When Planning Should Be Easy: On Solving Cumulative Planning Problems

Roman Barták, Filip Dvořák, Jakub Gemrot, Cyril Brom, Daniel Toropila

Charles University in Prague, Faculty of Mathematics and Physics, Malostranské nám. 25, 118 00 Praha 1, Czech Republic
name.surname@mff.cuni.cz

Abstract

This paper deals with planning domains that appear in computer games, especially when modeling intelligent virtual agents. Some of these domains contain only actions with no negative effects and are thus treated as easy from the planning perspective. We propose two new techniques to solve the problems in these planning domains, a heuristic search algorithm ANA* and a constraint based planner RelaxPlan, and we compare them with the state of the art planners, that were successful in IPC, using planning domains motivated by computer games.

Introduction

Though certain planning problems are assumed by the planning community to be easy for solving, these problems still appear in many practical applications. Hence it is useful to have efficient solving approaches for such "easy" problems. In this paper we will look at planning problems that appear in some computer games. These problems can be characterized as cumulative planning problems where actions have no negative effects. Finding a plan solving such a problem can indeed be done in polynomial time (Blum and Furst 1997) but it is more complicated to find the optimal plan (with the smallest sum of action costs).

There has been a huge progress in developing automated planners in recent years so the hope was that the best planners should be able to solve the above mentioned "easy" problems. However, our initial experimental evaluation showed that this is not the case. Hence we decided to develop specific planners for solving the cumulative planning problems to demonstrate that the planning technology is ready to solve such problems. The first planner we have developed belongs to the category of heuristic search planners, which is one of the most successful approaches to automated planning in recent years, at least, if we measure the success via the results of International Planning Competition (IPC). The second planner exploits constraint satisfaction technology, which is not that successful in IPC. This planner was originally designed just for comparison with the heuristic search planner but as we will see later, it performs quite well.

The paper is organized as follows. We will first give the motivation for studying cumulative planning problems originated in computer games. Then we will describe two approaches for solving the planning problem till optimality, namely a heuristic search algorithm ANA* and a constraint-based planner RelaxPlan. Finally, we will experimentally compare these two solving approaches with three classical planners: LAMA 2011, SelMax, and Fast Downward: Stone Soup 1 (FDSS1).

Motivation

It is known that planners can be useful for extending capabilities of intelligent virtual agents (IVAs) and make them more realistic. The classical example of successfully used planning techniques is the videogame F.E.A.R. (Orkin 2006) that uses an ad-hoc STRIPS-like planner to plan actions for its IVAs. Another example is the video game Iceblox that can be played by off-the-shelf PDDL-based planners (Bartheye and Jacopin 2008).

In this paper we focus on using planning techniques during the off-line validation of planning scenarios. Games played from the first or third person perspective, e.g., Hitman, Metro 2033, are usually split into multiple stand-alone scenarios that are played in a fixed sequence. Each scenario can usually be finished in several ways, in other words, there are multiple plans the player can adopt to fulfill the scenario. As game environments are getting increasingly more complex, designers may fail to realize that the scenario can be finished with simple plans, which

is a thing to avoid in game design, as players may bypass intended storyline, interesting content, or game events. Therefore, it is preferable to model every scenario as a problem domain that describes the environment as well as available player actions and then ask for "all" possible ways how to solve the scenario. The resulting plans can be reviewed and if an unintended solution is discovered, the designer can alter the environment to prevent adoption of such a plan. The described approach was used for instance in the context of Hitman (Pizzi et al. 2008). In this paper we adopted the idea of finding the shortest plan that may easily reveal if there are any unintended shortcuts.

