Computational Intelligence, Volume 21, Number 3, 2005

HYBRID ACE: COMBINING SEARCH DIRECTIONS FOR HEURISTIC PLANNING

DIMITRIS VRAKAS AND IOANNIS VLAHAVAS

Department of Informatics, Aristotle University of Thessaloniki, 54124, Greece

One of the most promising trends in Domain-Independent AI Planning, nowadays, is state-space heuristic planning. The planners of this category construct general but efficient heuristic functions, which are used as a guide to traverse the state space either in a forward or in a backward direction. Although specific problems may favor one or the other direction, there is no clear evidence why any of them should be generally preferred. This paper presents Hybrid-AcE, a domain-independent planning system that combines search in both directions utilizing a complex criterion that monitors the progress of the search, to switch between them. Hybrid AcE embodies two powerful domain-independent heuristic functions extending one of the AcE planning systems. Moreover, the system is equipped with a fact-ordering technique and two methods for problem simplification that limit the search space and guide the algorithm to the most promising states. The bi-directional system has been tested on a variety of problems adopted from the AIPS planning competitions with quite promising results.

Key words: planning; heuristic search; bi-directional search.

1. INTRODUCTION

Motivated by the work of Drew McDermott in the mid-90's on heuristic state-space planning, a number of researchers turned to this direction. During the last few years a great amount of work has been done in the area of domain-independent, state-space, heuristic planning, and a significant number of planning systems with remarkable performance were developed.

Hector Geffner in his recent work on HSP-2 (Bonet and Geffner 2001) studies the matter of search direction. The HSP-2 planning system enables the user to decide for the direction of the search. It is clear from the experimental results that there are specific problems, which favor one or the other search directions, but in general there is no clear evidence why any of the two directions should be preferred.

In this paper, we present a planning system for domain-independent, heuristic planning called Hybrid AcE that combines both progression (forward chaining) and regression (backward chaining). The system utilizes an improved version of the bi-directional search strategy of the BP (bi-directional planner) planning system among with two powerful heuristic functions based on the AcE (action evaluation) planning system.

The search begins from the *Initial State* and proceeds with a weighted A* search until the heuristic function is no longer capable of guiding the search. At that point the algorithm changes direction and regress the *Goals* trying to reach the best state found at the previous step. The direction of the search may change several times before a solution can be found. Each time the system changes direction, the appropriate heuristic is reconstructed and this enables the system to update the heuristic functions when needed and attack each part of the problem searching in the most appropriate direction.

The heuristic functions of Hybrid AcE work in two phases. The larger part of the calculations is performed off-line in a preplanning phase where each action is graded based on the goals of the problem. During search, these grades are used for estimating the distances between intermediate states and the goals. Apart from the heuristic functions, Hybrid AcE embodies two powerful domain-independent heuristic functions extending the heuristic function of the AcE planning system.

The planning system is also equipped with two methods, which reduce the quantity of information provided to the search algorithms to the minimum required to solve the problem.

These techniques manage to cut down the branching factor of the search without affecting the completeness of the planning system. As shown by experimental results, the application of the simplification methods results in less planning time and shorter plans at the same time.

The paper also presents experimental results of Hybrid AcE on a large number of problems from the international planning competitions and planning bibliography and compares the system with two single-direction planners utilizing the heuristics of Hybrid AcE. The experimental results illustrate the efficiency of the planning system and show that its hybrid search strategy is a very good option for modern heuristic planners.

The rest of the paper is organized as follows: Section 2 gives some background information concerning planning under the STRIPS (Fikes and Nilsson 1971) notation. Section 3 provides a brief review of the work related to state-space, heuristic planning, and other approaches to bi-directional planning. Section 4 describes the bi-directional search strategy ibr/n detail and deals with certain issues that arise while regressing the goals of a problem. Sections 5 and 6 describe in detail the heuristic functions of Hybrid AcE and the two fact-ordering techniques used to further refine the heuristics. Section 7 presents two techniques that work on top of the search modules simplifying the definition of the problems and thus making the problems easier to solve. Section 8 presents experimental results that illustrate the efficiency of the planning system on a large variety of problems, while Section 9 concludes the paper and poses future directions.

2. PRELIMINARIES

State-space planners usually adopt the STRIPS (Stanford Research Institute Planning System) notation. A planning problem in STRIPS is a tuple $<I,A,G>$ where I is the initial state, A a set of available actions, and G a set of goals.

States in STRIPS are represented as sets of atomic facts. All the aspects of the initial state of the world, which are of interest to the problem, must be explicitly defined in I. State I contains both static and dynamic information. For example, I may declare that object John is a truck driver and there is a road connecting cities A and B (static information) and also specify that John is initially located in city A (dynamic information). State G on the other hand, is not necessarily complete. G may not specify the final state of all problem objects even because these are implied by the context or because they are of no interest to the specific problem. For example, in the logistics domain the final location of means of transportation is usually omitted, because the only objective is to have the packages transported. Therefore, there are usually many states that contain the goals, thus in general, G represents a set of states rather than a simple state.

Set A contains all the actions that can be used to modify states. Each action A_i has three lists of facts containing:

(a) The preconditions of A_i (noted as prec(A_i)),
(b) The facts that are added to the state (noted as add(A_i)), and
(c) The facts that are deleted from the state (noted as del(A_i)).

The following formulae hold for the states in the STRIPS notation:

- An action A_i is applicable to a state S if $\text{prec}(A_i) \subseteq S$.
- If A_i is applied to S, the successor state S' is calculated as:

$$S' = S \backslash \text{del}(A_i) \cup \text{add}(A_i)$$

- The solution to such a problem is a sequence of actions, which if applied to I leads to a state S' such as $S' \supseteq G$.

Usually, in the description of domains, action schemas (also called operators) are used instead of actions. Action schemas contain variables that can be instantiated using the available objects and this makes the encoding of the domain easier.

3. RELATED WORK

One of the most promising trends in domain-independent planning was presented over the last few years. It is based on a relatively simple idea where a general domain-independent heuristic function is embodied in a heuristic search algorithm such as Hill Climbing, Best-First Search, or A*. A detailed survey of search algorithms can be found in Korf (1998). Examples of planning systems in this category are UNPOP (McDermott 1996), the ASP/HSP family (Bonet and Geffner 1999; Bonet and Geffner 2001; Bonet, Loerincs, and Geffner 1997), GRT (Refanidis and Vlahavas 1999), AltAlt (Nguyen, Kambhampati, and Nigenda 2000), FF (Hoffmann 2000), and AcE (Vrakas and Vlahavas 2002).

These planners rely on the same idea to construct their heuristic function. They first relax the planning problem by ignoring the delete lists of the domain operators. Then they start either from the *Initial s*tate or the *Goals* they construct a leveled graph of facts, noting for every fact f the level $L(f)$ at which it was first achieved. To evaluate a state S, the heuristic function takes into account the values of $L(f)$ for each $f \in S$.

The first planner in this category was UNPOP (McDermott 1996), a regression planner that constructed at each step a graph, named *greedy regression-match graph*. The graph was used in the search, for creating the heuristic function and cutting down the branching factor by pruning certain actions.

The direct descendant of UNPOP was HSP (Bonet et al. 1997), a similar planning system, which searches the state space in the forward direction though and constructs a more sophisticated heuristic function from a similar graph. HSP was followed by HSP-R (Bonet and Geffner 1999), a similar planer with two main differences from HSP. The search is performed in a backward manner and the heuristic is created in the opposite direction, which enables HSP-R to create the planning graph only once. HSP and HSP-R were later embodied in a unifying planning system called HSP-2 (Bonet and Geffner 2001). GRT (Refanidis and Vlahavas 1999) is another extension to HSP that searches in the forward direction and creates the heuristic backward once at the beginning.

Nigenda, Nguyen, and Kambhampati presented a hybrid planning system, named AltAlt (Nguyen et al. 2000), which was created using programming modules from STAN (Long and Fox 1998) and HSP-R. In a first phase, AltAlt uses the module from STAN to create a planning graph, from which it extracts a heuristic function that is used to guide the backward hill-climbing search, which is performed in an HSP-R manner.

One of the latest planners in this category and the most effective according to the results of the AIPS-00 planning competition is Hoffmann's FF planning system (Hoffmann 2000). FF constructs a graph similar to this of GRAPHPLAN from which it extracts a sketch plan. The sketch plan is then used in a forward enforced hill-climbing search in two ways. Firstly, the length of the sketch plan is used as an estimate for the distance between the *Initial* state and the *Goals* and secondly a set of *helpful* actions is extracted, which helps in cutting down the branching factor of the search.

