability.

We use two classes of POP rules: retraction rules and refinement rules. **Retraction rules** indicate POP plan elements that must be removed from the plan. As a result, they always have the *undo:* label in the consequent part of the rule. ***Refinement rules*** indicate POP plan elements that must be added to the plan. As a result, they always have the *do:* label in the consequent part of the rule. This distinction facilitates the systematic search performed by the adaptation algorithm that will be discussed in the next section.

Table 1: POP rules partially encoding the unstack-stack strategy

(1)	(3)
if $+ s_0 \rightarrow$ (on ?x ?y)	**if** $+ s$: (move ?x ?y table)
$+ s_0 \rightarrow$ (block ?y)	$+ s_0 \rightarrow$ (block ?y)
$- s \rightarrow$ (on ?x table)	$+ s_0 \rightarrow$ (on ?x ?y)
then do: s': (move ?x ?y table)	$- s_0 \rightarrow$ (on ?x ?y) s
do: $s_0 \rightarrow$ (on ?x ?y) s'	**then do:** $s_0 \rightarrow$ (on ?x ?y) s
(2)	(4)
if $+ s$: (move ?x ?y table)	**if** $+ s$: (move ?x table ?y)
$+ s'$: (move ?z table ?w)	$-$ ((on ?x ?y) @ s_∞)
$- s \rightarrow s'$	**then undo:** s:(move ?x table ?y)
then do: $s \rightarrow s'$	

Table 1 shows an example of POP rules in the Blocks World domain. These POP rules encode the strategy, which we call ***unstack-stack***, that first unstacks all blocks to the table and then stacks them in the required configuration. This is the strategy followed to generate the plan in Fig. 4. The first POP rule unstacks block *?x* to the table. The first two conditions check if block *?x* is on top of another block *?y* in the initial state. The third condition checks that no existing step unstacks *?x* to the table. This rule makes two refinements: it adds a step *s′* unstacking *?x* to the table and adds a causal link connecting the step s_0 to achieve a precondition of *s′*. The second POP rule ensures that unstacking steps (e.g., step *s*) are done before stacking steps (e.g., step *s′*). The third POP rule is intended as a refinement of an input plan so that it commits to the encoded strategy. It checks if a block (*?x*) that is unstacked by an step *s* is linked to the condition (*on ?x ?y*) in the initial state. If it is not, it adds a causal link connecting the condition and *s*. This rule can be triggered in situations where in

the initial state of the retrieved plan, block *?x* was on top of a block *?y* and later in that plan *?x* was unstacked to the table by an step *s*. This plan would not have been generated by the strategy encoded in Table 1. The fourth POP rule is a retraction rule. It removes any stacking step from the table that does not achieve a goal.

Any step removed by the fourth rule does not need to be added back because in the stack-unstack strategy, blocks are stacked only to achieve goals. After all these steps are removed, the four POP refinement rules of Table 1 will produce incomplete plans that can be further refined without backtracking on any of the refinements made by applying these rules. This is a highly desirable property as in some domains it might be difficult to obtain a collection of POP rules that produce a complete plan. Consequently, rules can be given for the more computationally complicated details (e.g., how to achieve the goals), leaving the rest to standard POP. Ideally, the intermediate plan produced from adaptation will be easier to complete than the initial plan. The unstack-stack strategy, partially encoded in Table 1, can be fully encoded to ensure that the resulting plans are complete. Furthermore, no backtracking will be needed during the plan adaptation process. Hence, when used in DCPOP-A, these rules will result in an omniscient plan adaptation algorithm. This will be confirmed in the experimental evaluation where no backtracking occurred in any of the plan adaptation instances. We omit presenting all the POP rules due to the lack of space.

Procedure DCPOP-A(S, G, A, π_{old}, R)
//**input**: initial state S, goals G, actions A, plan π_{old}, POP rules R
//**output**: complete plan for (S,G) or fail

1. π_{adj} ← adjust-plan(S, G, π_{old})
2. P ← doAllDCRetractions(π_{adj}, R)
3. **while** (P $\neq \emptyset$) **do**
4. π ← heuristicSelectPlan(P, A)
5.   P ← P − {π}
6.   **if** flaws(π) = $\emptyset$ **then**
7.   **return** π
8.   **else**
9.   P' ← doOneStepDCRefinements(π, R)
10.   **if** (P' = $\emptyset$) **then**
11.   f ← heuristicSelectFlaw(π)
12.   P ← P $\cup$ refinements(π, f, A)
13.   **else**
14.   P ← P $\cup$ P'
15. **return fail**