Some games are sequential in terms that the state of the virtual environment does not reoccur or the reoccurrence does not bring anything new for the player and therefore it can be ignored. This means that the vast majority of player actions can be treated as irreversible, which allows modeling possible negative effects (and negative preconditions) of actions as positive effects (and positive preconditions) by creating new atoms prefixed with "not ". This observation greatly simplifies planning problems as it brings the complexity of search for a satisficing solution from (possibly) EXPSPACE to PTIME. Briefly speaking, we can see the problem as a planning problem where actions have only "add" effects and no "delete" effects. A typical example is the game where the player collects keys to open doors and the doors stay opened indefinitely and the collected keys are never lost. Obviously it is enough to apply the action of opening particular doors (or collecting a particular key) at most once. This is the case of cumulative planning where the main problem is to decide whether or not a given action is part of the plan.

Solving Cumulative Planning Problems

As described in the previous section, we need to solve cumulative planning problems where actions have only positive effects. We use the classical propositional representation of planning problems so it means that no atom is ever deleted by executing the plan (atoms are only added). These problems are known to be solvable in polynomial time (Blum and Furst 1997). However, we attempt to find the shortest possible plan which is a problem that is NP-hard (the Set Cover Problem can be converted to the cumulative planning problem).

Though the assumed planning problems should be easier to solve than most problems from IPC, the initial experiments with the winning general planners from IPC showed that these planners have some difficulty with solving this type of problems. Hence we decided to implement specific planners for solving the cumulative planning problems to demonstrate that planning techniques can indeed be used to solve even large-scale planning problems in computer games.

Heuristic Search – ANA*

In classical planning, heuristic search is a common and successful approach to planning. For example, the winners of the latest IPC 2011, Fast Downward Stone Soup-1 (Helmert et al. 2011) and LAMA 2011 (Richter et al. 2011), are both heuristic search planners. The key components of a heuristic planner are the search algorithm and the heuristic estimators.

Heuristics. In our approach for solving the cumulative planning problems we take an advantage of the significant research effort invested in developing delete-relaxation heuristics (Helmert and Domshlak 2009). For our purpose we have chosen the current state-of-the-art heuristic among them, $h^{LM\text{-}cut}$ (Bonet and Helmert 2010). Due to its technical complexity, we do not describe the heuristic here, but for the purpose of this paper we assume that we have an admissible heuristic for the cumulative planning problem that can be computed in polynomial time.

Search algorithm. A* is a well-known search algorithm that always expands the most promising state. Given an admissible heuristic, A* is optimal, although for practical problems searching for the optimal solution tends to be computationally expensive. Anytime Non-parametric A* algorithm (van der Berg et al. 2011) is an A*-based algorithm that performs the greediest possible search to improve the current best solution. We have adapted ANA* for the cumulative planning.

```
solve^ana(s_0, Ac, Goal)
  if not goal reachable(s_0, Ac, Goal) then return ∅
  landmark actions ← find(s_0, Ac, Goal)
  plan ← ∅; G ← |Ac|
  op ← {s_0} // open queue
  cl ← ∅ // closed list
  while op ≠ ∅ & not arbitrary break do
    s ← best^ana(op); op ← op \ {s}
    h(s) ← h^LM-cut(s)
    cl ← cl ∪ {s}
    G ← min(G, g(s) + |relevant actions(s, Actions)|)
    foreach a ∈ applicable relevant actions(s) do
      next ← π(s,a,landmark actions)
      h(next) ← h(s) - cost(all applied actions)
      if is solution(next, Goal) then
        plan ← get plan(next)
      else if not (next ∈ op & g(next) ≥ g(op.next)) and
              not (next ∈ cl & g(next) ≥ g(cl.next)) then
        op ← op ∪ {next}
      end if
    end for
  end while
  return plan
```

Solver. The input of the algorithm is the initial state, the set of actions and the set of goal atoms. The output is the shortest plan found in the given time, or an empty set, if the algorithm did not find any plan. The algorithm is anytime; it can be terminated arbitrarily by breaking the main while-loop, in which case it returns the best complete plan it discovered. The initial step is a goal reachability check that decides reachability of the goal atoms in polynomial time. The next step of the algorithm is finding all landmark actions, which is easy in cumulative planning.