AcE (Vrakas and Vlahavas 2002) is a relatively new planner, which creates the heuristic once in the forward directions and then utilizes it in a backward search. The innovation of AcE relies in the fact that the heuristic calculates distances between actions and states, rather than facts and states. This enables the heuristic to track more interactions between independent facts.

Bi-directional search is a well-known search strategy mentioned in almost any textbook about Artificial Intelligence. However, it has not been broadly adopted as a search strategy. Especially in planning, there are only a few systems performing a combined search in both directions. These include PRODIGY (Veloso et al. 1995), NOLIMIT (Veloso 1994), FLECS (Veloso and Stone 1995), and RASPUTIN (Fink and Blythe 1998), which have been developed by researchers of the Carnegie Mellon University's PRODIGY project and are based on the combination of goal-directed backward chaining with simulation of plan execution (Veloso 1994). Although these planners perform some kind of search in both directions, they are actually forward-direction planners, which utilize the backward search as an action selection mechanism.

However, recently Vrakas and Vlahavas (2001) presented the BP planning system, which combines search in both directions. BP starts solving the problem in the forward direction until it reaches a point at which the heuristic cannot further guide the search. Then the planner changes direction, constructing from scratch the opposite heuristic, trying to reach the state where the previous search stopped. The planner may change direction several times before a solution is found. The BP was compared to two single-direction planners, utilizing the same heuristics as BP and proved to be more stable and better both in terms of planning time and plan lengths.

4. THE SEARCH STRATEGY OF HYBRID ACE

The planners presented in the previous section have shown quite impressive performance and they have proved to be able to handle a large variety of difficult problems. However, their performance is unstable and they frequently present precessions in their efficiency among different domains or even between problems of the same domain.

There are two main reasons for that behavior:

(a) Although the heuristic functions constructed by all the planners are general, they seem to work better with specific domains.
(b) There are domains and problems that clearly favor one of the two search directions (forward or backward).

The first argument, which is also a conclusion drawn from the experience of the authors, has been stated by Stone, Veloso, and Blythe (1994).

The second argument is the main conclusion drawn by Bart Massey in an extensive study in the directions of planning presented in Massey (1999). Bonet and Geffner have pushed the same argument one step further: "*Although we don't fully understand yet when HSP will run better than HSP-R, the results suggest nonetheless that in many domains a BP combining HSP-R and HSP could probably do better than each planner separately*" (Bonet and Geffner 1999). The answer to the question posed by Bonet and Geffner above has been answered by Massey in (Massey 1999), where the planning problems are discriminated into forward and backward problems, in the sense that strongly directed planners will find the problems of the opposite direction intractable.

Motivated by the conclusions stated above we developed Hybrid AcE, a heuristic state-space planner, which combines search in both directions. A part of the plan is constructed with the progression module (forward chainer) and the rest is constructed with the regression module (backward chainer). The subplan of the regression module is inversed and merged with that of the progression module to produce the final plan. However, the case is not always that simple, because usually the planner interleaves the execution of both modules several times before a solution is found. Each search module continues searching until a condition (noted as *switching criterion*) is met. The switching criterion presented below, urges the planner to change direction when the latter encounters either a dead end (Part A) or a plateau (Part B). A dead end is a situation where given a specific point in the search space there seem to be no promising paths to follow. This may occur either because the planner has made a wrong choice somewhere in the past and needs to backtrack or the heuristic function is not able to estimate the successor states correctly. A plateau on the other hand is a situation where all the states in the neighborhood seem to be of equal importance and the heuristic function cannot choose among them.

In the switching criterion presented below and also in the rest of the article, the term Agenda refers to a memory structure used to store all the states of the search space that have been encountered by the search algorithm, but not yet expanded. Actually, the Agenda also stores information about the steps needed to achieve each state of the Agenda, starting from the initial state. The term heuristic function h, also used in the criterion and throughout the article, refers to a function that estimates the distance between any given intermediate state S and the final state of the search (G in case of forward chaining). The estimations provided by the heuristic function are used as a means of selecting the next state for expansion, because the one with the lower heuristic value is one that is expected to be closer to the final state.

Switching Criterion

Let S' be the best state found by the search module (forward or backward) so far, S the best state in Agenda, T and m two thresholds set by the user and $h(S_i)$ the value of the heuristic function for state S_i.
If

$$h(S) > h(S') + T \quad \forall S : S \in Agenda \qquad \text{[Part A]}$$

OR

$$|B| > m^{*}|Agenda|, \text{ where} B \subseteq Agenda \text{ and } h(S_k) = h(S)\,\forall S_k \in B \qquad \text{[Part B]}$$

THEN

Change search direction

Following the previous discussion the criterion has two parts that identify dead ends and plateaus, respectively:

(a) The simplest way to identify a dead end is to check for child states that are closer to the goals than the current state. However, this is a very strict criterion, which would lead to constant reporting of false dead ends. Therefore, part A of the switching criterion adopts a more relaxed condition which allows the search to proceed with states that are worse than the current one, as long as they do not draw away from the goals too much, as defined by the constant T. In other words, given a starting state S, a move to a successor state S' is acceptable if $h(S') < h(S) + T$. Setting the value of T to 0 we get the strict criterion that only allows the search to move to more promising states.

(b) To identify a plateau the criterion checks the estimated distances $h(S_i)$ of all the states in the Agenda. If there is one state S_b which is clearly more promising than all the others ($h(S_b) < h(S_k)\,\forall S_k \in$ Agenda $- S_b$), then the algorithm proceeds with S_b.

If there are more than one "best" states in the Agenda then the algorithm must choose one randomly. As the number of "best" states increases it means that the heuristic function is not longer able to guide the search. The parameter m of the switching criterion defines the maximum allowed number of "best" states in relation to the total number of states in the Agenda. For example, if $m = 0.2$ then the criterion is met if at least 20% of the states in the agenda are identified as "best" according to the heuristic function.

Details about the search strategy are presented later in this chapter after a thorough description of the two search modules (progression and regression).

4.1. The Progression Module

The progression module employs a Best-First search method that starts from the initial state and moves forward trying to reach the goals. It is worth noting here that the initial state and the goals refer to the specific subproblem that is passed to the progression module and not necessary to the initial problem.

The progression module takes four arguments, which are: (a) the initial state I' of the subproblem, (b) the goals G' of the subproblem, (c) the maximum size *SOF_AGENDA* of the planning agenda, and (d) a heuristic function h capable of estimating distances between states. The progression module returns a new state S, which is the state closer to G' that the module could find. Figure 1 illustrates the algorithm of the progression module. In Figure 1, I' and G' refer to the initial state and the goals of the subproblem, respectively. Variables G and S' are used to store the current state and the successor state in each iteration, while K stores the best state found so far by the search module.

The progression module employs a simple forward Best-First search strategy with three main differences:

(a) The size of the planning agenda is limited by an upper limit *SOF_AGENDA*. This means that if there are N states ($N >$ *SOF_AGENDAz*) that should be stored in the Agenda, only the SOF_AGENDA most promising (according to h) states will be stored and the rest will be discarded.

(b) The search modules use two closed lists, noted as *Local* and *Global*, respectively. The scope of the *Local Close list* is limited to the current instance of the module. Each time the planner changes direction it is reset. However, the *Global Close list* retains its contents throughout the whole planning process and it is common for both search modules. The purpose of the *Global* list is not to allow the module to return subproblems (initial state and goals) that have been examined in the past. Note here, that the module is allowed to return an initial state that has been examined in the past, as long as the goals have changed.

(c) The progression module will stop the search when it has either solved the subproblem or the *Switching criterion* has been met. In the first case the planner will stop and return the merged plan, while in the second one the system will switch to the regression module.

