22

Extending AI Planning to Solve more Realistic Problems

Joseph Zalaket
Holy Spirit University of Kaslik
Lebanon

1. Introduction

Applying AI planning to solve real-world problems is still difficult despite many attempts done in this area. Classical planning systems are able to handle a limited number of symbolic data elements, without taking into account the numerical aspect of many real-world problems. Some recent planners have moved to solve more realistic problems involving resource consumptions and time management (Bresina et al., 2002; Bacchus & Ady, 2001; Edelkamp, 2002), but the most of these planners deal with the numerical side of the planning problem as an assisting feature to a main symbolic problem, without being able to tightly mix the two sides of the problem to be solved as one homogeneous problem. However, there are still many real-world problems that involve dominant even absolute numerical processing (Zalaket & Camilleri, 2004b; Hoffmann et al., 2007) and for which planning problem representation and data type handling are to be extended. In order to cover this latter type of problems, we propose many extensions that allow the application of the planning process evenly over symbolic and numerical data that can constitute any realistic planning problem.

Despite the multiple extensions that have been made to the Planning Domain Definition Language (PDDL) (Ghallab et al., 1998; Fox & Long, 2002; Gerevini & Long, 2005), this language is still insufficient for representing real-world problems that need complex action representation and complex state transformation expression. This lack motivates the organizers of the sixth international planning competition (IPC-6) to request a new extension to PDDL (the PDDL3.1 version). Inspiring from the continuous extensions to PDDL, we propose our first extension that concerns the data representation, in which we introduce the concept of using non-invertible functions to update the numerical and non-numerical data throughout a planning process. This type of functions allows the integration and the handling in an easy way of uncertainty as well as of temporal and numerical knowledge into planning.

As non-invertible functions can be only applied in forward traversal in a search space, hence we focus on the adaptation of forward planning systems to support the application of this kind of functions. We show that our technique can be used by any forward planner with a minor expansion, and we detail the extension of the Graphplan (Blum & Furst, 1995) structure and algorithm to support the execution of functions. The advantage of the

Graphplan compared to other forward planning algorithms is that: Graphplan is constructed in progression, thus it can support the application of our proposed functions like any other forward planning system. Furthermore, the compact structure of Graphplan and its capability of delaying the search of the plan (the sequence of operators) to the end of the graph construction process give it the ability of dealing with the effects of our proposed non-invertible functions during the regression search phase.
As we illustrate in this chapter, it is enough to simply save the effects of functions during the construction phase of the Graphplan to be able to extract the plan later in backward search. Saving these effects allows also the application of black-box functions with no side effects over state variables irrespective of their content. This can lead to the integration of control structures like conditional and iteration structures into functions in order to perform more complex computation.
As additional extensions: we present the relaxation of the numeric tasks of planning by ignoring the effects of actions that move away the values of numeric variables from their goal values.
We present the calculation of a heuristic function, which can be uniformly derived for numerical and symbolic facts from a relaxed planning graph-like structure.
We introduce the representation of numerical facts as multi-valued attributes in the relaxed graph to ease the search of a relaxed plan. In this way, the effects of the applied non-invertible functions will be transparent during the search for a relaxed plan.
Finally, we present some empirical results that show the effectiveness of our extensions to solve more realistic planning problems.

2. Apply functions in planning

Using functions in planning has been studied in Functional Strips (Geffner, 1999) and FSTRIPS (Schmid et al., 2002). Functional Strips has argued that the generated literals can be reduced by replacing relations by functions. We are still supporting this idea in our extension, but our main interest in functions is their ability to handle complex numerical and conditional effects, as well as to express complex preconditions and goals. For example, to plan the motion of a robot that is expected to travel from an initial position to a target position in the presence of physical obstacles in its environment. To avoid collisions between the robot and the obstacles, we have to use trigonometric functions that allow the robot to follow a circular path in its environment (Samson & Micaelli, 1993; BAK et al., 2000). Trigonometric expressions are simple to be solved by classical programming languages, but they can not be expressed in classical AI planning languages like PDDL3.0. For this reason we introduce the extension of AI planning to allow the application of external functions that can be written in any programming language. Inspiring from the representation of functions in object-oriented database, we add to the problem definition written in PDDL3.0 (Gerevini & Long, 2005) the declaration of external functions by specifying their execution path. In addition to their capabilities of solving numerical problems like the circular movement of a robot external function are able to handle all kind of conditional and probabilistic effects.
Non-invertible function can be only applied during a forward traversal in the search space, as at each state we can apply functions that generate the next state in that space. This process can continue by testing at each new state the satisfaction of the goal conditions, until

reaching a state that satisfies the goal or until no more memory space is available to continue the search. In this way, a function can act as a black-box irrespective of its content, but domain independent forward planning systems are not able to solve large planning problems because of the high space and time complexities. So, heuristic functions should be used to guide the search in forward chaining in order to allow the application of functions and the resolution of large planning problems. As Graphplan is constructed in forward chaining traversal, thus it can support the application of this kind of functions. It will be also possible to extract the plan in backward search from the Graphplan structure by navigating through the effects of the functions irrespective of their content.
To avoid all kind of side effects during the construction of the planning graph we restrict all the formal parameters of the functions to be constant, which means, a function can only return a value without updating any state variable. This returned value can then be affected to one state variable at a time.
Many real-world problems have complex characteristics, such that durative actions, temporal or uncertain conditions, resource consumption, numeric functions, etc. Most of these characteristics can be modeled with the PDDL3.0 language which is the mostly used by recent planners. PDDL3.0 gives the possibility of expressing optimization criterion such that maximization or minimization of resource consumption. Many planners have been extended from propositional versions to deal with optimization problems like Metric-FF (Hoffmann, 2002) and SAPA (Do & Kambhampati, 2001). The most of these extensions are paying attention to one or the other of the real world complex characteristics, each time with a special computation procedure which is added to a classical propositional planner.
We propose an extension to the Graphplan (Blum & Furst, 1995) that allows its structure to handle all kind of non-invertible functions application. In the next section, we start by presenting the extension of the planning language to allow the integration of external functions that we will integrate later to the Graphplan.

3. Language extension

The language that we propose to define a planning problem is an extension to PDDL3.0 language in which we introduce the integration of external functions to update numeric values in addition to the arithmetic expressions allowed in PDDL3.0. This extension allows the use of mathematical functions like COS, SIN, SQRT, EXP, ROUND,. . . and user defined functions instead of simple arithmetic expressions supported by PDDL3.0 in which only classical arithmetical operators (+, -, /, *) are allowed. The control flow (conditional statements and loops) can be used within an external function to hold up complex numerical computation and thus complex conditional effects of actions can be expressed and handled within the core of external functions. Note that, external functions can also be used to update non numeric variables such as propositional facts. But, we restrict in this chapter the use of external functions for updating numeric variables which is more beneficial for the planning process.
We also introduce the separation of the constraints from the precondition in the definition of actions, in a way that the constraints will be added to a new separate list that should be tested before the instantiation of the action. Hence, an action will be instantiated if and only if its constraint list is satisfied by the current state which can reduce the number of ground

actions, and consequently useless tests that should be done for precondition satisfaction will be avoided and the memory space used to store the ground actions will be reduced.

3.1 State space nature and transition

Each distinct instance of the world is called a state, denoted by s. The set of all possible states is called a state space S.

S is formed by two disjoint sets: a set of logical propositions P and a set of numeric variables N.