We initialize the *open* queue with the initial state, leave the closed list empty and set the value of the best known solution to be the total number of actions; since the goal is reachable, a plan that contains all the actions is a trivial solution. The main while-loop operates until all the states in the open queue have been explored or filtered out. Note that our queue automatically removes the states that cannot provably improve the best-known solution ($g(s) + h(s) \geq G$). In the first step of the iteration we find and remove the best state from the open queue according to the ANA* evaluation of states. At this point we find the heuristic value for the state according to $h^{LM\text{-}cut}$, add it to the closed list and improve the upper bound for the best solution. The *relevant action* is an action that can be further added into the plan; in other words, it is an action whose effects were not yet achieved and at least one of its effects is either an unachieved goal or a precondition of another relevant action. Consequently, we can see that we can construct a plan by adding all relevant actions into the current plan, which gives us potentially a new reduction of the upper bound for the cost of the best solution.

The inner foreach loop expands the current state by the application of all the actions that are both relevant and applicable (their preconditions are satisfied in the current state). The first step of the cycle applies the transition function π, which is slightly different from the usual transition function; we first apply the chosen action a, then we try to apply the largest number of landmark actions we can. This way we exploit the fact that since a landmark action must be in every plan exactly once, it does not matter when we apply it; hence we can apply it the first time it is possible. Since computing $h^{LM\text{-}cut}$ is expensive, in this step we calculate the heuristic value for the new state from the value of the parent state by reducing the heuristic value of the parent state by the costs of all the actions that were applied in the previous step. Lucky we are, this does not affect admissibility, since in cumulative planning the application of an action cannot increase the heuristic value as it can in classical planning (by the application of an action we only increase the number of covered atoms, but we do not lose any). The concept of using a heuristic value of the parent state is known as deferred evaluation (Richter and Helmert 2009). Finally, if we have reached the goal, we record the plan, otherwise we add the state into the open queue unless there exists either the same state in the open queue with the same or better cost or the same state is in the closed list with the same or better cost.

Since we use an admissible heuristic, the algorithm is complete; we never discard a state, unless we have either the same state with a less costly partial plan or a complete plan whose cost is lower than the admissible estimate of the state. Further, all the actions applicable for each state are systematically explored. Therefore, the algorithm eventually finds one of the optimal solutions.

The algorithm is sound, which again comes from the admissibility of the used heuristic and the systematical exploration of all states that are pruned only once their lower bound exceeds the global upper bound.

Constraint-based Planner – RelaxPlan

Classical constraint-based planners are based on the idea of translating the planning problem into a sequence of constraint satisfaction problems (CSPs) where the *i*-th CSP defines the problem of finding a plan of length *i* (Kautz and Selman 1992). This incremental approach is not necessary for cumulative planning problems where we are deciding "only" whether a given action is or is not part of the plan. Hence a single CSP can be used to describe the planning problem. This is similar to the original constraint model in the CPT planner (Vidal and Geffner 2004).

Constraint Model. In the planning problem we have two types of objects: atoms and actions. We need to decide which actions will be in the plan and what the positions (levels) of actions will be in the plan. By selecting the actions we are also deciding which atoms will become true and where in the plan. Obviously, an action can be in the plan only if all atoms from its precondition become true at the levels before the level of the action. Similarly, an atom will become true only if some action having this atom among its effects is in the plan. Then the atom becomes true at the level of the action. Hence the levels define the partial ordering of actions in the plan.

The above observations lead to the following constraint model. For each action a we introduce a variable $ActLevel_a$ describing the position of the action in the plan. Similarly, for each atom (predicate) p we introduce $PredLevel_p$. If we have m actions and n atoms in the problem then the domain for the *Level* variables is $\{0,\ldots,\min(m,n)\}$. Obviously, there must be at least one action at each level and at least one atom must become true at each level (if no new atom becomes true at certain level l then all actions from the next level can be processed at level l too). Note that we allow parallel plans so it is possible to have more than one action at a given level. For atoms we also introduce variables specifying the action that makes the particular atom p true: $PredAction_p$. The domain of this variable contains identifications of actions that have atom p among their effects. This is similar to the model from (Do and Kambhampati 2000). For the atoms that are true in the initial state we set $PredLevel_p = 0$ and $PredAction_p = 0$.