4.2. The Regression Module

The algorithm of the regression module is quite similar to that of the already described progression module, because the search strategy is symmetric. However, there are certain key points that need to be clarified. These points refer to common problems caused to regression planners by the representation of the goals. The main problem is that regression planners do not deal with states, as progression planners do, but with sets of states. This originates from

```
Progression Module
Input: I', G', SOF_AGENDA, h , Output: S
Set Agenda=[I'], K=I', Local Closed List=[]
While (Agenda $\neq \emptyset$)
begin
    G = the first state in the Agenda
    If G' $\subseteq$G  Return G
    If G$\notin$ Local Closed List
    begin
        If the Switching criterion is met Return (K)
        If h(G)<h(K) and {G',G} not in Global Close list then K=G
        For each action A' : prec(A')$\subseteq$G
        begin
            S'=G$\cup$ add(A')\del(A')
            Add S' to the Agenda sorted by h(S')
            If size of Agenda>SOF_AGENDA
                remove the last element from Agenda
        end
        Add G to the Local Closed List
    end
    Remove G from Agenda
End
Return K
```

FIGURE 1. The progression module of Hybrid AcE.

the fact that the goals do not usually form a complete state description. Therefore, a more sophisticated technique than simply reversing the actions, as done in GRT (Refanidis and Vlahavas 1999), is necessary to regress the goals.

The regression module makes extensive use of binary mutual exclusions between facts (Blum and Furst 1995). Two facts p and q are said to be mutual exclusive, denoted as $mx(p,q)$, if no valid state contains both of them. Mutual exclusions are calculated in a way similar to the one adopted by GRAPHPLAN (Blum and Furst 1995). Hybrid AcE progressively builds a graph of facts noting at the same time the mutual exclusions between them using the following formulae:

$mx(p,q)$ if:

$\forall$ action A': $p \in \text{add}(A') \rightarrow q \in \text{del}(A')$ and
$\forall$ action A': $q \in \text{add}(A') \rightarrow p \in \text{del}(A')$

$mx(p,q)$ *if:*

$\forall$ action A': $p \in \text{add}(A')$ and $\forall$ action B': $q \in \text{add}(B') \rightarrow \exists\, (x,y)$: $x \in \text{prec}(A')$,
$y \in \text{prec}(B')$, $mx(x,y)$

Bonet and Geffner in HSP-R define two criteria for identifying whether an action A is backwards applicable to a state S':

(a) Relativeness: $\mathrm{add}(A) \cap S' \neq \emptyset$
(b) Consistency: $\mathrm{del}(A) \cap S' = \emptyset$

The consistency check is not very strict and frequently HSP-R encounters invalid states. To prune these states, HSP-R checks to see if they contain pairs of facts that are mutually exclusive.

Hybrid AcE extends the consistency check of HSP-R to prune the actions that lead to invalid states before they are applied to the current state. More specifically, an action A is backwards applicable to a state S' if all the facts that are directly or indirectly added by A are consistent with the facts in S'. A fact f is indirectly added by an action A if it must hold to apply A ($f \in \mathrm{prec}(A)$) and it is not destroyed by A ($f \notin \mathrm{del}(A)$). Thus, the criteria for backwards applicability in Hybrid AcE can be formed as:

Regressibility Test

An action A is backward applicable to state S if:

1. $\mathrm{add}(A) \cap S' \neq \emptyset$
2. $\mathrm{del}(A) \cap S' = \emptyset$
3. $\neg\exists\,(q,p)$: $q \in \mathrm{add}(A), p \in S', \mathrm{mx}(q,p)$
4. $\neg\exists\,(q,p)$: $q \in (\mathrm{prec}(A) - \mathrm{del}(A)), p \in S', \mathrm{mx}(q,p)$

State S' that is produced after the backward application of action A to state S, is given by the following formula:

State Regression

$$S' = S - \mathrm{add}(A) \cup \mathrm{prec}(A)$$

The algorithm of the regression module is outlined in Figure 2. The only differences between the algorithms of the two search modules lie in the way they identify the applicable actions and produce the successor states. The other parts of the algorithm are identical:

(a) It employs a Best-First search strategy with limited agenda.
(b) It has a *Local Close* list and also makes use of the *Global Close* list.
(c) The module stops either when it has solved the subproblem or the *Switching criterion* is met.

In Figure 2, I' and G' refer to the initial state and the goals of the subproblem, respectively. Variables G and S' are used to store the current set of subgoals and the successor set in each iteration, while K stores the best set of subgoals found so far by the search module.

4.3. Combining the Two Search Modules

The underlying framework of the bi-directional search strategy is based on a relatively simple idea. Very frequently, single-directional planners reach a point in the search process where they cannot proceed effectively. Two of the main reasons for that behavior are: (a) the branching factor of the current subproblem becomes very large and (b) the subproblem

```
Regression Module
Input: I', G', SOF_AGENDA, h, Output: S
Set Agenda = [G'], K=G', Closed List=[]
While (Agenda ≠ ∅)
begin
    G = the first state in the Agenda
    If G⊆I' Return G
    If G∉Local Closed List
    begin
        If the Switching criterion is met Return (K)
        If h(G)<h(K) and {I',G'} not in Global Close list then K=G
        For each fact f in G
        begin
            For each action A' that has f in its add list
            begin
                If (del(A')∩G=∅)
                begin
                    Set E=prec(A')-del(A')∪add(A')
                    If there are no mutual exclusions between facts in E and G
                    Begin
                        S'=S∪prec(A')-del(A')
                        Remove double entries from S'
                        Add S' to the Agenda sorted by h(S')
                        If size of Agenda>SOF_AGENDA
                            remove the last element from Agenda
                    end
                end
            end
        end
        Add G to the Local Closed List
    end
    Remove G from Agenda
End
Return K
```

FIGURE 2. The regression module of Hybrid AcE.

is much too complex and the heuristic function becomes obsolete as the search goes on, especially when it is constructed only once at the beginning.

On the contrary, Hybrid AcE uses the *Switching criterion* to identify when the heuristic function of one of the two search modules is not further able to guide the search. In that case, the planner changes direction, reconstructs the appropriate heuristic function, and calls the corresponding search module. Thus the system keeps the heuristic functions updated and in long-term attacks the larger part of the problem from the direction that best suits it. The Switching criterion makes use of two thresholds m and T. Threshold m is set by the user and remains constant during the whole planning process. For the experiments we used the value of 0.1 for m (maximum allowed percentage of best states in the Agenda). Threshold T on the other hand, is dynamically updated during the planning process. It is initially set to T_{min} and it is changed every time a module completes its execution. If the module managed to improve its initial state then T is reset to T_{min}, otherwise T is increased by ΔT. For the experiments we used the value of 2 for both T_{min} and ΔT.

Hybrid AcE constructs the heuristic function in the backward direction and starts performing a forward directed search until it reaches a state S_B from where it is difficult to proceed towards the goals (*Switching criterion* is met). Then it changes direction, constructs the appropriate heuristic function and launches a backward search from the *Goals* towards S_B. If the backward search is also blocked after some steps in a state S_{B2}, the system will restart the planning process replacing the *Initial* state with S_B and the *Goals* with S_{B2}. The bi-directional search strategy of Hybrid AcE is outlined in Figure 3, where *St.plan* represents the plan from the *Initial* state to *St* for the progression module and the plan from the *Goals* to *St* for the regression one. The search algorithm is also exhibited through the flow chart of Figure 4.

The changes in the direction of the planning system aim to deal with the problems stated above and this can be understood with the following two arguments: (a) the change in the direction enables Hybrid AcE to update the heuristic function and thus make it more informative, (b) the adaptive way in which the system changes directions tends to solve the major part of the problem following the search direction, which best fits the specific problem.

5. HYBRID ACE'S HEURISTIC FUNCTIONS

Hybrid AcE uses two domain-independent heuristic functions based on the AcE planning system. The original heuristic function of AcE, is constructed, at a preplanning phase, in the forward direction and it can be utilized during search by a regression planner. The regression module of Hybrid AcE uses a variation of AcE's heuristic, which enables it to produce even more accurate estimates. For the progression module we implemented a reversed version of the same heuristic. Although the two heuristics are symmetric in the basic algorithm, there are some minor differences due to the fact that the goals of a problem do not form a complete state.

Both modules of the planning system adopt a weighted A* search strategy, where the total cost of a state S is calculated as: $w_1^* L(S) + w_2^* h(S)$. In this formula, $L(S)$ is the number of steps needed to achieve state S starting from the initial state, $h(S)$ is the value returned by the heuristic function and w_1 and w_2 are user-defined constants. For the implementation used for the purposes of this work w_1 was set to 3 and w_1 to 1.