We denote by P(s) the subset of logical propositions of a state s and by N(s) the subset of its numeric variables.

The transformation of the world from a state $s \in S$ into another state $s' \in S$ is done through the application of a set of actions A, such that $s' = t(s, A)$ where t is a state transition function. A state transition function t transforms s by 3 ways:

- Adds logical propositions to P(s)
- Removes logical propositions from P(s) (when propositions become false)
- Assigns new values to existing numeric variables in N(s)

3.2 Action definition

An action *a* is defined as a tuple (args, con, pre, eff), where:

- args is the list of arguments made by variables which represent constant symbols in S and/or numeric variables in N.
- con is the list of constraints. The constraints are tested before the instantiation of the actions to avoid instantiating actions with incoherent arguments. Constraints follow the same definition format as preconditions.
- $pre = pre_P \cup pre_N$ is the list of preconditions.
    - pre_P defined over P are propositional preconditions
    - pre_N are numerical preconditions, such that: $\forall c \in pre_N$, $c = (n\ \theta\ g)$, where $n \in N$, $\theta \in \{<, \leq, =, >, \geq\}$ and g is an external function or an expression.

Definition-1: *An expression is an arithmetic expression over N and the rational numbers, using the operators +, -, * and /.*

Definition-2: *An external function f is a constant function written in a high level programming language on the form of: type* $f(n_1, n_2, , n_m)$*, where arguments* $n_1, n_2, , n_m \in N$.

The function and all its arguments (if exist) are declared as constants in a way that the function calculates and returns a value without affecting any numeric state variable. By constant function we mean that the function is not allowed to internally modify the state variables.

- $eff = eff_P \cup eff_N$ is the list of effects.
    - $eff_P = eff_{P^+} \cup eff_{P^-}$: defined over P are positive and negative propositional effects that add or remove literals.
    - eff_N are numeric effects, such that: $\forall e \in eff_N$, $e = (n := g)$, where $n \in N$ and g is an external function or an expression.

Figure-1 illustrates the definition of the water jugs domain using the above extensions to the planning domain definition language.

Example-1: *The water jugs domain definition using condition list and external functions.*

```
(define (domain Jugs)
(:requirements :typing :fluents :external)
(:types jug int)
(:functions ((capacity ?j - jug) – int)
((fill ?j - jug) – int))
(:external java (and ( (path (int max (int, int, int)) (c:/javaplan/WaterJug.max))
(path (int min (int, int, int)) (c:/javaplan/WaterJug.min))))

(:action poor
        :parameters
                (? j1 ? j2 – jug)
        :constraints
        (not(= ?j1 ?j2))
        :precondition
                (and (<(fill ?j) (capacity ?j)) (> (fill ?j) 0))
        :effect
                (and (assign (fill ?j1) (max ((fill ?j1), (fill ?j2), (capacity ?j2)))
                (assign (fill ?j2) (min ((fill ?j1), (fill ?j2), (capacity ?j2))))


External functions written in Java at c:\javaplan\.;
 public class WaterJug{
        public static final int max (final int v1, final int v2, final int c2){
                if (v1+v2 > c2) return v1+v2-c2;
        return 0;
        }

        public static final int min (final int v1, final int v2, final int c2){
        if (v1+v2 < c2) return v1+v2;
        return c2;
        }
}
```

Fig. 1. The water jugs domain definition

Remark: An external function can be written in any host programming language in this example we show functions written in java language, but these same functions can be written in a different programming language like the C++ for example. A special parameter should be set to allow the planner to know which interpreter should call to execute the functions as it is the case for functions written in Java or if the functions are directly executable as for functions written in C or in C++.

3.3 Planning problem definition

A planning problem is defined as P=<S, A, s_I, G>:

1. A nonempty state space S, which is a finite or countable infinite set of states.
2. For each state s ∈ S, a finite set of applicable actions A(s). A state transition function *t* produces a state s′ = *t*(s, A) ∈ S, for every s ∈ S and applicable actions A(s).
3. An initial state s_I ∈ S, where s_I is made by initially true logical propositions and initial values of numeric variables.
4. A set of goal conditions G that should be satisfied at a state s_G ∈ S. The set of goal conditions G = $G_P \cup G_N$, where G_P defined over P are propositional goal conditions and G_N are numeric goal conditions that should be satisfied at a goal state s_G.

   Note that: ∀ $l \in G_N \Rightarrow l = (n\ \theta\ c)$, where $n \in N$, $\theta \in \{<, \leq, =, >, \geq\}$ and *c* is a constant numeric value, an external function or an arithmetic expression.

4. Graphplan extension

Graphplan (Blum & Furst, 1995) was the subject of many extensions (Smith and Weld, 1999; Do & Kambhampati, 2000; Cayrol et al., 2000) in which researchers have tried to adapt the Graphplan to solve more expressive planning problems than those that can be expressed in pure propositional STRIPS language (Fikes & Nilsson, 1971). The adaptation that we propose to the Graphplan allows its structure to handle the execution of external functions. This type of functions allows Graphplan to handle uncertainty and to integrate temporal and numerical knowledge. Before introducing our extension the the Graphplan and to its planning graph structure we briefly overview the Graphplan algorithm.

4.1 Graphplan overview

Graphplan solves problems represented in STRIPS language. An action in STRIPS has a name and a parameter list and it is specified in terms of preconditions and effects. Preconditions are conjunctions of positive literals and effects are conjunctions of positive and negative literals that are respectively stored into two separated lists the ADD list and the DEL list.

Graphplan alternates between two phases: graph construction and solution extraction. In the graph construction phase Graphplan expands a planning graph in forward chaining until either a solution is found or it is sure that no solution exists. Then, in solution extraction phase Graphplan searches for a solution into the planning graph in backward chaining, if no solution is found then Graphplan will continue by expanding one more level until a solution is found or until a sufficient condition indicates that there is no solution to the problem.

A planning graph consists of a sequence of levels, where $level_0$ corresponds to the initial state. Each level contains a set of literals and a set of actions. The actions of a level are those that have their preconditions reachable (satisfied and mutually consistent) by the literals of the same level. The literals of a next $level_{i+1}$ are the effects of the applicable actions of previous $level_i$. The literals that satisfy the preconditions of action instances at the same level are connected with these actions via direct edges. Also, the action instances are connected via direct edges with their literals effects at the next level. ADD edges connect the action to

its next level added literals and DEL edges relate the action to its next level deleted literals. To maintain the existence of literals from one level to its successor level Graphplan uses the "no-op" actions. A "no-op" action is an action that "does nothing", its only role is to allow every literal that appears at $level_i$ to also reappear at $level_{i+1}$.
To speed up its solution extraction Graphplan uses binary mutex relations for both actions and literals during graph expansion.
A mutex relation holds between two actions at a given level if there is:

- *Inconsistent effects:* one action negates an effect of the other.
- *Interference:* one literal effect of one action is the negation of a precondition of the other.
- *Competing needs:* a literal precondition of one action is mutually exclusive with a literal precondition of the other.