Basically, there are two types of constraints connecting the actions with the atoms:

$$PredLevel_p = ActLevel_{PredAction_a}$$

$$ActLevel_a > \max\{ PredLevel_p \mid p \in \text{precond}(a)\}$$

The first constraint says that the predicate becomes true at the level where the action giving that predicate is applied. The second constraint says that an action can be applied at the level following the level where all atoms from the action precondition are true. The action can also be applied later if it is applied at all. To distinguish whether the action is applied (it is a part of the plan) or not we introduce auxiliary variable *GoalLevel* specifying the level where all the goal predicates are true. Assume that G is a set of goal atoms, then $GoalLevel = \max\{ PredLevel_p \mid p \in G \}$. Any action applied before or at the *GoalLevel* is part of the plan which is indicated by the Boolean variable B_a:

$$B_a = 0 \Leftrightarrow GoalLevel < ActLevel_a.$$

These Boolean variables also define the objective function to be minimized (action cost can also be added):

$$Obj = \Sigma_a B_a.$$

Search Strategy. We applied the backward planning approach so we start with the set G of goal atom(s) and for each goal atom in this set we try to find an action having this atom among its effects (this action determines the atom). When this action is decided, its preconditions are added to the set of goal atoms and the process is repeated until the set of goal atoms becomes empty. To ensure that the predicates are not repeatedly added to G we filter out from G those predicates that have already been processed.

```
solve(Goals)
    Goals ← filterGoal(Goals)
    if Goals = ∅ then return true
    g ← selectGoal(Goals)
    Actions ← domain(PredAction_g)
    while Actions ≠ ∅ do
        a ← selectAction(Actions)
        PredAction_g ← a
        if solve(Goals \ {g} ∪ precond(a)) then return true
        un-assign PredAction_g
        Actions ← Actions \ {a}
        PredLevel_p < ActLevel_a            // C1
    end while
    return fail {remove constraints C1}
```

The goal atom is selected from the set G of goal atoms using the first-fail principle (variable selection in a CSP). In particular, we prefer atoms with the smallest number of possible determining actions (the domain of variable *PredAction* is the smallest one). In case of tie, we prefer atoms that appear later in the plan (the minimal value in the domain of variable *PredLevel* is maximal among all the atoms). Formally, we select the goal atom:

$\text{argmin}\{ (s,-k) \mid s = \text{size}(PredAction_i), k = \min(PredLevel_i), i \in Goals\}$,

where size(X) is the size of the domain of X and min(X) is the minimal value in the domain of X.

When the goal atom p is selected we need to decide the action determining this goal (value selection in a CSP). We prefer actions, which are already in the plan, to actions that have not been decided yet. In case of tie, we prefer actions that may appear earlier in the plan, that is, actions with the smallest minimal value in the domain of variable $ActLevel_a$. Finally, in case of tie, we prefer actions with the smallest number of preconditions. When action a is selected it is assigned to the variable $PredAction_p$ and the atoms from the precondition of a are added to the set of goal atoms. As there may be more actions, which give atom p, this step introduces a choice point. In case of backtracking and before assigning another action to $PredAction_p$ we post a constraint $PredLevel_p < ActLevel_a$ to ensure that action a will not give atom p in the alternative search branch (the option where a gives p has already been explored).

The reader may notice that we instantiate only the variables *PredAction*. This can be done thanks to maintaining consistency of inequality constraints "<" that discovers infeasibilities such as A < B < C < A, that is a cycle of actions. Hence if we decide which actions are giving which atoms we guarantee that there exists some allocation of actions to levels that forms a parallel plan.