5.1. The Regression Heuristic Function

The heuristic of the regression module works in two phases; the larger part of the calculations is performed once in a preplanning phase and only a few calculations are performed for each state in the search tree. This idea, which was introduced by the GRT planner,

```
Search Algorithm of BP
Input I, G, Output Plan
Set Plan1=Plan2=[], S=I, F=G, Direction=Forward, Global Close list=[] $T=T_{min}$
While $F \not\subset S$
Begin
    If Direction = Forward
    begin
        Create backward heuristic function $h_B$
        St=Progression_module(S,F,MAX_SOF_AGENDA,Threshold,$h_B$)
        If h(St) < h(S)
            Set Plan1=Plan1+St.plan, $T = T_{min}$, S=St
            Add {S,F} to the Global Close list
        Else
            $T = T+\Delta T$
        Direction = Backward
    end
    Else
    begin
        Create forward heuristic function $h_F$
        St=Regression_module(S,F,MAX_SOF_AGENDA,Threshold,$h_F$)
        If h(St) < h(F)
            Set Plan2=reversed(St.plan)+Plan2, $T = T_{min}$, F=St
            Add {S,F} to the Global Close list
        Else
            $T = T+\Delta T$
        Direction = Forward
    End
end
Return Plan1+Plan2
```

FIGURE 3. The algorithm of Hybrid AcE.

results in less overall time for the calculation of the state's estimations. However, especially in deep and complex search trees, the heuristic function may become obsolete after some time.

The first phase of the creation of the heuristic consists of assigning an integer value to each action in the problem. This number corresponds to an estimation of the distance between the action and the initial state of the problem. Roughly, the distance between an action *A* and a state *S* (noted as dist(*A*,*S*)) corresponds to the minimum number of actions that must be applied to *S* to reach a new state *S′* in which all the preconditions of *A* hold.

To calculate the distances for all the actions of the problem, the heuristic creates a layered graph, where each node corresponds to an action. The first level of the graph contains those actions that can be directly applied to the initial state. An action is included in the second level

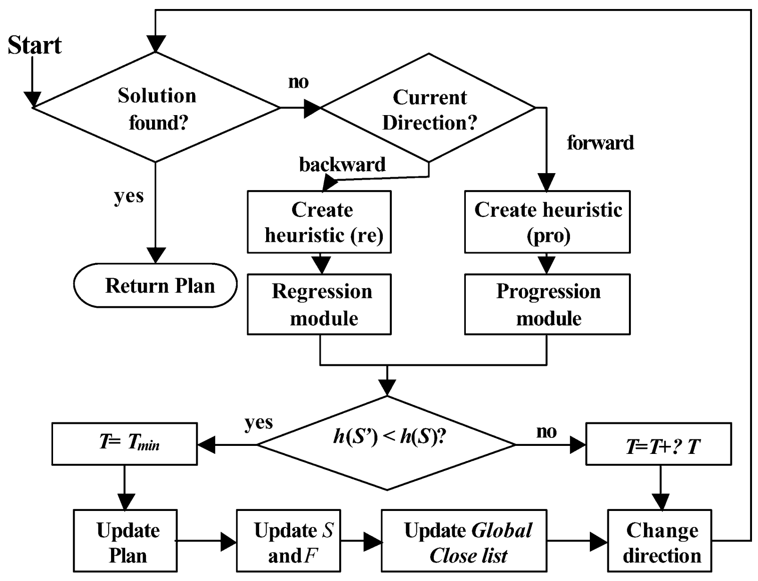

FIGURE 4. The flow chart of Hybrid AcE.

of the graph, if its preconditions are either in the initial state or can be achieved by actions in the first level. Similarly, an action is included in level L of the graph if its preconditions are either in the initial state or can be achieved by actions in levels 1 to $L - 1$. The graph is progressively built until all the actions of the problem have been included. Each time a new node is created in the graph the heuristic calculates the distance for the corresponding action, using the following formula:

$$\text{dist}(A) = \begin{cases} 1, & \text{if prec}(A) \subseteq I \\ 1 + \sum \text{dist}(X) : X \in CAS(\text{prec}(A)), & \text{otherwise} \end{cases} \tag{1}$$

where $CAS(S)$ (Cheaper Achieving Set) is a function returning the set of actions $\{A_p\}$ achieving all the facts of S with the minimum accumulated cost of dist(A_p). Note that CAS takes into account only actions that have already been included in the graph.

To find the minimum set of actions achieving a specific state, $CAS(S)$ has to calculate all the possible combinations of actions achieving S, and this process is combinatorial explosive. Therefore, $CAS(S)$ is approximated using a greedy algorithm, which is outlined in Figure 5.

The greedy algorithm used for $CAS(S)$ sequentially examines each fact in S trying to find the best of the actions achieving it. Although one of the criteria for grading candidate actions is the number of other facts in S that are coachieved with the examined fact, there are cases where CAS does not return the best set of actions. For instance, consider the following example:

$S = \{\text{p,q,r,f}\}$
dist $(A_1) = 3$, add $(A_1) = \{\text{p,q,r}\}$
dist $(A_2) = 2$, add $(A_2) = \{\text{p}\}$
dist $(A_3) = 1$, add $(A_3) = \{\text{q,r,f}\}$

```
Function CAS(S)
Input: a set of facts S
Output: a set of actions achieving S with near minimum accumulated dist
=====================================
Set $G = \emptyset$
$S = S - S \cap I$
Repeat
    f is the first fact in S
    Let act(f) be the set of actions achieving f
    for each action A in act(f) do
        $val(A)=dist(A) / |add(A) \cap S|$
    Let A' be an action in act(f) that minimizes val
    Set $G = G \cup A'$
    Set S = S - $add(A') \cap S$
Until $S = \emptyset$
Return G
```

FIGURE 5. *CAS* function.

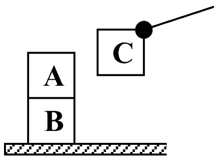

FIGURE 6. The initial state of example 1.

According to the algorithm in Figure 5, the best action achieving fact p is A_1, although dist(A_2) < dist(A_1) because A_1 achieves more facts of S than A_2. Therefore, $CAS(S) = \{A_1, A_3\}$ and the accumulated distance of $CAS(S)$ is $3 + 1 = 4$. However, the minimum set of actions achieving the facts of S, is $\{A_2, A_3\}$ with accumulated distance of 3. In most cases, however, CAS returns pretty good results in a small amount of time, as it can be seen by the following complexity analysis.

In the worst case, each action A$'$ selected by CAS will only achieve one fact of S and, therefore, the complexity of CAS will be $|S| * N$, where N is the number of the domain's actions. Because $|S| << N$ the order of the complexity is $O(N^2)$.

In the average case, the actions achieving a fact f are a small subset of the domain's actions and the size of the subset is N/K, where K is comparable to $|S|$ for each state S of the domain. Therefore, the complexity of CAS is $|S| * N/K$, which is in the order of $O(N)$.

We will illustrate the calculation of distances for actions with a concrete example of the Blocks-world domain (example 1). Suppose that the initial state of the problem is the one shown in Figure 6.

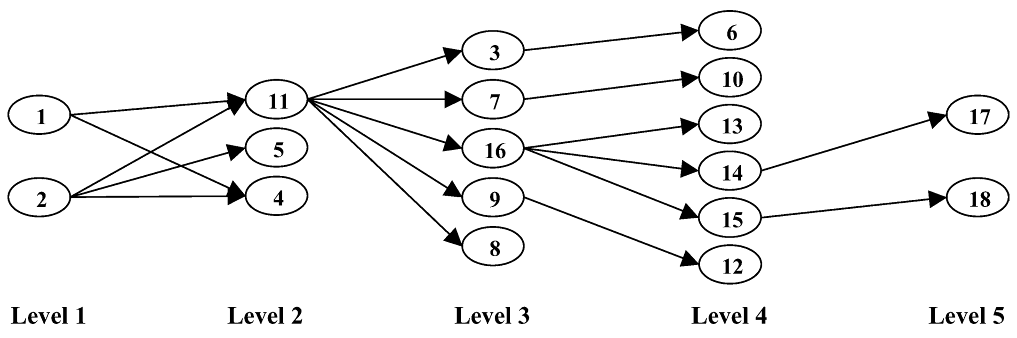

FIGURE 7. The layered graph for the example of Figure 6.