Two literals are mutually exclusive if one is the negation of the other or if each possible pair of actions that could achieve the two literals has a mutex relation.
Graphplan expands the planning graph up to a level in which all the literals of the goal are present without mutex relation between any pair of them. Then, it starts the solution extraction phase in a way that for each sub-goal at the last $level_n$, it tries to find an action instance at $level_{n-1}$ that has this sub-goal as an add effect and that is not mutually exclusive with another action instance that is already selected. The preconditions of the selected action instances become the new set of sub-goals at $level_{n-1}$ and so on until reaching the initial state ($level_0$). If Graphplan fails to find an action that can achieve a sub-goal, it backtracks and tries to find another path within the planning graph by trying other action instances. Graphplan uses the 'memoization' concept which allows it memorizing its trace to ease the backtracking process. Therefore, if Graphplan fails to find a set of consistent action instances at a certain level and backtracking becomes useless, then Graphplan expands an additional planning graph level and restarts its search until finding a solution or until reaching a saturated graph that can not be expanded any more.

4.2 Graphplan adaptation for integrating external functions

In algorithm-1 we present the adaptation of the Graphplan algorithm to support the handling of numeric variables and the execution of external functions. In our implementation the adapted Graphplan consists of 2 types of levels fact levels and action levels. Compared to the original Graphplan implementation a fact level is a combination of propositional and numeric facts, instead of being only propositional. An action level supports the execution of external functions and the evaluation of arithmetic expressions in addition to its capability of adding negative and positive propositional effects. Each numeric variable that belong to a fact level is represented as a multi-valued attribute in the planning graph structure. When an action updates a numeric variable using an external function or an arithmetic expression, we add this updated value as a new value to the multi-valued concerned attributed. In this way, we consider that the old value is deleted and a new value is added to the multi-valued attribute. This approach allows us to maintain the addition and deletion edges of actions for numeric values. An action that updates a numeric variable is related by an ADD edge to its new returned value and by a DEL edge to its previous value in the multi-valued attribute.
Numeric domains can be infinite and thus, instantiating actions for all possible values become impossible. For this reason, we propose an incremental instantiation of actions for numeric values. To allow incremental instantiation, we introduce the concepts of implicit

parameters and implicit conditions in order to maintain the precondition edges of the actions that relate them to numeric facts.

Definition-3: *A numeric variable that belongs to arguments of an external function is an implicit precondition action parameter.*

Definition-4: *A numeric variable affected by an update function application or assigned to an arithmetic expression is an implicit precondition action parameter.*

Definition-5: *An implicit precondition action parameter is related by a precondition edge to its action.*

This incremental instantiation is done by replacing the numeric variables by their corresponding values from the current fact level, which avoids the flood of numeric values and can be used for discrete and continuous numeric values. Only the variables that are parameters of external functions and the right hand side numeric variables of expressions are instantiated from the current fact level. All these incrementally instantiated variables are added implicitly to the action preconditions by direct edges. These edges to implicit preconditions will allow us to follow the trace of the functions application within the planning graph during the extraction process. Therefore, we run through the planning graph during the extraction of a plan without making difference between implicit and explicit preconditions. This technique allows us to use black-box functions, because we will be only concerned by the parameters and the returned value of a function to be able to follow its trace through edges during the extraction process.

Algorithm-1 (see Fig. 3) starts by calling an initialization subroutine (see. Fig.2) in which the construction of the planning graph begins by setting the first fact level to the initial state of the planning problem, then by testing if the goal conditions are satisfied at the initial state to return an empty plan, otherwise the algorithm instantiates the propositional variables of all the actions of the planning domain.

```
initialization()/* subroutine to initialize the adapted Graphplan algorithm (Fig-2)*/
begin/*initialisation*/
        Stop:= false;          /*condition to stop the graph expansion */
        i:=0;                  /*the planning graph level number*/
        Fact_i := S_0;         /*the initial planning graph fact level*/
        Facts := {Fact_i};     /*the collection of all the planning graph fact levels*/
        Actions := {};         /*the collection of all the planning graph action levels*/
        MutF := {};            /*the set of mutual exclusions between facts*/
        MutA := {};            /*the set of mutual exclusions between actions */
        Plan := ∅;
        A :={};    /* the set of ground actions */
        /*test if the goal is satisfied in the initial state*/
        Stop=testForSolution(Facts, Actions, MutF, MutA, G, Plan);
        /*instantiate propositional action variables*/
        for all a ∈ Act do
                Instantiate a over P;
                if con_P(a) = true then A:= A ∪ {a};
        end for
end
```

Fig. 2. Adapted Graphplan initialization subroutine

Algorithm-1: ***Adapted Graphplan with external function application***
Input: S_0: initial state, G: Goal conditions, Act: Set of actions
Output: Plan: sequence of ground actions
begin
 call initialization(); */* a subroutine that initializes variables*/*
 */*planning graph construction iterations*/*
 while (*not* Stop) **do**
 $Action_i$:={};
 for all a ∈ A **do**
 InstNumeric(pre_N(a), $Fact_i$);
 if con_N(a) = *true* in $Fact_i$ **then**
 if pre_P (a) ⊆ $Fact_i$ *and* pre_N (a) *are all true in* $Fact_i$ **then**
 InstNumeric(eff_N (a), $Fact_i$);
 $Action_i$:=$Action_i$ ∪ a ;
 PointPreconditions(a, $Fact_i$);
 end if
 end if
 end for
 Actions := Actions ∪ $Action_i$;
 */*Add the facts of previous level with their "no-op" actions"*/*
 i := i+1;
 $Fact_i$:= $Fact_{i-1}$;
 for *each* f ∈ $Fact_{i-1}$ **do**
 $Action_{i-1}$:=$Action_{i-1}$ ∪ "no-op";
 end for
 */*Apply the applicable instantiated actions*/*
 for *all* a ∈ $Action_{i-1}$ and a ≠ "no-op" **do**
 $Fact_i$:=$Fact_i$ ∪ eff_p^+ - eff_p^- ;
 for *each* e ∈ eff_N *such that* e = (n := g) **do**
 if g is an external function **then**
 call the function g;
 else
 evaluate the expression g;
 end if
 n=n ∪ g; */*add a new value to the muli-valued attribute n*/*
 end for
 Connect a to its added or deleted effects;
 / an updated numeric value is considered as being deleted than added*/*
 end for
 Facts := Facts ∪ $Fact_i$;
 calculate the mutex relations for $Action_{i-1}$ and add them to MutA;
 calculate the mutex relations for $Fact_i$ and add them to MutF;
 nonStop=testForSolution(Facts, Actions, MutF, MutA, G, Plan);
 end while
end

Fig. 3. The adapted Graphplan algorithm to support the application of external functions

Propositional instantiation is done once before the graph expansion by substituting the propositional variables by all the possible combinations of the propositional problem objects and by keeping only the instantiated actions that respect the validity of the list of propositional constraints (con_P list). We call these actions initially instantiated for their propositional objects the partially instantiated actions. We note that, none of the numeric variables are substituted before the planning graph expansion process.
Returning back to algorithm-1 (in Fig. 3) after initialization, now at each iteration *i* of the planning graph expansion, the algorithm instantiates the numerical variables of the partially instantiated actions for the actions that respect the validity of the list of numeric constraints (con_N list) and keeps only the actions that have their preconditions (propositional and numeric) satisfied at the fact level *i-1.* The obtained actions form the set of applicable actions at level *i-1*. The algorithm copies all the facts of level *i-1* to the fact level *i* and adds their corresponding "no-op" actions to the action level *i-1 (*one "no-op" for each fact). Each non-"no-op" action of the applicable actions of level *i-1* is applied at the iteration *i,* negative and positive propositional effects are simply added to fact level *i,* and the numeric effects are evaluated by interpreting their arithmetic expressions or by executing their external functions, and then by assigning their returned values to the corresponding numeric variables. After this step, the mutual exclusions between actions of level *i-1* and between actions of level *i* are calculated in the same manner as it is done in the original Graphplan algorithm. At the end of the iteration *i*, the algorithm tests if the goal conditions are satisfied at facts level *i* and there are no mutual exclusions between the facts satisfying the goal, otherwise the algorithm expands a new graph level until finding a valid plan or a sufficient condition (i.e. when the algorithms starts generating duplicated levels in the graph) that indicates that there is no solution to the problem.