Fig. 5: Pseudo-code of DCPOP-A

4.3 Domain-Configurable Partial-Order Plan Adaptation Algorithm

Fig. 5 presents the pseudocode of the proposed plan adaptation algorithm on top of POP. It receives as input the initial state, goal state, and actions. It also receives the plan to be adapted, π_{old}, and the POP rules R (as described in Section 4.2). The output is a complete plan solving (S,G,A) or fail if none is found. DCPOP-A begins by adjusting π_{old} relative to (S,G) (Line 1). Adjust plan works by repeatedly (1) removing a step s that mentions objects in the retrieved plan that are not mapped into

objects in the new problem, and (2) removing any ordering constraint or causal link connecting to/from s. This is a common step for adaptation in first-principles POP planning (e.g., [24], [13], [25]). Then, a set of plans is found by repeatedly applying retraction rules in R until none is applicable (Line 2). These plans are added to P, the list of current candidate plans to be refined. The next part of the pseudocode continues iterating while there is at least one candidate plan to be refined and no solution has been found (lines 3-14). When the list of candidate plans is empty, a failure is returned (Line 15). At each iteration, a candidate plan π is selected using the heuristics and is removed from P (lines 4 and 5). If this candidate plan has no flaws, it is returned (lines 6 and 7). Otherwise each plan computed by applying an applicable POP rule to π is added to P (lines 9 and 14). If no domain configurable refinements are found, standard POP refinements are added to P (lines 11 and 12). In principle, DCPOP-A could use any relevant plan and flaw selection heuristics described in [26] for lines 4 and 11; however our implementation uses last-in-first-out selection for both plans and flaws.

5   Example of Domain-Configurable Plan Adaptation

Fig. 6 shows an example of a plan obtained by adapting the plan from Fig. 4 using the unstack-stack strategy partly encoded in Table 1. The new problem has almost the same initial state as before with the exception that block F does not exist, and there are several differences in the goals. Underlined steps indicate steps retained from the retrieved plan. Continuous lines indicate causal links and ordering constraints retained from the retrieved plan (only a subset is shown). Dashed lines indicate new causal links and ordering constraints added (only a subset is shown). The steps marked "x" in Fig. 4 are steps that have been removed; The steps *move*(*E,F,table*) and *move*(*F,table,E*) were removed by adjust-plan because F does not occur in the new problem. The step *move*(*C,table,A*) was removed by the fifth POP rule because it is inconsistent with the goal. The step *move*(*B,table,C*) was added by the third POP rule because it achieves a goal. In this specific example the maximum possible number of steps is retained from the retrieved plan. In general, this is not the case because steps that could have been retained to form a complete plan will be removed if they are inconsistent with the unstack-stack strategy.
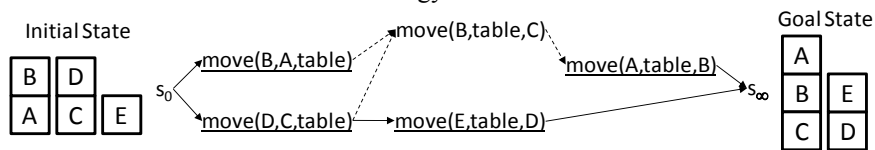

Fig. 6. Adapted partial-order plan

Recall from the example in Fig. 3 that Case 1 solves (*state 1*, *final*) and Case 2 solves (*state 2*, *final*). A caveat must be made that the search space in Fig. 3 represents states of the world whereas DCPOP-A's search space is a space of plans. However a mapping can be made from the state space to the plan space such that any transition made between states represents the corresponding action being added to the plan in the transition between plans. Continuing with the example, if we apply the

unstack-stack strategy to the new problem (*state 3*, *final*), then for both cases it will take the path for node p1 (meaning it will unstack all blocks). If Case 1 is being adapted then it will follow the plan laid out from p1 all the way to the goal state. If Case 2 is being adapted then it will add the step stacking C on D from p1 and then continue the rest of the plan from Case 2, which stacks the remaining blocks B and A in that order. So it takes 2 refinements if Case 1 is reused and 3 refinements if Case 3 (solves problem (*state 3*, *final*)) is reused and no exploration of failed nodes is made.