The actions in this problem are:

1. put-down (C)	7. put-down (A)	13. put-down (B)
2. stack (C,A)	8. stack (A,B)	14. stack (B,A)
3. stack (C,B)	9. stack (A,C)	15. stack (B,C)
4. pick-up (C)	10. pick-up (A)	16. pick-up (B)
5. un-stack (C,A)	11. un-stack (A,B)	17. un-stack (B,A)
6. un-stack (C,B)	12. un-stack (A,C)	18. un-stack (B,C)

The layered graph built by the heuristic function is presented in Figure 7. An arc from node X to node Y, is used to identify that action X achieves at least one of the preconditions of action Y. To maintain the readability of the graph we have omitted the arcs connecting nodes that are not in subsequent levels.

Because actions 1 and 2 can be applied to the initial state:

dist(1) = 1 and
dist(2) = 1
To calculate dist(11) we need CAS(prec(11)):
$S = \{$handempty,on(A,B),clear(A)$\}$
$S = S - S \cap I = \{$handempty$\}$
f = handempty
act(f) = $\{1{,}2\}$,
$A' = 1$ *both actions in act(f) have the same val, thus one of them is arbitrarily selected.*
$G = \{1\}$
$S = \emptyset$
CAS(prec(11)) = $\{1\}$ and
dist(11) = 1 + dist(1) = 1 + 1 = 2.

Similarly, the algorithm proceeds with the calculation of the distances for the rest of the actions.

The second part of the calculation of the heuristic function consists of utilizing the distances of the actions, during search, to evaluate each new state that is met. To calculate the heuristic value for a given state S_1, the algorithm uses $CAS(S_1)$ to find the near minimum set of actions achieving the facts of S_1 and then sums up the distances of the actions in $CAS(S_1)$. This can be seen as the process of evaluating an action A_{S1} for which: prec(A_{S1}) = S_1.

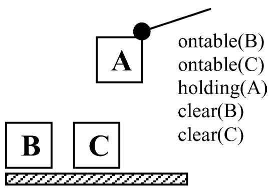

FIGURE 8. State S_a of example 1.

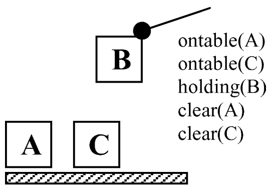

FIGURE 9. State S_b of example 1.

For example, to evaluate state S_a of Figure 8, we calculate CAS(S_a):

$S = \{ontable(B), ontable(C), holding(A), clear(B), clear(C)\}$
$S = S - S \cap I = \{ontable(C), holding(A), clear(B), clear(C)\}$
$f = ontable(C)$
$act(f) = \{1\}$
$A' = 1$
$G = \{1\}$
$S = \{holding(A), clear(B)\}$
$f = holding(A)$
$act(f) = \{10,11,12\}$
$A' = 11$
$G = \{1,11\}$
$S = \emptyset$
$CAS(S_a) = \{1,11\}$ and
$h(S_1) = \text{dist}(1) + \text{dist}(11) = 1 + 2 = 3$

Similarly, for state S_b in Figure 9 $h(S_b)$ is equal to dist(1) + dist(7) + dist(16) = 1 + 3 + 4 = 8

It is clear from the above examples that the heuristic still produces overestimates. To overcome this, we slightly modified the heuristic for Hybrid AcE. The general idea behind the modification, lays in the fact that when we select a set of actions to achieve the preconditions of an action A, we also achieve several other facts (denoted as *implied*(A)), which are not mutually exclusive with the preconditions of A. Supposing that this set of actions was chosen in the plan before A, then after the application of A, the facts in *implied*(A) would exist in the new state, along with the ones in the add-list of A. Taking all these into account, we produce a new list of facts for each action (named *enriched_add*), which is the union of the add-list and the implied list of this action. The heuristic function of Hybrid AcE uses the enriched instead of the traditional add-list in the CAS function but only in the second part that updates state S. Thus, the command *Set* $S = S - add(A') \cap S$ is altered to

Set $S = S - enriched\_add(A') \cap S$. According to experimental results, the modification in the heuristic results in smaller estimates, on average, and in better guidance of the search algorithm.

5.2. The Progression Heuristic Function

For the progression module we needed a heuristic able to estimate distances between any given state of the problem and the goals. Therefore, the heuristic should be created in the opposite direction of that of the heuristic for the regression module. Furthermore, the two heuristics should be of the same quality, as the primary objective of this work was to study the matter of search direction. Therefore, we created a reversed version of the heuristic of AcE, trying to overcome the problems imposed by the incompleteness of goals in a way that would not affect the estimating power of the heuristic.

The resulting heuristic works also in two phases; the major part of the calculations is performed once in a preplanning phase and the rest of the calculations are performed for each state encountered during search. In the preplanning phase the heuristic constructs a graph similar to the one by the regression heuristic with two main differences: (a) the nodes in the first level correspond to actions that can be directly applied to the goals instead of the initial state, and (b) the test for deciding which actions are applicable to a given set of facts (not necessarily a state) is more relaxed. Supposing that the goals of any given problem constituted a complete state description (S_G), then an action A would be applied to S_G iff all the facts of add(A) were contained in S_G. However, because this is not always the case and there are facts that are missing from the goals, all the actions would fail the criterion and it would not be able to complete the graph. Therefore, the heuristic of the progression module uses a simpler test (noted as the relaxed regressibility test).

Relaxed Regressibility Test

An action A is backwards applicable to level M of the graph if

$\exists$fact $f: f \in$ add(A), $f \in$ level M of the graph

The relaxed regressibility test is similar to the first criterion of the regressibility test of section 4.2, if we treat the levels of the graph as states. According to this test an action A can be applied to any level of the graph and, therefore, to the goals also if at least one of its add effects appear in that level. The main problem of the relaxation in the regressibility test is that it often results in shorter and denser graphs.

Consider for instance, the example of the Blocks world in Figure 10, where state S_G is encoded in two different ways. Encoding A forms a complete state description, while B does not. The two layered graphs for encoding A and B are presented in Figures 11 and 12, respectively. It is clear from the two graphs that if the goals formed complete state descriptions the graphs would be sparser and, therefore, more informative. However, in denser graphs the CAS function has more alternatives for selecting actions and usually returns sets with smaller accumulated distances that are generally preferred by planners. As it has been noticed by experimental results, there is a trade-off between these two facts and, therefore, the two encodings result in comparative performance.

During the creation of the graph the actions are graded using the following formula, which is basically formula (1) adapted to the opposite direction. Formula (2) uses *CRS* (Cheaper Requiring Set) function instead of *CAS*, which is outlined in Figure 13. *CRS*(S) differs from *CAS*(S) in that it returns a set of actions that have all the facts of S as preconditions instead of add effects.

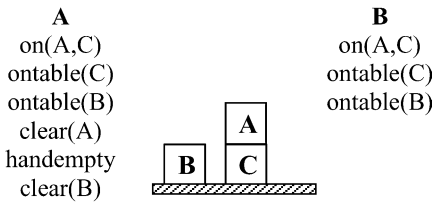

FIGURE 10. State S_G.

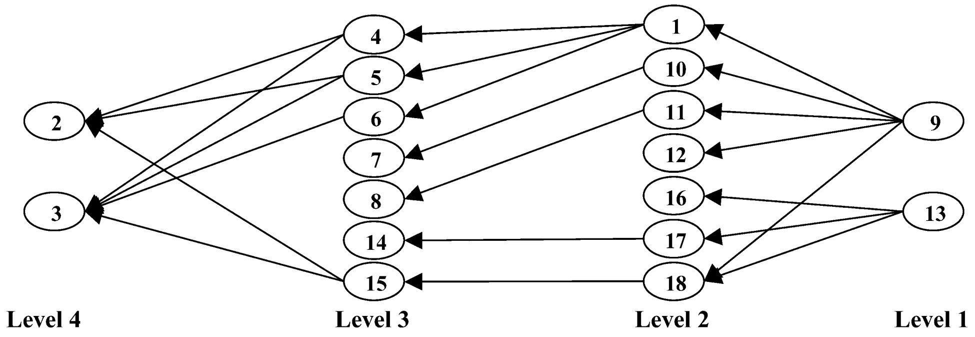

FIGURE 11. Layered graph for encoding A.

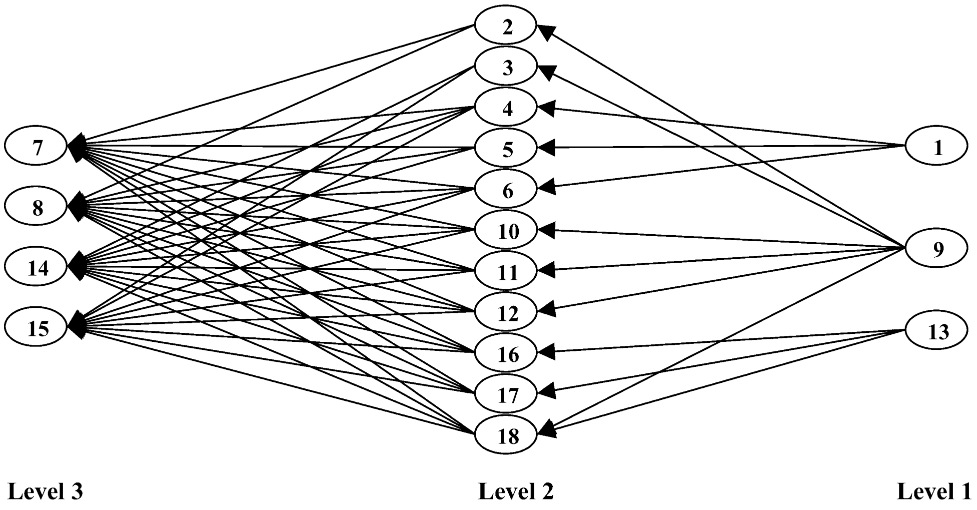

FIGURE 12. Layered graph for encoding B.

$$\text{dist}(A) = \begin{cases} 1, & \text{if add}(A) \cap G \equiv \emptyset \\ 1 + \sum \text{dist}(X) : X \in CRS(\text{add}(A)), & \text{otherwise} \end{cases} \tag{2}$$

During the planning phase, the heuristic uses CRS to estimate the distance between intermediate states and the goals. More specifically, to calculate $h(S_2)$ for a give state S_2, the heuristic calculates $CRS(S_2)$ and then sums up the distances of actions in the returned set.

```
Function CRS(S)
Input: a set of facts S
Output: a set of actions requiring S with near minimum accumulated dist
Set $G = \varnothing$
$S = S - S \cap G$
Remove from S the facts that are not preconditions of actions in graph
Repeat
    f is the first fact in S
    Let act(f) be the set of actions having f in their precondition lists
    for each action A in act(f) do
        $val(A) = dist(A) / |prec(A) \cap S|$
    Let A' be an action in act(f) that minimizes val
    Set $G = G \cup A'$
    Set $S = S - prec(A') \cap S$
Until $S = \varnothing$
Return G
```

FIGURE 13. *CRS* function.

6. ENHANCING THE HEURISTICS WITH FACT-ORDERING TECHNIQUES

Goal ordering for planning has been an active research topic over the last years and a number of techniques have been successfully adopted by state-of-the-art planning systems. The research so far has been focused on two tasks: (a) how to automatically extract as much information as possible about orderings among the goals of the problem, with minimum computational cost and (b) how to use this information during planning. McCluskey and Porteous with their work on PRECEDE (McCluskey and Porteous 1997) proposed a method for identifying goal orderings between pairs of atomic facts, based on direct domain analysis. The more recent work of Koehler and Hoffman on GAM (Koehler and Hoffmann 2000) have resulted in two techniques for identifying goal orderings, one based on domain analysis and another utilizing the information gained by the construction of a planning graph. The simplest and yet quite effective orderings extracted by these techniques have been described as *reasonable orders* and are based on the following idea:

"A pair of goals A and B can be ordered so that B is achieved before A if it isn't possible to reach a state in which A and B are both true, from a state in which A is true, without having to temporarily destroy A." (Porteous and Sebastia 2000).

IPP (Koehler and Hoffmann 2000) and FF (Hoffmann 2000) make use of reasonable orderings during planning through the construction of a goal agenda that divides the goals into an ordered set of subgoals. The planners sequentially try to achieve the first subgoal in the agenda, which has not yet been achieved. Experimental results have shown that the use of the goal agenda yields in significance improvement in terms of both planning time and plan quality.

Hybrid AcE adopts a slightly different method for identifying and utilizing possible orderings among facts of either the goals or the initial state of the problem. This method is based on mutual exclusions between facts of the domain. Because the planner calculates the set of binary mutual exclusions, to use them for the regression module, the overhead imposed by the calculation of orderings is negligible.

The technique of fact-ordering works in two phases: In the first phase, which is executed when the planner calls one of the two modules, the technique searches for orderings between pairs of facts of the goals or the initial state. The second phase is embedded in the two heuristic functions as a supplementary way for grading states. More specifically, after the evaluation of a state S_k, the technique searches in S_k for violations of orderings and for each violation the state is penalized, that is, the outcome of the heuristic function is increased by a constant number (100 in the current implementation). To understand the rationale behind this, consider the following example.

Goals $= \{a,b,c,d,e\}$
The goals can only be achieved in the following order: b,a,d,c,e
$\{a,e,c,d\} \subseteq S'$ and $b \notin S'$

By disregarding the orderings, state S' can mislead the heuristic function, because it seems like it needs to achieve only one fact to reach the goals. However, according to the orderings there is no way to achieve fact b as long as the other goals still hold. Therefore, to reach the goals starting from S' the planner needs to take extra actions to temporarily destroy facts a, e, c, and d, and reachieve them in the right order after goal b has been achieved. The penalty added to the estimated distance between S' and the goals can be seen as the need for these extra steps. The progression module uses an implementation of the above method for ordering the goals, while the regression module uses a variation for ordering the facts of the initial state. These two implementations are described in detail in the following sections.

6.1. Ordering the Goals

The progression module uses function *OB* (Ordered Before), which is outlined in Figure 14, to identify the possible orderings between pairs of goals. This step has to be repeated during subsequent calls of the search module if the regression module has altered the goals.

The orderings extracted by OB are used in the planning phase, to refine the results of the heuristic function. More specifically, after the evaluation of a state S, the heuristic function searches state S for possible violations of the goal orderings. Fact f of a state S is violating an ordering if:

$f \in Goals$ and $\exists$ fact g: $g \notin S$ and $\mathrm{OB}(g,f) = \text{true}$

For every ordering violation found in state S, the latter is penalized (i.e., the estimated distance between S and the Goals is increased by a constant number).

6.2. Ordering the Facts of the Initial State

The regression module uses function *OB-R* (Ordered Before for Regression), outlined in Figure 15, which is iteratively ran on every pair of facts in the initial state to identify the possible orderings.

In accordance to the goal ordering technique, after the evaluation of a state S, the regression heuristic function searches state S for possible violations of the orderings to penalize it. Fact f of a state S is violating an ordering if:

$f \in I$ and $\exists$ fact g: $g \notin S$ and $\mathrm{OB\text{-}R}(g,f) = \text{true}$

```
Function OB
Input: Goals a and b
Output: True (a should be ordered before b) or False (a should not be ordered
before b)
For each action O: a∈add(O)
begin
    Result =true
    For each fact f: f∈prec(O)
    begin
        If mx(b,f)=true
            Result=false
    end
    If result = true return false
end
Return true
```

FIGURE 14. OB function.

```
Function OB-R
Input: Initial facts a and b
Output: True (a should be ordered before b) or False (a should not be ordered
before b)
For each action O: a∈del(O)
    Result =true
    For each fact f: f∈ (prec(O)-del(O) ∪ add(O))
        If mx(b,f)=true
            Result=false
    If result = true return false
Return true
```

FIGURE 15. OB-R function.

7. PROBLEM SIMPLIFICATIONS

Both search modules of Hybrid AcE are equipped with techniques that analyze the domain to extract information that can be later used for speeding up the planning process and improve the quality of the plans. These techniques aim at simplifying the definition of the problem in a way that makes it easier for the planner to solve without jeopardizing completeness. The first technique, which works off-line before the planning process, identifies and removes useless facts from the definition of the initial state of the problem, while the second one is applied during search in every pair of $<I',G'>$ removing any achieved independent subgoal. Although the efficiency of the techniques varies among problem categories, they are domain independent, in a sense that they are based on the general representations of domains and problems. The computational overhead imposed by the techniques is negligible compared to the total planning time, even in cases where they do not manage to improve the system.

7.1. Simplifying the Initial State

This simplification technique concerns facts of the *Initial* state that are useless for the planning process. For example, in the *logistics* domain the *Initial* state may contain facts noting the initial location of packages, which do not need to be moved. This information is present in the description of the world, for means of completeness, but increases the branching factor of the problem with useless actions.

Hybrid AcE employs a simple but efficient method for eliminating useless facts from the *Initial* state, which is based on the backward graph built by the progression heuristic function. After the initial construction of the backward graph, the method eliminates all the facts from the *Initial* state that are not referenced by any action in the graph. If a fact f is not referenced by any action of the graph, there is no way to reach a state S where $f \in S$, by regressing the *Goals*. This means that f does not contribute at all in the process of reaching the *Goals* of the problem. Thus, it is safe to remove it from the *Initial* state without jeopardizing completeness.

There is no actual overhead imposed by the above method, because the backward graph is built by the progression heuristic function no matter if the method for eliminating useless facts is applied or not. The method is not complete, in the sense that it does not identify all the forms of information that could be safely removed from the definition of the problem.

7.2. Ignoring Achieved Facts

The technique for ignoring achieved facts concerns cases where the state being expanded by the search algorithm contains facts that also belong to the goals. This is a very frequent case in search, because in complex problems the goals are achieved in a step-by-step base during the search. In other words, as the search algorithm moves towards the solution, it encounters states that contain more and more goals until it reaches a state containing all the goals of the problem. By removing these facts from both the goals of the problem and the state under expansion the benefits are twofold: (a) the branching factor is cut down, because all the actions referring to the removed facts will not be considered and this speeds up searching and (b) the heuristic function focuses on the remaining goals guiding the search towards them and, therefore, shorter plans are found.

Consider for instance, the example of Figure 16, where the problem is to move three balls among three connected rooms. The three first levels of the search tree for this example is shown in Figure 17, where only two nodes are expanded in level 3, due to space limitations. The total number of nodes in the first three levels of the search tree is 33; 26 of which belong to the third level, and the average branching factor for nodes in level 1 and 2 is 5.1.

The corresponding search tree for the same problem, when achieved goals are ignored, is presented in Figure 18. The tree consists only of 11 nodes for the first three levels and the average branching factor is 2.8. It is clear from this example that the benefits for applying a method for removing achieved goals can be really significant, especially in cases where the goals consists of a large number of independent goals.

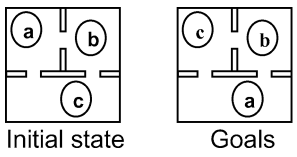

FIGURE 16. An example of the balls problem.

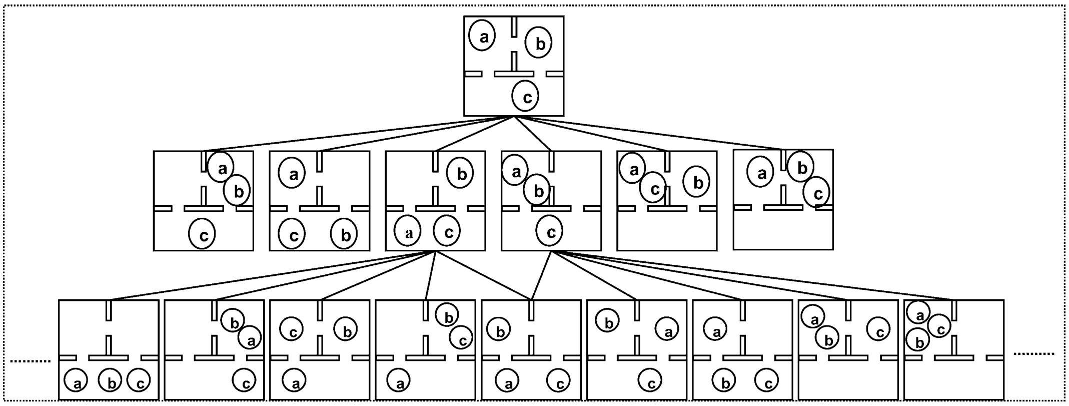

FIGURE 17. Search tree for Figure 16.

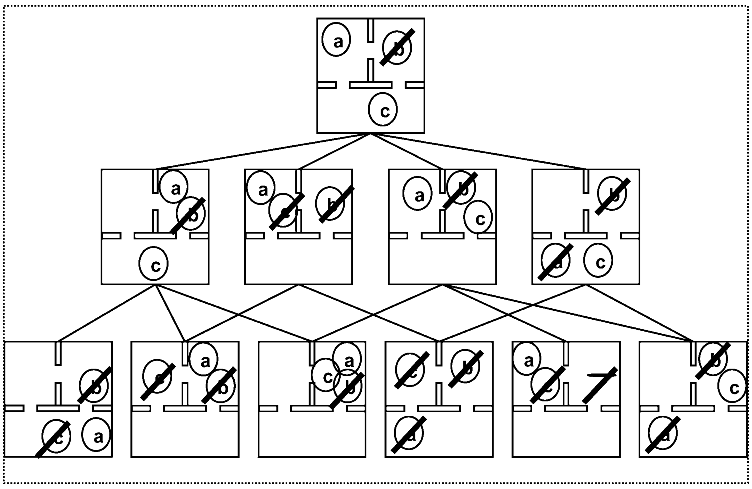

FIGURE 18. Search tree for Figure 16 ignoring achieved goals.

Although methods for removing achieved goals can significantly speed up the planning process and help in producing better plans, they must take into account all the possible interactions among the facts of a state to preserve completeness. It is a common situation in planning, to have specific objects that serve as mediators for changing the properties of other objects. For example, in logistics-based domains there are objects such as trucks, airplanes or drivers that are necessary to move the packages. Because there are objects that depend on others, the simplification methods should only remove achieved goals that do not contribute, in any way, to the achievement of the rest of the goals.

The proposed technique works in two phases to identify and safely remove achieved goals from the definition of the problem.

In the first phase, which is executed once before the actual planning phase, the technique performs a limited search in the space of the facts to identify dependencies between the objects of the problem.

Definition 1. Action set Ω covers Object O_1 if all the facts that refer to O_1 can be achieved by subsequently apply actions from Ω starting from any given state.

Definition 2. Object O_1 is independent if it can be covered by the set of actions that refer to O_1.

Definition 3. Object O_2 depends on object O_1 if
O_2 is not independent
$\ni Z$: $O_1 \in Z$, Z covers O_2, $Z - \{O_1\}$ does not cover O_2.

During this search the algorithm creates a leveled graph of the objects, where the level of each object O (noted as $ob\_lev(O)$) corresponds to the degree of independency of that object. More specifically, for object O:

$ob\_lev\ (O) = 0$ if O is independent
$ob\_lev\ (O) = 1$ if O can be covered by actions referring to O and to objects of level 0
...
$ob\_lev\ (O) = k$ if O can be covered by actions referring to O and to objects of levels 0 to $k - 1$.

In the second phase, which is executed online during the search, the technique removes from the definition of the problem the achieved goals that refer to objects for which there are no pending dependencies.

Definition 4. There is a pending dependency between objects O_1 and O_2, if:

1. The goals that refer to O_2 have not yet been achieved,
2. O_2 depends on O_1.

More specifically, given a random state S and the goals of the problem G, a fact p ($p \in S$ and $p \in G$) is removed from both S and G if:

$level(p) \geq level(q_i))$ *for each* $q_i \in S$.

In the previous expression level(f) corresponds to average($ob\_lev(O_i)$) for the objects that are referenced by fact f. This is necessary because, a fact may refer to more than one object and, therefore, to decide if a fact depends on another all the referenced objects must be taken into account. Note here that although the algorithm does not take into account direct dependencies between objects, it only decides to remove a fact f if the objects referenced by f belong to the last levels of the graph and, therefore, no other objects depend on them.

8. EXPERIMENTAL RESULTS

To test the efficiency of the bi-directional planning system, we additionally implemented two single-direction planners embodying the two search modules of Hybrid AcE. The two additional planners, noted as AcE-P and AcE-R implementing the progression and regression module, respectively, were equipped with the same techniques as Hybrid AcE and the same setup for the planning parameters was made for all three planners. For the tests, we selected a set of 270 problems from 9 domains adopted from the three AIPS planning competitions.

The three planners were implemented in C++ and compiled with the GNU C++ compiler. The platform used for the tests was a SUN Enterprise 450 Server with four processors at 400 MHz and 2 GB of shared memory running Solaris 8 operating system. For each experiment, we recorded the CPU time needed to solve the problem and the number of steps in the resulting plan. The maximum CPU time for each run was limited to 120 seconds after which the planner would stop and report to the problem as being unsolvable.