In the following we detail the processing of the main functions used by algorithms-1 (Fig. 3):

- **boolean testForSolution**(Facts: the set of all fact levels, Actions: the set of action levels,
  MutF: the set of fact mutexes, MutA: the set of action mutexes,
  G: the set of goal conditions, Plan: ordered set of actions to be returned){
  */*this function tests if G is satisfied in Facts and if a valid plan can be found*/*
  if G *is satisfied in* Facts **then**
  if Actions = {} **then**
  Plan:={};
  return true;
  elseif *the graph is saturated* **then**
  Plan:={'failure'};
  return true;
  else // search for a valid plan
  return ExtractPLAN (Facts, Actions, MutF, MutA, G, Plan)
  end if
  end if
  return false;

}

- **boolean ExtractPLAN**(Facts: the set of all fact levels, Actions: the set of action levels,
  MutF: the set of fact mutexes, MutA: the set of action mutexes,

G: the set of goal conditions, , Plan: ordered set of actions to be returned){

In backward-chaining start by the set of goals at the last level *n*, find a set of actions ("no-op"s included) at level n-1 that add these goal facts. The preconditions to these actions form a set of subgoals at level *n− 1*. Find the actions at level *n-2* that add these subgoals and so forth until reaching the first level *0*. If at a certain level the subgoals can not be reached then try to find a different set of actions that add these subgoals and continue.
If *level 0 is reached* **then**
Plan:= $\cup_{0, n}$ Set of actions that add subgoals;
return true;
end if
//more levels have to be expanded
return false;
}

- **InstNumeric**(N: numeric variables set, Fact: fact level){
  Instantiate the parameters of the external functions and the numeric variables of the expressions in N (e.g. numerical effects of actions) by values from Fact
  (e.g. Numerical fact level items)
}

- **PointPreconditions**(*a*: instantiated action, Fact: fact level){
  Add as preconditions pointers to *a* all the facts from Fact that appear in $pre_P(a)$ as well as in $pre_N(a)$ and in $eff_N(a)$;
  */*A numeric state variable is added implicitly as a precondition to a ground action if it is assigned to an external function or if it appears in an expression or if it is a parameter of an external function of this ground action*/*
}

In figure 4 (see Fig. 4) we show the structure of the planning graph that handles multi-valued numeric variables. The *fact level 0* represents the initial state. It contains two propositional facts "$proposition_1$" and "$proposition_2$" and three numeric variables "n_1", "n_2", "n_3" that have respectively "v_{10}", "v_{20}", "v_{30}" as single values. We note that, at initial state each numeric variable has a single value. At *action level 0*, "*Action1*" and "*Action2*" are considered as having their preconditions satisfied at *fact level 0*. "*Action1*" has 2 satisfied numeric preconditions "$n_1= v_{10}$" and"$n_2=v_{20}$". "*Action2*" has one satisfied numeric precondition "$n_2=v_{20}$" and one satisfied propositional precondition "$proposition_2$" which is true at the current fact level.
Applying the two actions at *level 0*, "*Action1*" is considered to update the values of numeric variables "n_1" and "n_2" respectively to "v_{11}" and "v_{21}" each of which is the result of a function application or an arithmetic expression evaluation. As "*Action1*" modifies the value of "$n_1= v_{10}$" to "$n_1= v_{11}$" (respectively the value of "$n_2= v_{20}$" to "$n_2= v_{21}$") then it is considered as adding "$n_1= v_{11}$" (respectively "$n_2= v_{21}$") and deleting "$n_1= v_{10}$" (respectively "$n_2= v_{20}$"). "*Action2*" is considered as adding "$proposition_3$" and deleting "$proposition_2$".

All facts that are pre-existed in the previous fact level (*level 0*) are connected as being added from this level by "no-op" actions to the next level (*level 1*).

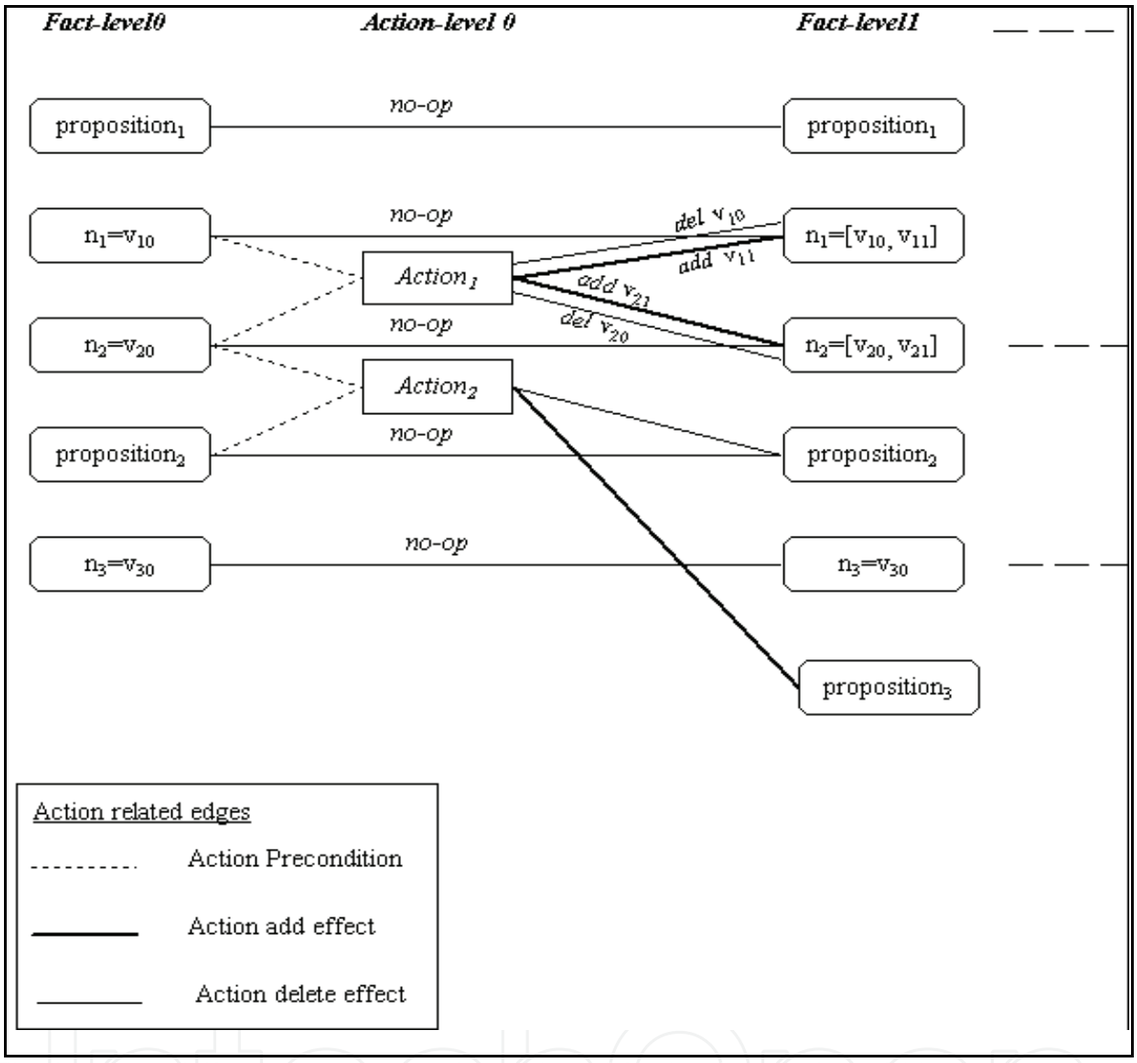

Fig. 4. Adapted Graphplan construction with multi-valued numeric variables

We remark in this figure that, propositional facts are growing horizontally (by line) from level to level and numeric facts are growing vertically (by column). Therefore, the number of numeric variables will not vary during the graph expansion and will stay the same at the last fact level. However, the number of values of each numeric variable can increase with each new expanded fact level.