Note finally that optimization is realized using the standard branch-and-bound approach. In particular, when procedure Solve finishes we set $Obj \leftarrow \min(Obj)$ which moves (via constraint propagation) all non-used actions after *GoalLevel*, that is, outside the plan. The value of *Obj* is then used as a bound when continuing search and looking for the solution with a smaller value of *Obj*.

Experimental Results

To compare the presented planning techniques we generated random problems modeling the problems from computer games as discussed in the introduction. The problems can be characterized by parameters (n, m, i, j). We generate a planning graph with n action layers where each layer contains 1 to m actions (random distribution) and each action has i positive preconditions (randomly selected atoms from the previous state layers) and j positive effects (randomly generated atoms including those from the previous state layers). Only the actions in the first layer contain a single precondition – a special atom *start* that forms the initial state. The last action layer contains at least one action that has a single effect *goal*. The task is to select the smallest number of actions that form a plan achieving the *goal* atom. Figure 1 shows the structure of the used planning graph. Though this type of domains seems very restricted; it suffices to model many planning problems appearing in computer games.

We generated 432 instances of random planning problems with the number of action layers (n) ranging from 1 to 204. For most problems we used 10 actions per

layer (m), with the exception of a few selected configurations for which we generated instances with 1 to 20 actions per layer. For all generated actions we randomly selected 1 to 3 atoms as preconditions and 2 to 3 atoms as effects. All planning problems were generated using the standard PDDL syntax.

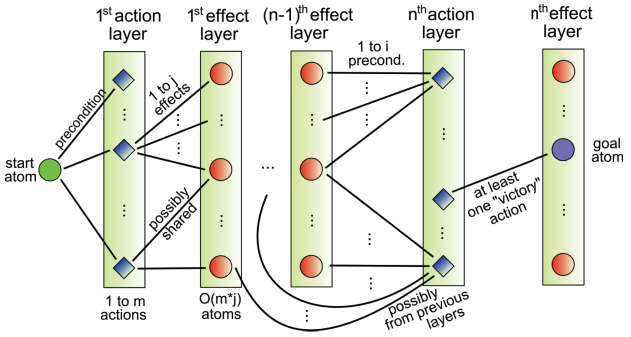

Figure 1. Planning graph structure used in the experiments.

The experiments ran on Intel Xeon CPU E5335 2.0 GHz processor with 8GB RAM under Ubuntu Linux 8.04.2 (Hardy Heron). The ANA* planner was implemented in Java 1.6, the RelaxPlan planner was implemented using the clpfd library of SICStus Prolog 4.2.0. Because of the big number of tested problem instances, the time limit for solving a single planning problem was set to 5 minutes.

In order to evaluate the performance of the introduced new planners we decided to compare them to the three state-of-the-art domain-independent planners that were successful in IPC 2011. Namely, as we were interested in finding optimal (shortest) plans, the obvious choice was the winner of the sequential optimal track, Fast Downward: Stone Soup 1 (FDSS1) (Helmert et al. 2011), which uses a portfolio of selected successful planning techniques, such as BJOLP (Big Joint Optimal Landmark Planner) (Domshlak et al. 2011), LM-cut (A* with the landmark-cut heuristics) (Helmert and Domshlak 2009), or M&S-bisim1 and M&S-bisim2 (A* with two different merge-and-shrink heuristics) (Nissim et al. 2011). Each ingredient of the portfolio is then assigned a given amount of time limit based on its performance using the method described in (Helmert et al. 2011). Both the selected techniques and their assigned time are derived based on the experiments with the planning domains used for the IPC. However, since we were interested in solving the cumulative planning problems, we felt the urge to compare the performance also with a planner whose parameters were not derived based on the IPC domains. Therefore we chose the third best-performing planner from the IPC 2011 (the second place was taken by yet another variation of the FDSS planner), the SelMax planner, which combines two state-of-the-art admissible heuristics using an online learning approach (Domshlak et al. 2011). Finally, we used LAMA 2011 planner (Richter et al. 2011), the winner of the sequential satisficing track of IPC 2011.