6   Empirical Evaluation

We performed experiments by encoding POP rules for the logistics transportation domain and blocks world. In the logistics transportation domain, packages must be relocated into target locations. There are two transportation means: trucks, which can be used to relocate packages within locations in the same city and airplanes, which can be used to relocate packages that are in different cities. The blocks world is a puzzle-like domain in which piles of blocks on a table must be reconfigured into a target configuration. The basic restriction is that blocks can only be moved either from the top of a pile to the top of another pile or to the table. We encoded the unstack-stack strategy described in Section 4.2 for the blocks world; therefore we attained an omniscient plan adaptation algorithm.

6.1   Transportation Domain Encoding

For the logistics transportation domain we encoded the following basic strategy:

1. We remove steps from the retrieved plan that load and unload any packages not at the destination city. So for example if a package p_5 needs to be relocated to loc_2 in $city_3$, then any load and unload steps of that package in any city other than $city_3$ will be removed. This eliminates potential threats that would cause backtracking.
2. We take advantage of any steps in the plan relocating a package into the destination location in the destination city by keeping those steps and adding connecting steps as needed. So, for example, if the retrieved plan relocates p_5 from a location loc_6 in $city_3$ to loc_2 but in the new problem the package begins at $airport_4$ in $city_3$, then steps are added that relocate p_5 into loc_6 from $airport_4$.
3. We added steps to the plan that relocate packages to the destination city if needed, taking advantage of any existing steps driving a truck or flying an airplane whenever possible. So for example, if p_5 was initially in loc_7 in $city_2$, then it will be relocated to an airport in $city_2$. If an existing step in the plan already moves p_5 from loc_7 to an airport in $city_2$, this step will be reused. Otherwise a new step will be created. Steps are also added relocating p_5 from $airport_1$ to $airport_4$. If an existing step in the plan flies p_5 from $airport_1$ to $airport_4$ it will be reused. Otherwise a new step is created.

These encodings ensure that DCPOP-A will be omniscient when solving problems in the logistics transportation domain.

6.2   Experimental Setup

For each domain we constructed a case base of 100 cases and a testing set of 10 problems. All problems have the same goals but their initial state is randomly generated. For the blocks world the goal is to achieve a 5-block pile and for logistics a particular configuration of 4 packages required to be at 4 different locations. The initial state for the blocks world is a configuration of the 5 blocks. So the total number of problems that can be generated is 501. The initial state for the transportation domain is a configuration of 3 cities, each having 3 locations (including 1 airport), each city has 1 truck and there are 2 airplanes. So the total number of problems that can be generated, given that the packages can start in any of the 9 locations and that the start locations of the trucks and airplanes are fixed, is 6561. For each problem p in the testing set we adapt each of the cases c stored in the case base. We run each problem-case pair (p,c) 30 times and average the results. So the total runs for each domain was $10 * 100 * 30 = 30,000$ runs.

Fixing the goals is a simplifying way to simulate how our retrieval algorithms would work with an omniscient adaptation algorithm. Namely, we will simply retrieve any case that achieves the same goals regardless of the similarity. In experiments reported in [27], it is shown that modifying features in the initial state can result in a significant change in the adaptation process on top of a partial order planner. For historical context, in Prodigy/Analogy [1] retrieval occurs by iterating two steps. At the first step the system uses a hash table to identify if there are cases stored achieving the same number goals and then, in the second step, computes similarity based on the initial state. If a sufficiently similar case is found (e.g., the similarity of the initial states is greater than a pre-defined threshold) then the case is retrieved. Otherwise it repeats the two steps by removing one goal. With an omniscient adaptation algorithm the second step would be unnecessary. A similar process to Prodigy/Analogy is performed in CAPlan/CbC and derSNLP.

6.3   Results

Fig. 7 shows the run-time results for the blocks world (left) and the logistics transportation domain (right) respectively. The x axis corresponds to the 100 cases * 10 problems and the y axis correspond to the average time in seconds over the 30 runs for each (case, problem) adaptation process. The x-axis is sorted so the first 100 averaged data points are shown with the first given problem, then again the next 100 points with the second given problem, and so forth. Thus, the vertical bars in the graphs separate data for each of the 10 problems; between those bars (i.e., for a given problem), the data points show the averaged times to adapt each of the cases in the CB into a solution for the problem. In the blocks world domain, we observe that the running times for adapting each case to a given problem is clustered around the same time intervals. For example, for the 4th problem the average time to adapt all cases is 0.155 seconds with a standard deviation of 0.012 seconds. We observed similar results across all other problems. In the logistics domain, there is no significant time difference between solving times across all given problems; the average problem solving time, across all pairs (case, problem), is 9 seconds with a standard deviation