5. Heuristic search

Solving domain independent planning problems is PSPACE-complete (Bylander, 1994). To reduce this complexity a heuristic function can be used to guide the search for a plan in the search space instead of using a blind search strategy.

The idea of deriving a heuristic function consists of formulating a simplified version of the planning problem by relaxing some constraints of the problem. The relaxed problem can be solved easily and quickly compared to the original problem. The solution of the relaxed problem can then be used as a heuristic function that estimates the distance to the goal in the original problem.
The most common relaxation method used for propositional planning is to ignore the negative effects of actions. This method was originally proposed in (McDermott, 1996) and (Bonet et al., 1997) and then used by the most of propositional heuristic planners (Bonet & Geffner, 2001; Hoffman, 2001; Refanidis & Vlahavas, 2001).
With the arising of planners that solve problems with numerical knowledge such as metric resources and time, a new relaxation method has been proposed to simplify the numerical part of the problem. As proposed in Metric-FF (Hoffmann, 2002) and SAPA (Do & Kambhampati, 2001), relaxing numerical state variables can be achieved by ignoring the decreasing effects of actions. This numerical relaxation has been presented as an extension to the propositional relaxation to solve planning problems that contain propositional and numeric knowledge.
Knowing that, some planning problems contains actions that strictly increase or decrease numeric variables without alternation, other problems uses numeric variables to represent real world objects that have to be handled according to their quantity (Zalaket & Camilleri, 2004a) and thus applying the above proposed numerical relaxation method can be inadmissible to solve this kind of problems. In this section, we start by explaining the relaxed propositional task as it was proposed for STRIPS problems (McDermott, 1996), we introduce a new relaxation method for numerical tasks in which we relax the numeric action effects by ignoring the effects that move away numeric variable values from their goal values, then we present the calculation of a heuristic function using a relaxed planning graph over which we apply the above relaxation methods, and finally we present the use of the obtained heuristic to guide the search for a plan in a variation of hill-climbing algorithm.

5.1 Propositional task relaxation

Relaxing a propositional planning task can be obtained by ignoring the negative effects of actions.
Definition-6: *Given a propositional planning task $P=<S, A, s_I, G>$, the relaxed task P' of P is defined as $P'=<S, A', s_I, G>$, such that: $\forall a \in A$ and $eff_P(a) = eff_P^+(a) \cup eff_P^-(a) \Rightarrow \exists a' \in A' / \quad eff_P(a') = eff_P^+(a)$(which means $eff_P(a') = eff_P(a) - eff_P^-(a)$).*
And thus, $A' = \{ con_P(a), Pre_P(a), eff_P^+(a), \forall a \in A \}$.
The relaxed plan can be solved in polynomial time as it is proven by bylander (Bylander, 1994).

5.2 Numerical task relaxation

Relaxing a numerical planning task can be obtained by ignoring the negative effects of actions that move away numeric values from the goal values.
Definition-7: *Given a numerical planning task $V=<S, A, s_I, G>$, the relaxed task V' of V is defined as $V'=<S, A', s_I, G>$, such that: $\forall a \in A$ and $eff_N(a) = eff_N^+(a) \cup eff_N^-(a)$, such that:*
$\forall (n:=v) \in eff_N(a)$, where n is a numeric variable and v is a constant numeric value that can be the result of an arithmetic expression or an executed external function.
Positive numeric effects $eff_N^+(a)$ and negative numeric effects $eff_N^-(a)$ are defined as follows:

$\forall$ $(n=v_I) \in s_I$, *where* v_I *is a constant numeric value that represents the initial value of n.*

if

$(n\ \theta\ v_G) \in G$, *where* $\theta \in \{<, \leq, =, >, \geq\}$ *and* v_G *is a constant numeric or the result of an arithmetic expression or an executed external function.*

if

distance $(v, v_G) \leq$ ***distance*** (v_I, v_G) ***and distance*** $(v_I, v) \leq$ ***distance*** (v_I, v_G) *(the current value v of the numeric variable n is closer to the goal value* v_G *of n than the initial value* v_I *from the initial value side.)*

then

$(n:=v) \in eff_N^+ (a)$

else

$(n:=v) \in eff_N^- (a)$

endif

else

$(n:=v) \in eff_N^+ (a)$ *//(n does not appear in the goal state)*

end if

Example of the distance calculation:

Assume that:

- We have a numeric variable n which is equal to *0* at the initial state ($v_I=0$) and is equal to *5* in the goal state ($v_G=5$).
- We have an action *a*, which assigns to n respectively the values $v_1=-3$, $v_2=-1$, $v_3=1$, $v_4=5$, $v_5=7$, $v_6=11$.

In this case the distance can be calculated as: $distance\ (v_j, v_i)=|v_j - v_i|$

By testing for relaxed action effects:

$distance(v_I, v_G)= |v_G - v_I|=5$

- $v_1=-3$: $distance(v_1, v_G)=|v_G - v_1|=8 > distance(v_I, v_G) \Rightarrow (v_1=-3) \in eff_N^- (a) \Rightarrow v_1=-3$ *is ignored in the relaxed task.*
- $v_2=-1$: $distance(v_2, v_G)=|v_G - v_2|=6 > distance(v_I, v_G) \Rightarrow (v_2=-1) \in eff_N^- (a) \Rightarrow v_2=-1$ *is ignored in the relaxed task.*
- $v_3=1$: $distance(v_3, v_G)=|v_G - v_3|=4 \leq distance(v_I, v_G)$ *and* $distance(v_I, v_3)=|v_3 - v_I|=1 \leq distance(v_I, v_G) \Rightarrow (v_3= 1) \in eff_N^+ (a) \Rightarrow v_3=1$ *is held in the relaxed task.*
- $v_4=5$: $distance(v_4, v_G)=|v_G - v_4|=0 \leq distance(v_I, v_G)$ *and* $distance(v_I, v_4)=|v_4 - v_I|=5 \leq distance(v_I, v_G) \Rightarrow (v_4= 4) \in eff_N^+ (a) \Rightarrow v_4=4$ *is held in the relaxed task.*
- $v_5=7$: $distance(v_5, v_G)=|v_5 - v_G|=2 \leq distance(v_I, v_G)$, *but* $distance(v_I, v_5)=|v_5 - v_I|=7 > distance(v_I, v_G) \Rightarrow (v_5= 7) \in eff_N^- (a) \Rightarrow v_5=7$ *is ignored in the relaxed task.*
- $v_6=11$: $distance(v_6, v_G)=|v_6 - v_G|=6 > distance(v_I, v_G) \Rightarrow (v_6= 11) \in eff_N^- (a) \Rightarrow v_6=11$ *is ignored in the relaxed task.*

Remarks:

The distance formula can vary according to the comparison operator used in the goal state, but it is the same for all numeric values used in the initial and the goal state.