Table 1 summarizes the results of our experimental evaluation. As it can be seen, out of the 432 tested instances the existing state-of-the-art optimal planners, FDSS1 and SelMax, managed to solve 233 and 235 planning problems, respectively, without providing any sub-optimal solutions for the unsolved problems. This fact was the reason to include also the LAMA 2011 planner to our experiments, which managed to provide a (sub-)optimal solution for all tested instances, however, only 64 of them were solved optimally. Nevertheless as the column Total Cost shows, LAMA 2011 found many optimal plans, just the proof of optimality was missing. On the other hand, the first of the newly proposed planners, the ANA* planner, solved 241 planning problems, while providing a sub-optimal solutions for all but one testing instance. The clear winner of our experiments is the RelaxPlan planner, which solved optimally 313 problems and moreover it also provided sub-optimal solutions for all of the instances. Also, it required the least amount of time necessary to finish the computation. Though ANA* reached the best total cost of found plans, the differences between the three best planners are only tiny, which is interesting especially for LAMA 2011 that provided the guarantee of optimality only in 64 cases, as opposed to RelaxPlan which solved optimally 313 instances.

Planner	***Solved***	***Optimally Solved***	***Total Cost***	***Total Time***
FDSS1	233	233	-	30.58h
SelMax	235	235	-	17.66h
LAMA 2011	432	64	17768	30.88h
ANA*	431	241	17681	16.70h
RelaxPlan	432	313	17756	11.51h

Table 1. Planner performance comparison for solving 432 random cumulative planning problems.

Figure 2 depicts the difference between the times needed to optimally solve the planning problems. For clarity we only include data for ANA*, RelaxPlan and SelMax planners (the SelMax planner exhibited the best performance out of the three existing planners).

As it can be seen, the new planners not only provide the combined advantages of modern satisficing and optimal planners, but also greatly outperform them when solving cumulative planning problems.

Conclusions

Computer games and digital entertainment provide many challenges for artificial intelligence and in particular for planning. Though some planning problems in these areas