of 0.5 seconds. The results for both domains support our hypothesis that regardless of which any two cases are retrieved for a given problem, their adaptation times will be roughly the same, regardless of the individual cases similarity to the new problem, and given that the problems solved by the cases and the new problem are of the same size.
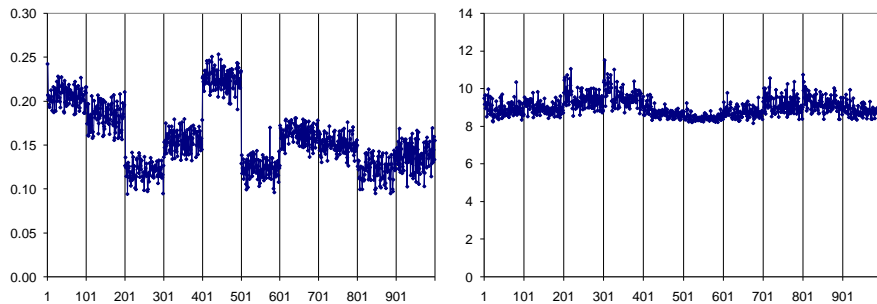

Fig. 7. Adaptation times for blocks world (left) and logistics (right)

7   Conclusions

In this paper we revisited the adaptation and retrieval trade-off traditionally held as a principle in case-based reasoning research and even practice. We argue that this principle involves an implicit assumption that the adaptation algorithm is capable of solving problems fast enough only when the retrieved case is reasonably similar to the input problem. If the retrieved case is not sufficiently similar to the new problem then the adaptation effort will take a significant amount of time. To make this assumption explicit, we analyzed this principle under two boundary conditions: (1) for naïve and (2) for omniscient adaptation algorithms. Using a simple complexity analysis, we conclude the adaptation-retrieval trade-off does not necessarily exist for the naïve adaptation procedures. We also claim that the adaptation-retrieval trade-off does not necessarily hold for connected domains, and validated this hypothesis empirically on two classical connected domains used widely in the case-based planning literature.

A provocative implication of our results is that, because for omniscient plan adaptation there is no adaptation-retrieval trade-off, we can substantially reduce the case base by simply having one case for each combination of goals indexed by a hash table so that the retrieval procedure would run in constant time by simply identifying the case that achieves the same goals (or even use an empty CB and plan from scratch). However, running time is but one criterion by which we can measure the effectiveness of the overall case-based reasoning process. Other, arguably at least as important criteria, such as quality of the resulting plan, should be considered. Indeed, one of the motivational scenarios for case-based reasoning is a case base storing a collection of hand-crafted solutions. In this scenario, the retrieval task is to find a very similar case, if not the most similar one, and the adaptation task is to commit to the retrieved plan as much as possible. Throwing away half of the solution, as encoded in the unstack-stack strategy, would not make any sense in this scenario regardless of

how fast the adapted solution is generated. Instead we envision highly tuned POP rules that are sensible towards retaining crucial steps identified by the user and retrieval procedures that ensure that plans meeting certain constraints are produced, as recent research on retrieval has suggested (e.g., [28], [29]).

For future work we are planning to investigate the adaptation-retrieval trade-off for non connected domains such as the logistics transportation domain with one-way routes. Unlike connected domains, there is no guarantee that the adjusted plan can always be extended to reach a solution. Hence, the question is whether POP rules can be written that rapidly remove parts of the adjusted plan in such a way that (1) a significant portion of the retrieved plan is reused in the adjusted plan and (2) this adjusted plan can rapidly be refined to obtain a solution.

References

1. Veloso, M. M.: Planning and Learning by Analogical Reasoning. Springer Verlag (1994)
2. Muñoz-Avila, H., Weberskirch, F.: Planning for Manufacturing Workpiececs by Storing, Indexing and Replaying Planning Decisions. In: AIPS-96, pp. 150-157. AAAI Press (1996)
3. Ihrig, L., Kambhampati, S.: Storing and Indexing Plan Derivations through Explanation-Based Analysis of Retrieval Failures. JAIR 7, 161-198 (1997)
4. Koehler, J.: Avoiding Pitfalls in Case-based Planning. In: AIPS-94, pp. 104–109. (1994)
5. Francis, A., Ram, S.: A Comparative Utility Analysis of Case-based Reasoning and Control-rule Learning Systems. In: ECML-95. (1995)
6. Gomes, P., Pereira, F. C., Seco, N., Pavia, P., Carreiro, P., Ferreira, J. Bento, C. Combining Case-Based Reasoning and Analogical Reasoning in Software Design. In: Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science. (2002)
7. Smyth, B., and McKenna, E. Footprint-Based Retrieval. In ICCBR-99. Springer (1999)
8. Chang H., Dong L., Liu F., Lu W., Indexing and Retrieval in Machining Process Planning Using Case-Based Reasoning. Artificial Intelligence in Engineering, Vol. 14 (2000)
9. Bonzano, A., Cunningham, P., Smyth, B.: Using Introspective Learning to Improve Retrieval in CBR: A Case Study in Air Traffic Control. In: ICCBR-97. LNCS, vol. 1266, pp. 291-302. Springer (1997)
10. Stahl, A., Gabel, T. Using Evolution Programs to Learn Local Similarity Measures. In Proceedings of ICCBR 2003, Trondheim, Norway. Springer (2003)
11. Ihrig, L., Kambhampati, S.: Design and Implementation of a Replay Framework Based on a Partial Order Planner. In: AAAI/IAAI-96, pp. 849-854. AAAI Press (1996)
12. Muñoz-Avila, H., Weberskirch, F.: A Case Study on the Mergeability of Cases with a Partial-Order Planner. In ECP-97. LNCS, vol. 1348, pp. 325-337. Springer (1997)
13. van der Krogt, R., de Weerdt, M.: Plan Repair as an Extension of Planning. In ICAPS-05, pp. 161-170. AAAI Press (2005)
14. Gerevini, A., Serina, I.: Fast Plan Adaptation through Planning Graphs: Local and Systematic Search Techniques. In: Proc. of AIPS-00, pp. 112-121. AAAI Press (2000).
15. Warfield, I., Hogg, C., Lee-Urban, S., Muñoz-Avila, H.: Adaptation of Hierarchical Task Network Plans. In: FLAIRS-07, pp. 429-434. (2007)

16. Goel, A., Ali, K., Donnellan, M., Gomez, A., Callantine, T. Multistrategy Adaptive Navigational Path Planning. IEEE Expert, 9(6):57-65, December (1994)
17. Tonidandel, F., Rillo, M.: Case Adaptation by Segment Replanning for Case-Based Planning Systems. ICCBR 2005: 579-594 (2005)
18. Fikes, R., Nilsson, N.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence 2, 189-208 (1971)
19. Nau, D., Au, T., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J., Wu, D., Yaman, F.: Applications of SHOP and SHOP2. IEEE Intelligent Systems 20(2), 34-41. (2005)
20. Knoblock, C.: Generating Parallel Execution Plans with a Partial-Order Planner. In: AIPS-94, pp. 98-103. AAAI Press (1994)
21. Paulokat, J., Wess, S.: Planning for Machining Workpieces with a Partial-Order, Nonlinear Planner. In: Proceedings of the AAAI 1994 Fall Symposium on Planning and Learning. AAAI Press (1994)
22. Nguyen, X., Kambhampati, S.: Reviving Partial Order Planning. In: IJCAI-01, pp. 459-466. Morgan Kaufmann (2001)
23. Vidal, V., Geffner, H.: Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming. Artificial Intelligence 170(3), 298-335. (2006)
24. Hanks, S., Weld, D. S.: A Domain-Independent Algorithm for Plan Adaptation. JAIR 2, 319-360 (1995)
25. Kuchibatla, V., Muñoz-Avila, H.: An Analysis on Transformational Analogy: General Framework and Complexity. In: ECCBR-06. LNCS, vol. 4106, pp. 458-473. Springer (2006)
26. Younes, H., Simmons, R.: VHPOP: Versatile Heuristic Partial Order Planner. Journal Of Artificial Intelligence Research 20, pp. 405-430. (2003)
27. Muñoz-Avila, H., Hüllen, J.: Feature Weighting by Explaining Case-Based Planning Episodes. In: EWCBR-96. LNCS, vol. 1168, pp. 280-294. Springer (1996)
28. McSherry, D.: Completeness Criteria for Retrieval in Recommender Systems. In: ECCBR-06. LNCS, vol. 4106, pp. 9-29. Springer (2006)
29. Nicholson, R., Bridge, D., Wilson, N.: Decision Diagrams: Fast and Flexible Support for Case Retrieval and Recommendation. In: ECCBR-06. LNCS, vol. 4106, pp. 136-150. Springer (2006)