Each numeric variable that appears in the initial state and doesn't appear in the goal conditions is automatically added to the positive numeric effects, because the values of these variables are often used as preconditions for actions and thus, they can not be ignored.

Figure 5 (see Fig. 5) shows (in red) how negative numeric effects of an action that updates a numeric variable *n* are considered. It also shows (in blue) the positive numeric effects of the action which are considered according to the initial and the goal values of the variable *n*. Note that, exchanging the values of n between initial and goal states will not affect the ranges of selected positive and negative numeric effects.

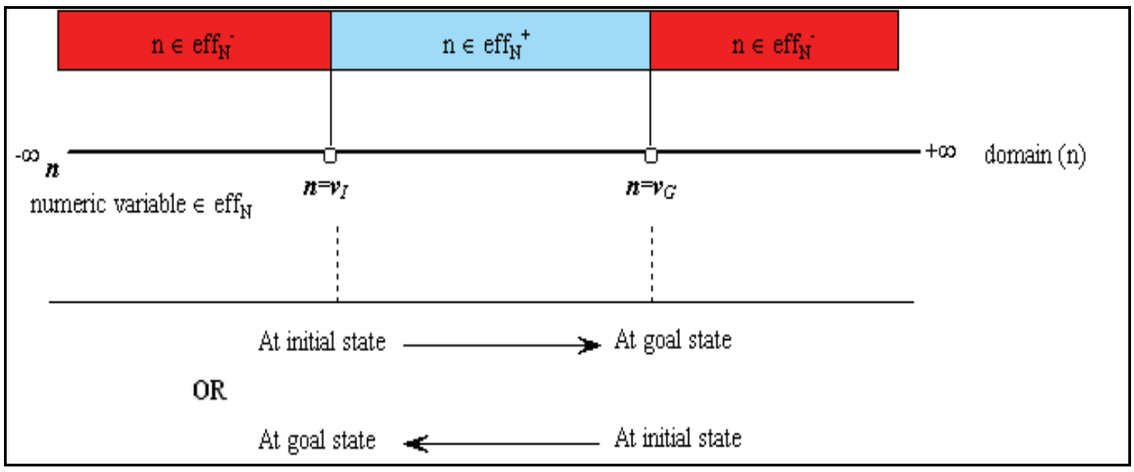

Fig. 5. Choosing negative and positive numeric action effects

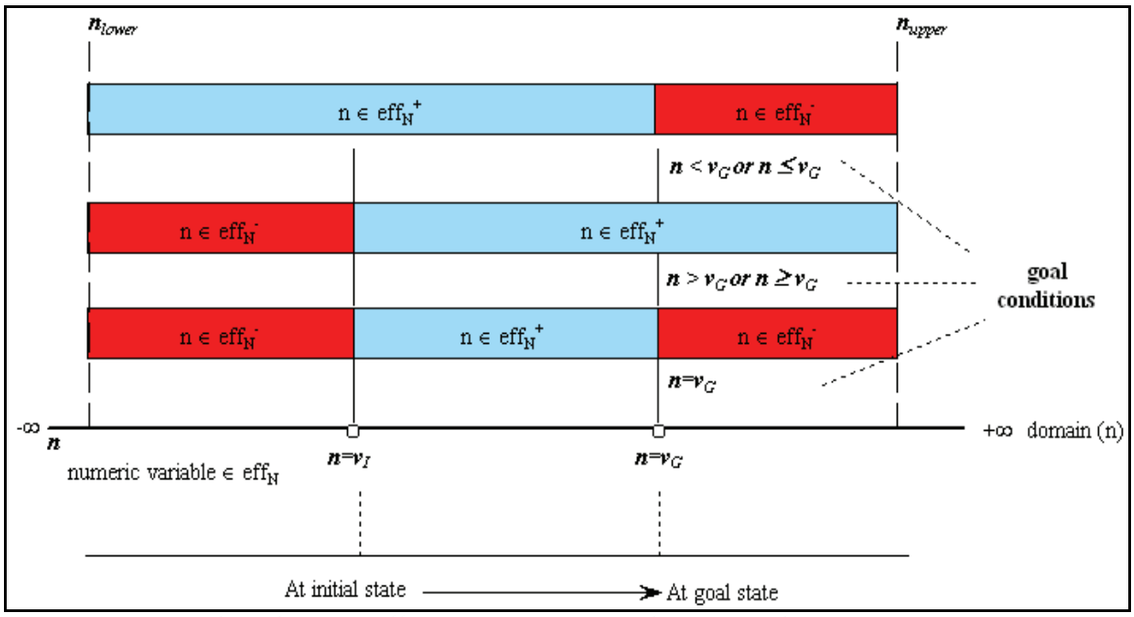

Fig. 6. Numeric relaxed action effects variation according to goal comparison operators.

Figure 6 (see Fig. 6) shows how the selection of negative (in red) and positive (in blue) numeric effects depends on the comparison operator used for comparing the numeric variable *n* in the goal conditions. Therefore, the distance formula is calculated according to the operator used irrespective of the values of n in initial and goal states. As can be observed in this figure, a tighter range of positive numeric effects can be obtained when the equal

operator is used to compare the value of n in the goal conditions, and consequently a smaller search space will be generated for the relaxed problem, which accelerates the process of search for a plan for that problem.

5.3 Mixed planning problem relaxation

Definition-8: *Given a mixed propositional and numerical planning problem $P=<S, A, s_I, G>$, the relaxed problem P' of P is defined as $P'=<S, A', s_I, G>$, such that: $\forall a \in A$ and $eff(a)= eff_P(a) \cup eff_N(a)$ and $eff_P(a) = eff_P^+(a) \cup eff_P^-(a)$ and $eff_N(a) = eff_N^+(a) \cup eff_N^-(a) \Rightarrow \exists a' \in A' / \quad eff(a')= eff_P^+(a) \cup eff_N^+(a)$.*
And thus, $A' = \{ con_P(a), Pre(a), eff^+(a) = eff_P^+(a) \cup eff_N^+(a), \forall a \in A \}$.
Definition-9: *A sequence of applicable actions $\{a_1, a_2, \ldots, a_n\}$ is a relaxed plan for the planning problem $P=<S, A, s_I, G>$ if $\{a'_1, a'_2, \ldots, a'_n\}$ is a plan of its relaxed problem $P'=<S, A', s_I, G>$.*

6. Relaxed planning graph with functions application

Like the planning graph structure used in the adapted Graphplan algorithm, the relaxed planning graph consists of 2 types of levels fact levels and action levels. Algorithm-2 (see Fig. 7) shows how the relaxed planning graph is expanded until reaching a fact level that satisfied the goal conditions or until obtaining consecutive duplicated fact levels. This test is done by using the function *testForSolution(Facts, Actions, G, Plan),* which will be modified compared to the one used in the adapted Graphplan implementation (Fig. 3).
Compared to algorithm-1 (Fig. 3), algorithm-2 (Fig. 7) applies only the positive propositional and numeric effects of actions for generating the next fact level, as discussed in sextion-5. An additional relaxation is added to the planning graph construction in algorithm-2, which consists of ignoring the mutual exclusion between facts and between actions. Therefore, the initialization subroutine for algorithm-2 will be the same as in Fig. 2 but without the mutual exclusion lists. This latter relaxation allows the relaxed planning graph to apply conflicting actions in parallel, and thus to reach the goal state faster in polynomial time.
The test for solution