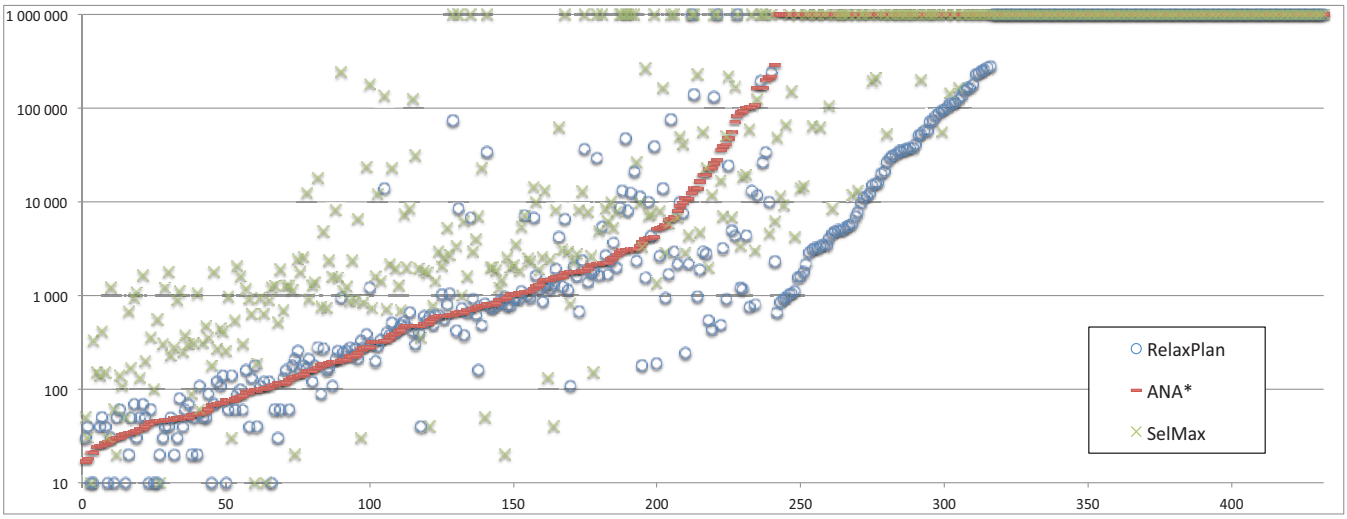

Figure 2. Comparison of runtimes (logarithmic scale) for generated cumulative planning problems. X axis represents 432 planning problems sorted by ANA* solving time (RelaxPlan solving time is used for sorting when ANA* exceeded the time limit), Y axis represents the runtime in milliseconds required to solve a given planning problem optimally. For the cases when time out (set to 5 minutes) occurred we used the value of 1.000.000 to depict them, in order to visually separate such cases from the optimally solved instances more clearly.

seem easy, for example the cumulative planning problems, the current state-of-the-art planners have some difficulties to solve them. Hence we proposed two new planners for solving the cumulative planning problems. These planners are not fine-tuned regarding the implementation but they are still beating the best planners from IPC when applied to specific planning problems appearing in certain computer games. So far we did a somehow restricted experimental study but the results naturally raise a more general question – whether the IPC domains examine the planners well in relation to the problems appearing in practice. We left this question un-answered, but the results from this paper suggest that there is indeed a gap between academic planning techniques and close to real-life problems.

Acknowledgement

Research is supported by the Czech Science Foundation under the contract P103/10/1287, and by the Grant Agency of Charles University as projects 306011 and 9710/2011.

References

Bartheye, O. and Jacopin, É. 2008. Connecting PDDL based off the shelf planners to an arcade game. *ECAI Ws on AI in Games*.

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90, 281 300.

Bonet, B. and Helmert, M. 2010. Strengthening Landmark Heuristics via Hitting Sets. *ECAI 2010*, IOS Press, 329 334.

Do, M.B. and Kambhampati, S. 2000. Solving planning graph by compiling it into CSP. *AIPS 2000*, AAAI Press, 82 91.

Domshlak, C.; Helmert, M.; Karpas, E.; Keyder, E.; Richter, S.; Röger, G.; Seipp, J. and Westphal, M. 2011. BJOLP: The Big Joint Optimal Landmarks Planner. *IPC 2011,* 91 95.

Domshlak, C.; Helmert, M.; Karpas, E. and Markovitch, S. 2011. The SelMax Planner: Online Learning for Speeding up Optimal Planning. *IPC 2011,* 108 112.

Helmert, M. and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? *ICAPS 2009*, AAAI Press, 162 169.

Helmert, H.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S. and Westphal, M. 2011. Fast Downward Stone Soup, *IPC 2011*, 38 45.

Kautz, H. and Selman, B. 1992. Planning as satisfiability. *ECAI 1992*, 359 363.

Nissim, R.; Hoffmann, J. and Helmert, M. 2011. The Merge and Shrink Planner: Bisimulation based Abstraction for Optimal Planning. *IPC 2011*, 106 107.

Orkin, J. 2006. Three states and a plan: the AI of FEAR. *Game Developers Conference*.

Pizzi, D.; Cavazza, M.; Whittaker, A. and Lugrin, J. L. 2008. Automatic Generation of Game Level Solutions as Storyboards. *AIIDE 2008*, AAAI Press, 96 101.

Richter, S. and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. *ICAPS 2009*, AAAI Press, 273 280.

Richter, S.; Westphal, M. and Helmert, M. 2011. LAMA 2008 and 2011, *IPC 2011,* 50 54.

van den Berg, J.; Shah, R.; Huang, A. and Goldberg, K. 2011. ANA*: Anytime Nonparametric A*. *AAAI 2011*, AAAI Press, 105 111.

Vidal, V. and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *AAAI 2004*, AAAI Press, 570 577.