TABLE 2. Experimental Results

	Problems			Steps			Time		
	AcE-P	AcE-R	H-AcE	AcE-P	AcE-R	H-AcE	AcE-P	AcE-R	H-AcE
Assembly	24	28	29	24.90	28.50	25.10	2.08	0.42	1.15
Blocks-3	25	24	27	16.50	12.90	13.80	3.77	2.69	1.90
Blocks-4	28	28	28	38.00	37.60	36.30	0.55	0.65	0.85
Driver	22	30	30	18.80	27.00	20.20	1.30	0.59	0.89
Gripper	25	22	25	52.50	67.20	51.40	0.45	1.60	0.97
Hanoi	17	25	26	67.50	66.10	64.80	2.67	4.13	1.45
Logistics	30	28	30	58.30	58.30	58.50	2.17	0.60	0.78
Miconic	23	26	25	17.50	32.50	16.80	3.83	0.67	0.93
Zeno	29	21	26	17.60	23.30	17.70	2.54	2.78	1.89
Overall	223	232	246	34.12	39.09	33.85	2.12	1.48	1.19

Table 1gives a concentrated view of the experimental results obtained by running each one of the three planning systems on the 270 problems in the set. For each domain Table 1 presents the number of problems solved by each planner, the average number of steps in the resulting plans and the average number of seconds taken by the planner to solve the problems. Note here that the number of problems is a crucial parameter because, especially in hard problems, there was a significant number of failures in finding a solution. For example, in the Hanoi domain Ace-P managed to solve only 17 problems out of 30. The last line of Table 1 presents the total number of problems solved by each planner and the means for plan length and planning time over the whole problem set.

It can be seen from the data in Table 1 that between the two single-direction planners, AcE-R is more stable, because it solved approximately 4% more problems in significantly less time (30% on average) than AcE-P. As far as plan length is concerned though, AcE-P performed much better, because on average it found shorter plans than AcE-R by approximately five steps. However, it can also be seen from Table 1 that these general conclusions do not hold when the comparison includes specific domains. For example, in the Zeno domain AcE-P was better than AcE-R in all three criteria, while in the Blocks-3 domain AcE-R found shorter plans and managed to solve less problems than AcE-P.

Concerning Hybrid AcE, it is clear from the results that the bi-directional search strategy enables it to tackle successfully more difficult problems. Hybrid AcE managed to solve more problems than both AcE-P and AcE-R in every domain except for Miconic and Zeno. In overall H-AcE solved 23 problems more than AcE-P and 14 problems more than AcE-R. Concerning plan length, H-AcE found shorter plans on average in four domains and quite good plans in the rest domains. Thus, in overall, H-AcE performed slightly better than AcE-P (0.27 steps) and much better than AcE-R (5.24 steps). Finally, concerning planning time, H-AcE clearly outperformed AcE-P and AcE-R, solving each problem quite fast. H-AcE needed on average almost half the time needed by AcE-P and approximately 20% less time than AcE-R.

9. CONCLUSIONS AND FUTURE WORK

It is a generally accepted fact that there are certain domains or certain problems of domains that can be tackled more efficiently by forward planners and others that can be tackled more efficiently by backward ones. The matter of direction in planning is an active field of research and yet no clear answer has been given to the question of which direction should be generally preferred. This paper described Hybrid AcE, a bi-directional planning system that switches between forward and backward search, based on the outcome of a complex criterion on the efficiency of the heuristic function and the morphology of the problem. Hybrid AcE uses two domain-independent heuristic functions that extend the one of the AcE planning system enhanced with a powerful fact-ordering technique and two additional techniques for problem simplification.

The bi-directional system has been tested on a large set of planning problems from various domains and compared to two single-direction planners utilizing the same heuristics and optimization techniques. Experimental results show that Hybrid AcE solves the major part of each problem in the direction that best fits it. Therefore, it manages to have a significantly more stable performance, scaling up to harder problems than the single-direction planners, returning shorter plans in less time. Although there is still room for improvement, because Hybrid AcE is occasionally outperformed by one of the single-direction planners, the results show that the hybrid search strategy is a very stable option, which in overall bests both forward and backward search.

In future, we plan to extend the planning system with an even more sophisticated criterion for changing direction, which is the crucial part of the search algorithm. For this reason we plan to employ Machine Learning techniques to automatically extract knowledge that associates the characteristics of the problem in hand and the current state of the search with the next step taken by the algorithm (proceed with search or change direction). Moreover, we plan to enrich the system with improved heuristic functions and optimization techniques that will enable it to handle even more complex problems.

REFERENCES

BLUM, L., and M. FURST. 1995. Fast planning through planning graph analysis. *In* Proceedings, 14th International Joint Conference on Artificial Intelligence, Montreal, Canada, pp. 636–642.

BONET, B., and H. GEFFNER. 1999. Planning as heuristic search: New results. *In* Proceedings, ECP-99, Durham, UK.

BONET, B., and H. GEFFNER. 2001. Planning as heuristic search. Artificial Intelligence, Special issue on Heuristic Search. **129**(1–2):5–33.

BONET, B., G. LOERINCS, and H. GEFFNER. 1997. A robust and fast action selection mechanism for planning. *In* Proceedings, 14th International Conference of the American Association of Artificial Intelligence (AAAI-97), Providence, Rhode Island, pp. 714–719.

FIKES, R., and N. NILSSON. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, **2**:189–208.

FINK, E., and J. BLYTHE. 1998. A complete bidirectional planner. *In* Proceedings, Fourth International Conference on AI Planning Systems. Pittsburgh, Pennsylvania, pp. 78–85.

HOFFMANN, J. 2000. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. *In* Proceedings, 12th Int. Symposium on Methodologies for intelligent Systems. Charlotte, USA, pp. 216–227.

KOEHLER, J., and J. HOFFMANN. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. JAIR, (12), Vol 12, pp. 338–386.

KORF, R. 1998. Artificial intelligence search algorithms. *In* CRC Handbook of Algorithms and Theory of Computation. *Edited by* M. J. Atallah. CRC Press, Boca Raton, FL, pp. 36–1 to 36–20.

LONG, D., and M. FOX. 1998. Efficient implementation of the plan graph in STAN. JAIR, **10**:87–115.

MASSEY, B. 1999. Directions in planning: Understanding the flow of time in planning. Available as a Technical Report from the University of Oregon.

MCCLUSKEY, T., and J. PORTEOUS. 1997. Engineering and compiling planning domain models to promote validity and efficiency. Artificial Intelligence, **95**:1–65.

MCDERMOTT, D. 1996. A heuristic estimator for means-end analysis in planning. *In* Proceedings, AIPS-96. Edinburgh, UK, pp. 142–149.

NGUYEN, X., S. KAMBHAMPATI, and R. NIGENDA. 2000. AltAlt: Combining the advantages of graphplan and heuristics state search. *In* Proceedings, 2000 International Conference on Knowledge-based Computer Systems, Bombay, India.

PORTEOUS, J., and L. SEBASTIA. 2000. Extracting and ordering landmarks for planning. *In* Proceedings, 18th Workshop of the UK Planning and Scheduling SIG. Milton Keynes, UK.

REFANIDIS, I., and I. VLAHAVAS. 1999. *GRT*: A domain independent heuristic for STRIPS worlds based on greedy regression tables. *In* Proceedings, Fifth European Conference on Planning, Durham, UK, pp. 346–358.

STONE, P., M. VELOSO, and J. BLYTHE. 1994. The need for different domain-independent heuristics. *In* Proceedings, AIPS-94, Chicago, USA.

VELOSO, M. 1994. Planning and learning by analogical reasoning. Springer-Verlag. New York, NY.

VELOSO, M., J. CARBONELL, A. PEREZ, D. BORRAJO, E. FINK, and J. BLYTHE. 1995. Integrating planning and learning: The PRODIGY architecture. Journal of Experimental and Theoretical Artificial Intelligence, Vol. 7(1), pp. 25–52.

VELOSO, M., and P. STONE. 1995. FLECS: Planning with a flexible commitment strategy. JAIR (3), Vol. 7(1), pp. 81–120.

VRAKAS, D., and I. VLAHAVAS. 2001. Combining progression and regression in state-space heuristic planning. *In* Pre-Proceedings of the Sixth European Conference on Planning, Toledo, Spain, pp. 1–12.

VRAKAS, D., and I. VLAHAVAS. 2002. A heuristic for planning based on action evaluation. *In* Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, Applications. Varna, Bulgaria, pp. 61–70.