- **boolean testForSolution**(Facts: the set of all fact levels, Actions: the set of action levels, G: set of goal conditions, Plan: ordered set of the actions to be returned){

```
/*this function tests if G is satisfied in Facts and if a relaxed plan can be found*/
if G is satisfied in Facts then
if Actions = {} then
        Plan:={};
                return true;
elseif the graph is saturated then
        Plan:={'failure'};
                return true;
elseif G is satisfied at Facts[final_level] then
                // extract a relaxed plan see algorithm-3
        ExtractRelaxedPLAN (Facts, Actions, G, Plan)
end if
end if
return false;
}
```

Algorithm-2: ***Relaxed planning graph with external function application***
Input: S_0: initial state, G: Goal conditions, Act: Set of actions

Output: Plan: sequence of ground actions

```
begin
        call initialization(); /* a subroutine that initializes variables*/
        /*relaxed planning graph construction iterations*/
        while (not Stop) do
                Action_i:={};
                for all a ∈ A do
                    InstNumeric(pre_N(a), Fact_i);
                    if con_N(a) = true in Fact_i then
                        if pre_P (a) ⊆ Fact_i and pre_N (a) are all true in Fact_i then
                                InstNumeric(eff_N (a), Fact_i);
                                Action_i:=Action_i ∪ a ;
                                PointPreconditions(a, Fact_i);
                        end if
                    end if
                end for
                Actions := Actions ∪ Action_i;
                /*Add the facts of previous level with their "no-op" actions"*/
                i := i+1;
                Fact_i := Fact_i-1;
                for each f ∈ Fact_i-1 do
                        Action_i-1:=Action_i-1 ∪ "no-op";
                end for
                /*Apply the applicable positive instantiated actions*/
                for all a ∈ Action_i-1 and a ≠ "no-op" do
                        Fact_i:=Fact_i ∪ eff_p+
                        for each e ∈ eff_N+ do
                                if g is an external function then
                                        call the function g;
                                else
                                        evaluate the expression g;
                                end if
                                /*add a new value to the muli-valued attribute n*/
                                n=n ∪ g;
                        end for
                        Connect a to its added effects;
                end for
                Facts := Facts ∪ Fact_i;
                nonStop=testForSolution(Facts, Actions, G, Plan);
        end while
end
```

Fig. 7. The relaxed planning graph construction algorithm

6.1 Relaxed plan extraction

Once the relaxed planning graph is constructed using the algorithm-2 (Fig. 7) up to a level that satisfies the goals, the extraction process can be applied in backward chaining as shown in algorithm-3 (Fig. 8) which details the *ExtractRelaxedPLAN* function called by the *testForSolution* function of in algorithm-2 as detailed in section-6:

Algorithm-3: ***Extract plan in backward chaining from the relaxed planning graph***
Name: **ExtractRelaxedPlan**

Input: Facts: Set of fact levels, G: Goal conditions, Actins: Set of action levels

Output: Plan: sequence of ground actions

```
begin
    Plan:={};
    G_final_level:= { g ∈ Facts[final_level] / g satisfies G};
    for i = final_level to 1 do
        G_i-1:={};
        for each g ∈ G_i do
            acts:= {actions at level final_level-1 that add g};
            selAct:= get_first_element_of(acts);
      if act ≠ "no-op" then
        for act ∈ acts do
                    if act= 'no-op' then
                selAct:=act;
                break;
            end if
            // Select the action that has the minimum number of preconditions
            if nb_preconditions_of(act)< nb_preconditions_of (selAct) then
                selAct:=act;
            end if
        end for
      end if
      plan:=plan ∪ selAct;
      G_i-1:= G_i-1 ∪ { f ∈ Facts[i-1] s.t. f is a precondition of selAct};
        end for
    end for
end
```

Fig. 8. Plan extraction from a relaxed planning graph

Each sub-goal in the final fact level (the level that satisfies the goal conditions), is replaced by the preconditions and by the implicit preconditions (definitions 3 and 4) of the action that adds it and the action is added to the list of relaxed plan. Normally, a "no-op" action will be preferred if it adds a sub-goal. If there is not a "no-op" action that adds the sub-goal and there is more than one action that add it, then we choose the action that has the minimum number of preconditions and implicit preconditions from these latter. We replace the sub-goal fact by the facts that serve as preconditions and implicit preconditions to the chosen

action. We can backtrack in the graph to choose another action adding the sub-goal if a selected action doesn't lead to a solution. Once all goals of the final level are replaced by the sub-goals of previous level, this previous level becomes the final level and the sub-goals become the new goals. This process is repeated until reaching the first fact level. The resulting heuristic is considered as the distance to the goal and it is calculated by counting the number of actions of the relaxed plan.

$h = \sum_{i=0}^{final\_level-1} | a_i |$, where $[a_0, a_1, ..., a_{final\_level-1}]$ is the relaxed plan.

Note that, during the backward plan extraction, we don't make any difference between numeric and propositional facts as all facts even that are results of applied functions are accessed via action edges that are stored in the planning graph structure.

6.2 Heuristic planner running over the effects of applied functions

The main search algorithm that we use to find a plan in the original problem is a variation of hill-climbing search guided by the heuristic *h* detailed in section-6.1. The heuristic is calculated for each state *s* in the search space. At each step we select the child having the lowest heuristic value compared to other children of the same parent to be the next state step, and so on until we reach a state with a heuristic equal to zero. If at some step, algorithm-2 doesn't find a relaxed plan that leads a state *s* to the goal state then the heuristic *h* will be considered as infinite at this step.

Each time a state is selected (except of the initial state) the action which leads to this selected state is added to the plan list. The variation of hill-climbing is when a child having the lowest heuristic is selected, if its heuristic value is greater than the parent state heuristic then the child can be accepted to be the next state step as long as the total number of children exceeding the parent heuristic value is less than a given threshold number. Another variation of hill climbing is: The number of consecutive plateaus (where the calculated heuristic value stays invariable) is accepted up to a prefixed constant. After that a worst-case scenario is launched. This scenario consists of selecting the child who has the lowest heuristic greater than the current state heuristic (invariable), and then to continue the search from this children state by trying to escape the plateau. This scenario can also be repeated up to a prefixed threshold.

In all the above cases, if hill-climbing exceeds one of the quoted thresholds or when the search fails to find a plan the hill-climbing is considered as unable to find a solution and an A* search begins. As HSP and FF, we have added to hill climbing search and to A* search a list of visited states to avoid calculating a heuristic more than once for the same state. At each step a generated state is checked to see if it exists in the list of visited states in order cut it off to avoid cycles. According to our tests, we have noticed that most of the problems can be solved with hill-climbing algorithm. Only some tested domain problems (like ferry with capacity domain) have failed with hill-climbing search so early. But, the solution has been found later with the A* search.

7. Empirical results

We have implemented as prototypes all the above algorithms in Java language. We have run these algorithms over multiple foremost numeric domains that necessitate non classical

handling such as the water jugs domain, the manufacturing domain, the army deployment domain and the numeric ferry domain as introduced in (Zalaket & Camilleri 2004a). We note that, some of these domains such as manufacturing and army deployment are usually expressed and solved with scheduling or with mathematical approach.
Our tests can be summarized in three phases: In the first phase, we have started by running a blind forward planning algorithm that supports the execution of external functions. Our objective at this phase was only to study the feasibility and the effectiveness of integrating such functions written in a host programming language to planning in order to accomplish some complex computation. In the second phase, we have run the adapted Graphplan algorithm with which we have obtained optimal plans for all the problems, but it was not able to solve large problems. In the third phase, we have run the heuristic planner over all the above cited domains. Larger problems are solved with this planner, but the generated plans were not always optimal as it was the case in the second phase.
We have made minor efforts for optimizing our implementation in the one or the other of the above phases. Even though, we can conclude that the heuristic algorithm is the most promising one despite its non-optimal plans. We think that some additional constraints can be added to this algorithm to allow it generating better plans quality. We also remark that some planning domains can be modelled numerically instead of symbolically to obtain extremely better results. For example, in the numeric ferry domain the heuristic algorithm was able to solve problems that move hundreds of cars instead of tenth with classical propositional planners.

8. Conclusion

In this chapter, we have presented multiple extensions for classical planning algorithms in order to allow them to solve more realistic problems. This kind of problems can contain any type of knowledge and can require complex handling which is not yet supported by the existing planning algorithms. Some complicated problems can be expressed with the recent extensions to PDDL language, but the main lack remains especially because of the incapacity of the current planners. We have suggested and tested the integration to planning of external functions written in host programming languages. These functions are useful to handle complicated tasks that require complex numeric computation and conditional behaviour. We have extended the Graphplan algorithm to support the execution of these functions. In this extension to GraphPlan, we have suggested the instantiation of numeric variables of actions incrementally during the expansion of the planning graph. This can restrict the number of ground actions by using for numeric instantiation only the problem instances of the numeric variables instead of using all the instances of the numeric variable domain which can be huge or even infinite. We have also proposed a new approach to relax the numeric effects of actions by ignoring the effects that move away the values of numeric variables from their goal values. We have then used this relaxation method to extract a heuristic which we have used it later in a heuristic planner.
According to our tests on domains like the manufacturing one, we conclude that scheduling problems can be totally integrated into AI planning and solved using our extensions. As future work, we will attempt to test and maybe customize our algorithms to run over some domains adapted from the motion planning, in order to extend the AI planning to also cover

the motion planning and other robotic problems currently solved using mathematical approaches.

9. References

Bacchus, F. & Ady, M. (2001). Planning with resources and concurrency a forward chaining approach. *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, August, 2001, Seattle, Washington, USA

Bak, M.; Poulsen, N. & Ravn, O. (2000). Path following mobile robot in the presence of velocity constraints. *Technical report*, Technical University of Denmark, 2000, Kongens Lyngby, Denmark

Blum, L. & Furst, L. (1995). Fast planning through planning graph analysis. *Proceedingsof the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1636–1642, August, 1995, Montreal, Quebec, Canada

Bonet, B. & Geffner, H. (2001). Planning as heuristic search. *Journal of Artificial Intelligence*, 129:5–33,2001

Bonet, B.; Loerincs, G. & Geffner, H.(1997). A robust and fast action selection mechanism for planning. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 714–719, July, 1997, convention center in Providence, Rhode Island

Bresina, L. J.; Dearden, R.; Meuleau, N; Smith, E. D. & Washington, R. (2002) Planning Under Continuous Time and Resource Uncertainty: A Challenge for AI. *Proceedings of the AIPS Workshop on Planning for Temporal Domains*, pages 91–97, April, 2002, Toulouse, France

Bylander, T. (1994). The computational complexity of propositional strips planning. *Journal of Artificial Intelligence*, 69:165–204, 1994

Cayrol, M. ; Régnier, P. & Vidal, V. (2000). New results about LCGP, a least committed graphplan. *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000),*pages 273–282, 2000, Breckenridge, CO, USA

Do, B. & Kambhampati, S. (2000). Solving planning graph by compiling it into a CSP. *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, 2000, Breckenridge, CO, USA

Do, B. & Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. *Proceedings of the 6th European Conference on Planning (ECP 2001)*, September, 2001, Toledo, Spain

Edelkamp (2002). Mixed propositional and numerical planning in the model checking integrated planning system. *Proceedings of the AIPS Workshop on Planning for Temporal Domains*, April, 2002, Toulouse, France

Fikes, R.E. & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Journal of Artificial Intelligence*, 2:189–208, 1971.

Fox, M. & Long, D. (2002). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Proceedings of the 7th International Conference on Artificial Intelligence Planning and Scheduling (AIPS- 2002)*, April, 2002, Toulouse, France

Geffner, H. (1999). Functional strips: A more flexible language for planning and problem solving. *Logicbased AI Workshop,* June, 1999, Washington D.C.

Gerevini, A. & Long, D. (2005). Plan constraints and preferences for PDDL3. *Technical Report Technical report*, R.T. 2005-08-07, Dept. of Electronics for Automation, 2005, University of Brescia, Brescia, Italy

Ghallab, M.; Howe, A.; Knoblock, G.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D. & Wilkins, D. (1998). PDDL : The planning domain definition language, version 1.2. *Technical Report CVC TR-98 003/DCS TR-1165. Yale Center for Computational Vision and Control*, October, 1998, Yale, USA

Hoffman, J. (2001) FF: The fast-forward planning system. *AI Magazine*, 22:57 – 62, 2001.

Hoffmann, J. (2002). Extending FF to numerical state variables. *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI2002),* pages : 571-575, July, 2002, Lyon, France

Hoffmann, J.; Kautz, H.; Gomes, C. & Selman B. (2007). SAT encodings of state-space reachability problems in numeric domains. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07),* pages 1918– 1923, January, 2007, Hyderabad, India

McDermott, D. (1996). A heuristic estimator for means ends analysis in planning. *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems.* May, 1996, Edinburgh, UK.

Refanidis, I. & Vlahavas, I. (2001). The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, 15:115–161, 2001.

Samson, C. & Micaelli, A. (1993). Trajectory tracking for unicycle-type and Two steering-wheels mobile robots. *Technical report*, Institut National de Recherche en Automatique, 1993, Sophia-Anitpolis, France

Schmid, U.; Müller, M. & Wysotzki, F. (2002). Integrating function application in state based planning. *Proceedings of the 25th Annual German Conference on AI: Advances in Artificial Intelligence*, pages 144 – 162, September 2002, Aachen, Germany.

Smith, D. & Weld, D. (1999). Temporal planning with mutual exclusion reasoning. *Proceedings of 16th InternationalJoint Conference on Artificial Intelligence (IJCAI-99)*, August, 1999, Stockholm, Sweden

Zalaket, J. & Camilleri, G. (2004a). FHP : Functional heuristic planning. *Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2004)*, pages 9–16, September, 2004, Wellington, New Zealand

Zalaket, J. & Camilleri, G. (2004b). NGP: Numerical graph planning. *proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004),* pages 1115–1116, August, 2004, Valencia, Spain.

Frontiers in Robotics, Automation and Control

Edited by Alexander Zemliak

ISBN 978-953-7619-17-6
Hard cover, 450 pages
Publisher InTech
Published online 01, October, 2008
Published in print edition October, 2008

This book includes 23 chapters introducing basic research, advanced developments and applications. The book covers topics such us modeling and practical realization of robotic control for different applications, researching of the problems of stability and robustness, automation in algorithm and program developments with application in speech signal processing and linguistic research, system's applied control, computations, and control theory application in mechanics and electronics.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen