YAHSP2: Keep It Simple, Stupid

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Abstract

The idea of computing lookahead plans from relaxed plans and using them in the forward state-space heuristic search YAHSP planner has first been published in 2003. We show in this paper that this simple idea still leads to very efficient planners in comparison with state-of-the-art planners, in terms of running time. We describe the new implementation of lookahead search that has been made in the second version of the YAHSP planner, which has been considerably simplified since the first implementation. We then show through an extensive comparison, over all existing IPC benchmarks, that the resulting YAHSP2 planner outperforms state-of-the-art planners in terms of cumulated number of solved problems and running time. We also briefly describe YAHSP2-MT, an attempt to parallelize YAHSP2 for multi-core machines with shared memory.

Introduction

Since the 6th edition of the deterministic part of the International Planning Competition (IPC) in 2008, an emphasis has been put on solution quality rather than on speed in computing a single plan. In the 2008 and 2011 competitions, deterministic planners were run during a fixed amount of time and their objective was to find the best possible plan within this time constraint. Although for real-world applications plan quality is generally as important as finding a solution (if not even more), we think that designing fast planners is still a relevant task. Particularly, deterministic planners can be embedded into wider systems that frequently call them with different initial states, goals or even domain definitions, and use the solution plans for a particular objective. For example, the probabilistic planners FF-Replan (Yoon, Fern, and Givan 2007) and RFF (Teichteil-Königsbuch, Kuter, and Infantes 2010), winners of the probabilistic tracks of the non-deterministic IPCs in 2004 and 2008 respectively, make heavy use of the FF planner to solve determinized problems extracted from the probabilistic one. They then combine the solutions given by FF into a policy for the probabilistic problem. Another example is the DAE_X planner (Bibai et al. 2010; Dréo et al. 2011), which embeds the YAHSP planner (Vidal 2004) into an evolutionary algorithm whose objective is to produce optimized plans. Optimization is performed through the evolution of a population of individuals, which represent sequences of intermediate goals that must be reached in turn from the initial state to the goal of the problem, by successive calls to YAHSP with an upper bound on the number of expanded nodes. Within a typical single run of DAE_X during 30 minutes, YAHSP may be called hundreds of thousands of times. The need for a fast planner to embed in DAE_X motivated the design of the YAHSP2 planner. Indeed, in opposition to modern planners such as LAMA (Richter and Westphal 2010) which require heavy preprocessing techniques for each different problem, even on the same domain (transformation to SAS+, landmark generation, landmark orderings, etc.), YAHSP does not perform any preprocessing, computing everything on-the-fly during search. Embedded into a wider system, search in YAHSP for a new initial state and goal can then be performed immediately, allowing fast and frequent calls.

The goals in the design of a new version of YAHSP were (1) to extend its expressivity to cost-based and temporal planning, and (2) to simplify its implementation with efficiency in mind. The former has been easily performed: YAHSP2 simply does not take into account costs and durations when computing a single solution, and performs a post-deordering (Bäckström 1998) of the sequential solution plans to produce concurrent plans (de facto forbidding temporally expressive planning). The idea behind this is that the planner embedded into DAE_X should concentrate on the task of finding a plan, working only on the combinatorial problem, the optimization being held by the evolutionary algorithm. In order to fully use the time contract of 30 minutes in the IPC, search in YAHSP2 alone is pursued when solutions are found and states whose cost (plan length, sum of action costs, or makespan after deordering) exceeds the best cost found so far are pruned. One subtlety is that deordering for temporal planning is made during search, in order to be able to perform that pruning. The latter goal in the design of YAHSP2, simplicity, has consisted in simplifying the way relaxed plans and lookahead plans are computed, and removing many other ideas introduced in the first version of YAHSP which were not strictly needed to reach good performance. Indeed, some of these ideas were useful in some cases on the very limited number of benchmarks available when YAHSP has been conceived, but do not reveal to be that interesting when performing experiments on the full set of benchmarks which is now available.

We provide in this paper a complete picture of the techniques and algorithms used in the YAHSP2 planner as it has been entered into the 7th International Planning Competition. We also show through an extensive experimental evaluation that YAHSP2 improves the state-of-the-art (before the competition!) in terms of cumulated number of solved problems and running time efficiency for finding a single plan. We finish by a short description of YAHSP2-MT, an attempt to benefit from multi-core processors in lookahead heuristic search planning previously detailed in (Vidal, Bordeaux, and Hamadi 2010).

Background

The basic STRIPS model of planning can be defined as follows. A *state* of the world is represented by a set of ground atoms. A *ground action* a built from a set of atoms A is a tuple $\langle pre(a), add(a), del(a)\rangle$ where $pre(a) \subseteq A$, $add(a) \subseteq A$ and $del(a) \subseteq A$ represent the preconditions, add effects and del effects of a respectively. A *planning problem* can be defined as a tuple $\Pi = \langle A, O, I, G\rangle$, where A is a finite set of *atoms*, O is a finite set of ground actions built from A, $I \subseteq A$ represents the *initial state*, and $G \subseteq A$ represents the *goal* of the problem. The *application* of an action a to a state s is possible if and only if $pre(a) \subseteq s$ and the resulting state is defined by $s' = (s \setminus del(a)) \cup add(a)$. A *solution plan* is a sequence of actions $\langle a_1, \ldots, a_n\rangle$ such that for $s_0 = I$ and for all $i \in \{1, \ldots, n\}$, the intermediate states defined by $s_i = (s_{i-1} \setminus del(a_i)) \cup add(a_i)$ are such that $pre(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$. This simple STRIPS model has been enriched in many ways through the evolution of PDDL. However, the objective in the design of YAHSP2 is to consider the combinatorial difficulty of finding a solution plan only, and thus we stick to the basic STRIPS model. Action costs and durations are simply ignored, a temporal plan being obtained by a deordering of a valid sequential plan.

The *lookahead strategy* implemented in the first version of YAHSP has been described in (Vidal 2004). Briefly, the idea is to produce in polynomial time a sequence of actions that hopefully can bring search closer to a goal state, and to introduce this state in the open list of a best-first search algorithm just as if it was a normal state. To this end, relaxed plans (Hoffmann and Nebel 2001) which are often of high quality are used in YAHSP to compute such a sequence. This is performed by a simple algorithm which tries to apply as much actions as possible from a relaxed plan to the state for which it has been computed. When no more action can be applied, a simple *repair strategy* tries to replace an action of the relaxed plan by another one, taken from the global set of actions, which can be applied and produces an unsatisfied precondition of another action in the relaxed plan. The idea of producing lookahead plans and states has been recently enriched, for example by the computation of low-conflicts relaxed plans and a repair strategy based on insertion instead of replacement (Baier and Botea 2009), or by computing lookahead plans in a different way than extracting them from relaxed plans, using sophisticated techniques such as landmarks and causal chains (Lipovetzky and Geffner 2011).

YAHSP2: The Algorithms

In the design of the second version of the YAHSP planner, we took the opposite direction: instead of augmenting the techniques and components used inside the planner, we simplified its design and removed many unnecessary steps, following in that the KISS principle: “Keep It Simple, Stupid”. The motivations behind this work were first to implement a planner that could be easy to maintain and to embed into a wider system such as DAE_X, and second to better understand what makes YAHSP an efficient planner. Indeed, if some ideas were sometimes useful on the small set of benchmarks available when YAHSP was written, experiments on the much larger set of benchmarks now available changes the picture. The implementation, with respect to the version described in (Vidal 2004), has been modified and simplified in the following main ways:

- The relaxed plans used to compute lookahead plans are not any more computed from relaxed planning graphs. We found more convenient and easy to extract relaxed plans directly from the computation of a critical path heuristic such as h^{add} or h^1: all what is needed is a cost associated to each action. This has the advantage to avoid the need of complex data structures to build planning graphs, and considerably simplifies the algorithm.
- The heuristic value of states is no longer the length of relaxed plans, but the h^{add} value of the goal set. Among several variants that we have experimented, we found that using h^{add} for both evaluating states and extracting lookahead plans was a good strategy.
- Some refinements introduced in YAHSP are abandoned, due to their lack of robustness on the whole set of benchmarks. Among them are helpful actions first introduced in FF and used in YAHSP to define a lexicographic order on the nodes to be expanded (always preferring nodes coming from the application of an helpful action). Although some recent experiments show that they may be of interest (Richter and Helmert 2009), their use in YAHSP finally does not reveal to be that interesting. Also, goal-preferred actions (actions that do not delete a goal) which were used to compute twice a relaxed planning graph: the first one with goal-preferred actions only, and the second one with all actions of the problem in case of a failure in reaching the goals, are not used any more.

The simplified design of YAHSP2 allows us to completely describe the algorithms, which are implemented in around 450 lines of C code. The prerequisites are a parsing and grounding process (without any complex preprocessing such as mutex, landmarks, etc.), and a few helpers to easily access some data (in particular, the list of actions which consume, add and delete an atom are precomputed). States are implemented with bit vectors such that checking the presence of an atom in a state is performed in constant time. The open and closed lists are represented with red-black trees. Nodes of the search tree are tuples $n = \langle s, p, t, l, f, a\rangle$ where s is a state, p is the parent node of n, t is the sequence of actions (a single action for a classical transition, a sequence for lookahead states) yielding n from p, l is the length of

Algorithm 1: plan-search

```
input  : a planning problem $\Pi = \langle A, O, I, G \rangle$ and a weight
         $\omega$ for the heuristic function
output : a plan if search succeeds, $\bot$ otherwise
$open \leftarrow closed \leftarrow \emptyset$
create a new node $n$:
    $n.state \leftarrow I$
    $n.parent \leftarrow \bot$
    $n.steps \leftarrow \langle\rangle$
    $n.length \leftarrow 0$
$n' \leftarrow$ compute-node$(\Pi, \omega, n, open, closed)$
if $n' \neq \bot$ then return extract-plan$(n')$
else
    while $open \neq \emptyset$ do
        $n \leftarrow \arg\min_{n \in open} n.heuristic$
        $open \leftarrow open \setminus \{n\}$
        foreach $a \in n.applicable$ do
            create a new node $n'$:
                $n'.state \leftarrow (n.state \setminus del(a)) \cup add(a)$
                $n'.parent \leftarrow n$
                $n'.steps \leftarrow \langle a \rangle$
                $n'.length \leftarrow n.length + 1$
            $n'' \leftarrow$ compute-node$(\Pi, \omega, n', open, closed)$
            if $n'' \neq \bot$ then return extract-plan$(n'')$
    return $\bot$
```

Algorithm 2: compute-node

```
input  : a planning problem $\Pi = \langle A, O, I, G \rangle$, a weight $\omega$
         for the heuristic function, a node $n$, the open and
         closed lists
output : a goal node if search succeeds, $\bot$ otherwise; open
         and closed are updated
if $\exists\, n' \in closed \,|\, n'.state = n.state$ then return $\bot$
else
    $closed \leftarrow closed \cup \{n\}$
    $\langle cost, app \rangle \leftarrow$ compute-hadd$(\Pi, n.state)$
    $gcost \leftarrow \Sigma_{g \in G}\, cost[g]$
    if $gcost = 0$ then return $n$
    else if $gcost = \infty$ then return $\bot$
    else
        $n.applicable \leftarrow app$
        $n.heuristic \leftarrow n.length + \omega \times gcost$
        $open \leftarrow open \cup \{n\}$
        $\langle state, plan \rangle \leftarrow$ lookahead$(\Pi, n.state, cost)$
        create a new node $n'$:
            $n'.state \leftarrow state$
            $n'.parent \leftarrow n$
            $n'.steps \leftarrow plan$
            $n'.length \leftarrow n.length + \text{length}(plan)$
        return compute-node$(\Pi, \omega, n', open, closed)$
```

the plan reaching n from the initial state, f is the numerical heuristic evaluation of s and a is the set of actions applicable in s. The notations $n.state$, $n.parent$, $n.steps$, $n.length$, $n.heuristic$ and $n.applicable$ refer to s, p, t, l, f and a respectively. The operator $\oplus$ concatenates two sequences.

Algorithm 1 (`plan-search`) constitutes the core of the best-first search algorithm (a weighted-A* here). The first call to `compute-node` allows to find a solution to the problem without search, by recursive calls to the lookahead process. Nodes are extracted from the open list following their heuristic evaluation and are expanded with the applicable actions (already computed and stored in nodes inserted into the open list), and a solution plan is returned as soon as possible. In the version submitted to the 7th IPC, search is pursued in order to improve the solution, with pruning of partial plans whose quality is lower than that of the best plan found so far. Also, the weight ω is set to 3.

Algorithm 2 (`compute-node`) first performs duplicate state detection, even if the quality (length, cost or makespan) of the plan which yields such a state is improved; as we deliberately avoid optimization. It then computes the heuristic, checks if the goal is obtained or contrarily cannot be reached, and updates the node with the heuristic and the applicable actions given by `compute-hadd`. The node is then stored in the open list and a lookahead state/plan is computed by a call to `lookahead`. A new node corresponding to the lookahead state is then created and `compute-node` is recursively called. Recursion is stopped when a goal state, a duplicate state or a dead-end state is reached.

Algorithm 3 (`compute-hadd`) computes h^{add} and returns a vector of costs for all atoms and actions, as well as actions applicable in the state for which h^{add} is computed obtained as a side-effect. Several ways are possible to compute h^{add}, e.g. by mutually recursive functions triggered by the updates; the one shown here has the advantage to be very simple and efficient, even if it looks laborious at first sight because of multiple iterations over the whole set of actions.

Algorithm 4 (`lookahead`) computes a lookahead state/plan from a relaxed plan given by a call to `extract-relaxed-plan`. Once a first applicable action of the relaxed plan is encountered, it is appended to the lookahead plan and the lookahead state is updated. A second applicable action is then sought from the beginning of the relaxed plan, and so on. When no applicable action is found, a repair strategy tries to find an applicable action of minimum cost from the whole set of actions, in order to replace an action of the relaxed plan which produces an unsatisfied precondition of another action of the relaxed plan, and the process loops.

Algorithm 5 (`extract-relaxed-plan`) computes a relaxed plan from a vector of action costs. A sequence of goals to produce is maintained, starting from the goals of the problem. The first one is extracted, and an action which produces it with the lowest cost is selected and stored in the relaxed plan. Its preconditions are appended to the sequence of goals, and the process loops until the sequence of goals is empty. An atom already satisfied, i.e. produced by an action of the relaxed plan, is not considered twice. The relaxed plan is finally sorted before being returned, by increasing costs first, and for equal costs by trying to order first an action which does not delete a precondition of the next action.

Algorithm 3: compute-hadd

```
input  : a planning problem $\Pi = \langle A, O, I, G \rangle$ and a state $s$
output : the vector of action and atom costs and the set of
         actions applicable in $s$

foreach $a \in O$ do
    $cost[a] \leftarrow \infty$
    $update[a] \leftarrow (pre(a) = \emptyset)$
foreach $p \in A$ do
    if $p \in s$ then
        $cost[p] \leftarrow 0$
        foreach $a \in O \mid p \in pre(a)$ do
            $update[a] \leftarrow true$
    else $cost[p] \leftarrow \infty$
$app \leftarrow \emptyset$
$loop \leftarrow true$
while $loop$ do
    $loop \leftarrow false$
    foreach $a \in O$ do
        if $update[a]$ then
            $update[a] \leftarrow false$
            $c \leftarrow \Sigma_{p \in pre(a)}\ cost[p]$
            if $c < cost[a]$ then
                $cost[a] \leftarrow c$
                if $c = 0$ then $app \leftarrow app \cup \{a\}$
                foreach $p \in add(a)$ do
                    if $c + 1 < cost[p]$ then
                        $cost[p] \leftarrow c + 1$
                        foreach $a \in O \mid p \in pre(a)$ do
                            $loop \leftarrow true$
                            $update[a] \leftarrow true$
return $\langle cost, app \rangle$
```

Algorithm 4: lookahead

```
input  : a planning problem $\Pi = \langle A, O, I, G \rangle$, a state $s$, and
         a vector of action costs $cost$
output : a lookahead state and a lookahead plan

$plan \leftarrow \langle\rangle$
$rplan \leftarrow$ extract-relaxed-plan($\Pi, s, cost$)
                                   // with $rplan = \langle a_1, \ldots, a_n \rangle$
$loop \leftarrow true$
while $loop$ do
    $loop \leftarrow false$
    if $\exists\, i \in \{1, \ldots, n\} \mid pre(a_i) \subseteq s$ then
        $loop \leftarrow true$
        $i \leftarrow \min(i \in \{1, \ldots, n\} \mid pre(a_i) \subseteq s)$
        $s \leftarrow (s \setminus del(a_i)) \cup add(a_i)$
        $plan \leftarrow plan \oplus \langle a_i \rangle$
        $rplan \leftarrow \langle a_1 \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \rangle$
    else
        $i \leftarrow j \leftarrow 1$
        while $\neg loop \wedge i \leq n$ do
            while $\neg loop \wedge j \leq n$ do
                if $i \neq j \wedge add(a_i) \cap pre(a_j) \neq \emptyset$ then
                    $candidates \leftarrow \{a \in O \mid pre(a) \subseteq s$
                        $\wedge\, add(a_i) \cap pre(a_j) \cap add(a) \neq \emptyset\}$
                    if $candidates \neq \emptyset$ then
                        $loop \leftarrow true$
                        $a \leftarrow \arg\min_{a \in candidates} cost[a]$
                        $rplan \leftarrow \langle a_1 \ldots, a_{i-1}, a, a_{i+1},$
                            $\ldots, a_n \rangle$
                $j \leftarrow j + 1$
            $i \leftarrow i + 1$
return $\langle s, plan \rangle$
```

Algorithm 5: extract-relaxed-plan

```
input  : a planning problem $\Pi = \langle A, O, I, G \rangle$, a state $s$, and
         a vector of action costs $cost$
output : a relaxed plan for $\Pi$

$rplan \leftarrow \langle\rangle$
$goals \leftarrow \langle g \mid g \in G \rangle$
$satisfied \leftarrow s$
while $goals \neq \emptyset$ do
    $g \leftarrow$ pop-first($goals$)
    if $g \notin satisfied$ then
        $satisfied \leftarrow satisfied \cup \{g\}$
        $a \leftarrow \arg\min_{a \in O \mid g \in add(a)} cost[a]$
        if $a \notin rplan$ then
            $rplan \leftarrow rplan \oplus \langle a \rangle$
            $goals \leftarrow goals \oplus \langle p \mid p \in pre(a) \rangle$
sort $rplan = \langle a_1, \ldots, a_n \rangle$: $\forall a_i, a_j \in rplan \mid i < j$,
    $cost[a_i] < cost[a_j] \vee (cost[a_i] = cost[a_j] \wedge$
    $(del(a_i) \cap pre(a_j) = \emptyset$ if possible))
return $rplan$
```

Experiments

We performed extensive experiments on the whole set of benchmarks, from the 1st to the 6th IPC, that YAHSP2 can handle (i.e. without ADL and numerical domains). The objective of the experiments is to demonstrate that a simple heuristic search planner with a lookahead strategy is competitive with the state-of-the-art in terms of number of solved problems and running time. All experiments are performed on an Intel Xeon X5670 running at 2.93GHz with 4GB of memory and a timeout of 30 minutes.

Sequential Planning

Seven planners are compared on 1534 sequential planning problems. Costs have been removed from domains of the 6th IPC, in order to run planners that do not accept them such as FF and LPG-td. The planners are FF (Hoffmann and Nebel 2001), LAMA (Richter and Westphal 2010), LPG-td (Gerevini, Saetti, and Serina 2003), Mp (Rintanen 2010), SGPlan6 (Chen, Wah, and Hsu 2006), YAHSP version 1 with two different settings: $\mathrm{Y1}_{\mathrm{lbfs}}$ similar to YAHSP2 and $\mathrm{Y1}_{\mathrm{lobfs}}$ with the "optimistic" strategy (i.e. expanding first nodes coming from the application of an helpful action), and YAHSP2. Most of these planners have been awarded at previous IPCs, except the recent Mp and YAHSP2 planners. We included Mp as it is the first SAT-based planner competitive with other types of satisficing planners (Rintanen 2010).

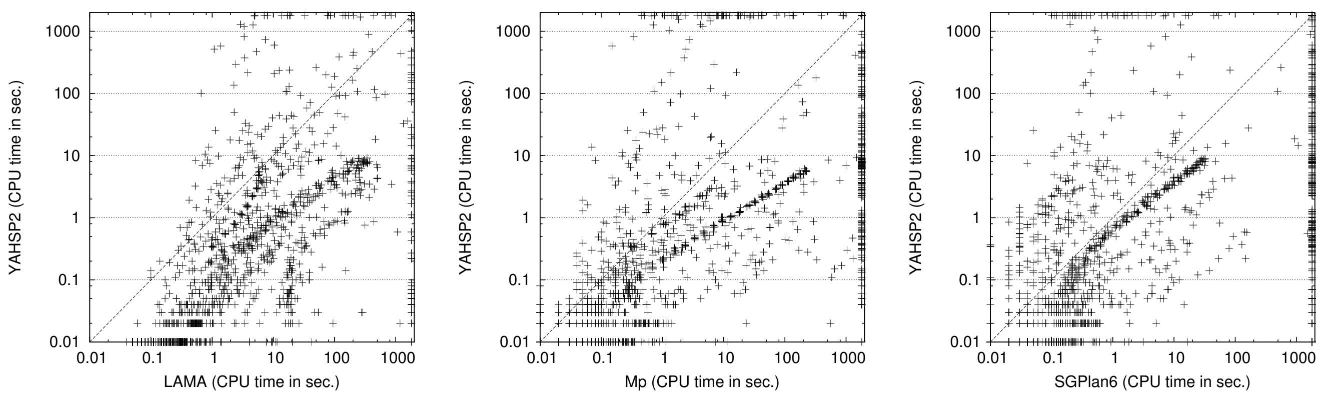

Figure 3: Comparison of the total running time for the three best sequential planners (except YAHSP1) versus YAHSP2.

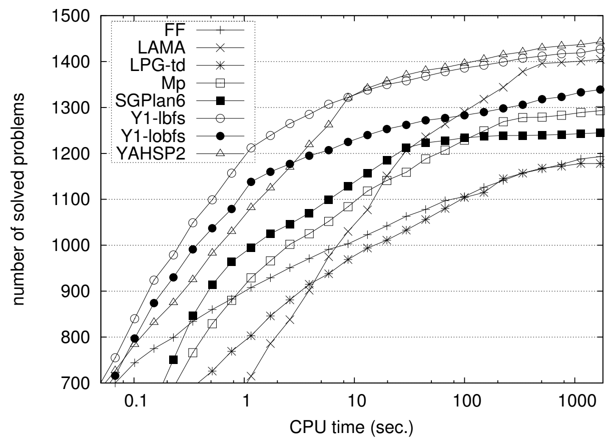

Figure 1: Cumulated number of solved problems for sequential planners in function of the total running time.

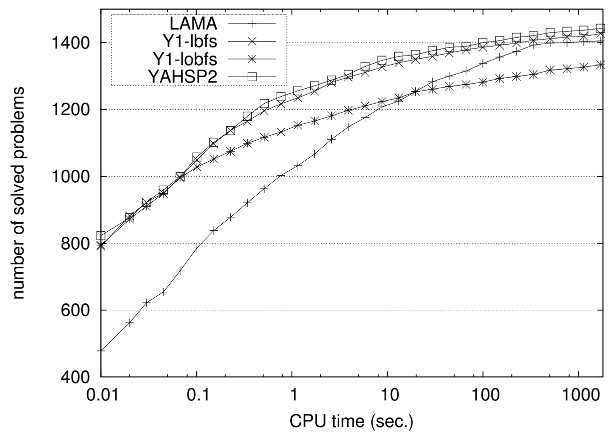

Figure 2: Cumulated number of solved problems in function of the search time for the four best sequential planners.

Figure 1 shows the cumulated number of solved problems in function of the total running time. For each CPU time t on the x axis, the corresponding value on the y axis gives the number of problems solved in under t seconds. YAHSP2 and Y1_{lbfs} clearly outperform the other planners. Y1_{lbfs} is a bit faster than YAHSP2 for problems solved in under 10 seconds, but YAHSP2 finally solves more problems. This mainly comes from the parser of YAHSP1, which has been better designed and is much more efficient than that of YAHSP2. The comparison with Y1_{lobfs} clearly confirms that giving priority to nodes coming from helpful actions was finally not a so good idea, in conjunction with the lookahead strategy. LAMA nearly reaches YAHSP2, solving 1405 problems (91.6%) with respect to 1444 (94.1%) for YAHSP2, but is significantly slower than YAHSP2. One reason is that it performs a heavy preprocessing step in order to translate to SAS+ and to compute landmarks, but Figure 2 which compares the search time only of the four best planners shows that search in LAMA is less efficient than in YAHSP2 and Y1_{lbfs}. It should be mentioned that although (Rintanen 2010) shows that Mp outperforms LAMA, this is probably due to the 300 seconds timeout which clearly disadvantages LAMA: on small runtimes it is the slowest among all planners compared here, but finally is in the top three. Figure 3 depicts scatter plot comparisons of the running time between YAHSP2 and the three other best planners (except YAHSP1), which are LAMA, Mp and SGPlan6. YAHSP2 very often outperforms them by several orders of magnitude. Finally, Table 1 shows the detail of the number of solved problems, over each IPC and each domain.

Temporal Planning

Four planners are compared on 664 temporal planning problems. The planners are LPG-td, SGPlan6, TFD (Eyerich, Mattmüller, and Röger 2009) and YAHSP2. The first three ones have been awarded at previous IPCs.

Figure 4 shows the cumulated number of solved problems in function of the total running time. YAHSP2 outperforms all planners, solving 594 problems (89.5%) against 434 problems (65.4%) for SGPlan6, the second best planner. SGPlan6 outperforms LPG-td that solves 403 problems (60.7%), which itself outperforms TFD that solves 287 problems (43.2%). Figure 5 depicts scatter plot comparisons of

IPC	domain	#pbs	#solved (difference with best)							
			FF	LAMA	LPG-td	Mp	SGPlan6	$Y1_{lbfs}$	$Y1_{lobfs}$	YAHSP2
1	grid	5	**5**	**5**	**5**	4 (1)	**5**	**5**	**5**	**5**
	gripper	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	logistics	35	**35**	**35**	29 (6)	22 (13)	**35**	**35**	**35**	31 (4)
	movie	30	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	mprime	35	34 (1)	**35**	**35**	**35**	33 (2)	**35**	**35**	**35**
	mystery	30	18 (4)	**22**	20 (2)	18 (4)	19 (3)	18 (4)	20 (2)	**22**
	total	155	142 (5)	**147**	139 (8)	129 (18)	142 (5)	143 (4)	145 (2)	143 (4)
	% solved		91.6%	**94.8%**	89.7%	83.2%	91.6%	92.3%	93.5%	92.3%
2	blocks	60	48 (12)	55 (5)	**60**	52 (8)	39 (21)	42 (18)	41 (19)	47 (13)
	miconic	150	**150**	**150**	**150**	**150**	**150**	**150**	**150**	**150**
	freecell	60	**60**	58 (2)	12 (48)	40 (20)	59 (1)	**60**	**60**	**60**
	logistics	198	197 (1)	196 (2)	**198**	178 (20)	**198**	**198**	**198**	**198**
	total	468	455 (4)	**459**	420 (39)	420 (39)	446 (13)	450 (9)	449 (10)	455 (4)
	% solved		97.2%	**98.1%**	89.7%	89.7%	95.3%	96.2%	95.9%	97.2%
3	depots	22	**22**	20 (2)	**22**	**22**	**22**	19 (3)	20 (2)	**22**
	driverlog	20	16 (4)	**20**	**20**	**20**	17 (3)	**20**	**20**	19 (1)
	freecell	20	**20**	**20**	3 (17)	11 (9)	19 (1)	**20**	**20**	**20**
	rovers	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	satellite	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	zenotravel	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	total	122	118 (3)	120 (1)	105 (16)	113 (8)	118 (3)	119 (2)	120 (1)	**121**
	% solved		96.7%	98.4%	86.1%	92.6%	96.7%	97.5%	98.4%	**99.2%**
4	airport	50	30 (16)	38 (8)	45 (1)	**46**	43 (3)	39 (7)	39 (7)	45 (1)
	pipesworld-notankage	50	36 (14)	44 (6)	43 (7)	36 (14)	0 (50)	**50**	48 (2)	44 (6)
	pipesworld-tankage	50	22 (27)	39 (10)	26 (23)	24 (25)	10 (39)	**49**	21 (28)	43 (6)
	promela-optical-telegraph	14	2 (12)	2 (12)	1 (13)	**14**	**14**	13 (1)	13 (1)	6 (8)
	promela-philosophers	29	14 (15)	13 (16)	2 (27)	**29**	**29**	**29**	5 (24)	**29**
	psr-small	50	42 (8)	**50**	48 (2)	**50**	**50**	**50**	47 (3)	**50**
	satellite-strips	36	**36**	34 (2)	**36**	32 (4)	35 (1)	**36**	**36**	**36**
	total	279	182 (84)	220 (46)	201 (65)	231 (35)	181 (85)	**266**	209 (57)	253 (13)
	% solved		65.2%	78.9%	72.0%	82.8%	64.9%	**95.3%**	74.9%	90.7%
5	openstacks	30	7 (23)	**30**	22 (8)	20 (10)	23 (7)	**30**	**30**	**30**
	pathways	30	10 (20)	28 (2)	**30**	**30**	**30**	20 (10)	26 (4)	29 (1)
	pipesworld	50	6 (44)	40 (10)	20 (30)	23 (27)	17 (33)	**50**	22 (28)	43 (7)
	rovers	40	16 (24)	**40**	30 (10)	**40**	30 (10)	**40**	**40**	**40**
	storage	30	18 (12)	19 (11)	**30**	**30**	**30**	25 (5)	21 (9)	18 (12)
	tpp	30	12 (18)	**30**	15 (15)	**30**	20 (10)	**30**	**30**	**30**
	trucks	30	4 (26)	13 (17)	5 (25)	**30**	6 (24)	11 (19)	14 (16)	16 (14)
	total	240	73 (133)	200 (6)	152 (54)	203 (3)	156 (50)	**206**	183 (23)	**206**
	% solved		30.4%	83.3%	63.3%	84.6%	65.0%	**85.8%**	76.2%	**85.8%**
6	cybersec	30	0 (30)	**30**	6 (24)	6 (24)	6 (24)	12 (18)	10 (20)	**30**
	elevators	30	**30**	**30**	25 (5)	**30**	**30**	**30**	**30**	**30**
	openstacks	30	**30**	**30**	**30**	15 (15)	27 (3)	**30**	**30**	**30**
	parcprinter	30	**30**	25 (5)	29 (1)	**30**	**30**	**30**	26 (4)	**30**
	pegsol	30	**30**	29 (1)	11 (19)	**30**	12 (18)	**30**	**30**	**30**
	scanalyzer	30	**30**	**30**	24 (6)	28 (2)	29 (1)	28 (2)	26 (4)	28 (2)
	sokoban	30	27 (2)	26 (3)	0 (29)	6 (23)	8 (21)	24 (5)	25 (4)	**29**
	transport	30	29 (1)	**30**	20 (10)	23 (7)	**30**	**30**	**30**	**30**
	woodworking	30	17 (13)	29 (1)	16 (14)	**30**	**30**	29 (1)	26 (4)	29 (1)
	total	270	223 (43)	259 (7)	161 (105)	198 (68)	202 (64)	243 (23)	233 (33)	**266**
	% solved		82.6%	95.9%	59.6%	73.3%	74.8%	90.0%	86.3%	**98.5%**
total		1534	1193 (251)	1405 (39)	1178 (266)	1294 (150)	1245 (199)	1427 (17)	1339 (105)	**1444**
% solved			77.8%	91.6%	76.8%	84.4%	81.2%	93.0%	87.3%	**94.1%**

Table 1: Number and percentage of solved problems in all sequential domains of the IPCs from 1998 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

the running time between YAHSP2 and the three other planners, and confirms that YAHSP2 has much better performances. The detail of the number of solved problems over each IPC and each domain can be found in Table 2.

YAHSP2-MT: A Multi-Threaded Planner

We now briefly describe YAHSP2-MT, a multi-threaded version of YAHSP2 which aims at benefiting from the computing power offered by multi-core processors with shared memory. A more detailed description can be found in (Vidal, Bordeaux, and Hamadi 2010).

The key idea is similar to that of KBFS (Felner, Kraus, and Korf 2003): always expanding first the best node of the open list, giving a maximum trust to the heuristic, may lead search to unpromising parts of the search space; while better parts could have been reached by expanding nodes ranked

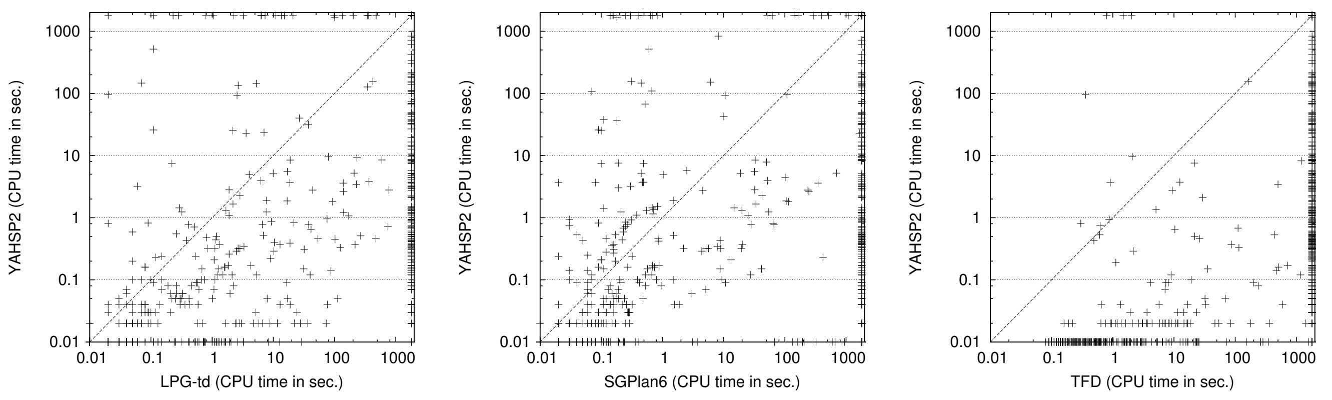

Figure 5: Comparison of the total running time for all temporal planners versus YAHSP2.

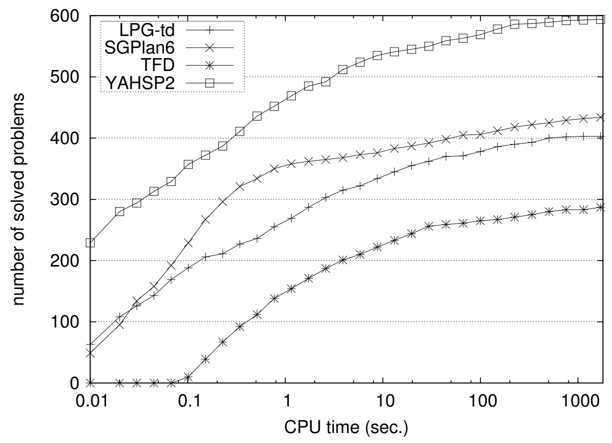

Figure 4: Cumulated number of solved problems for temporal planners.

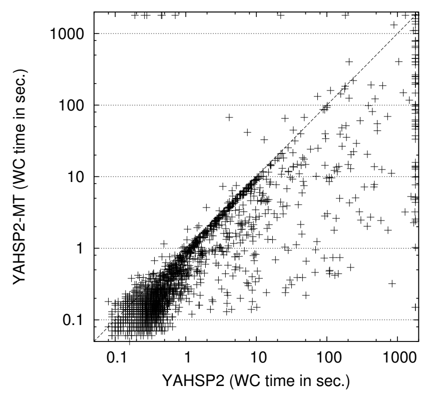

Figure 6: Comparison between the sequential version and the multi-threaded version with restarts of YAHSP2.

lower by the heuristic. KBFS expands the K best nodes of the open list, and then adds all their children to the open list. With the goal of avoiding as much as possible modifications to the existing YAHSP2 code, we simply start K threads that share the same open and closed list, expanding nodes in a concurrent way. This can be made very easily by inserting OpenMP directives between carefully selected lines of code. This simple strategy is used in conjunction with restarts triggered by limits on the number of evaluated nodes, where each restart increases the number of active threads. We also used a slightly different strategy than (Vidal, Bordeaux, and Hamadi 2010): two distinct open and closed lists are each attacked by half of the threads. The first half behave classically, whereas the second half runs an incomplete algorithm, pruning nodes which are obtained with the same number of actions and have the same heuristic value. Figure 6 compares the wall-clock time between YAHSP2 and YAHSP2-MT on a 12-core machine with 24GB of memory and a wall-clock timeout of 30 minutes, on the full set of 2198 problems. The restart strategy starts from 1 thread and goes up to 384 threads (128 for the version submitted to the 7th IPC). YAHSP2 solves 2038 problems (92.7%), while YAHSP2-MT solves 2082 problems (94.7%). We can see that very often, the multi-threaded version offers super-linear speedups. Furthermore, much less problems are solved faster by the sequential version than in previous tests (Vidal, Bordeaux, and Hamadi 2010), probably because a 4-core machine was used.

Conclusion

We described in this paper the new version of YAHSP, a heuristic search planner that uses a lookahead strategy. Its design has been led by an objective of simplicity, both in the algorithms and the source code, implying many changes with respect to the first version. The resulting planner outperforms state-of-the-art sequential and temporal planners in terms of cumulated number of solved problems and running time. We deliberately avoided analyzing plan quality, as the goal was to produce a fast planner easily embeddable into a wider system such as the DAE_X planner. Thus, we expect YAHSP2 to be outperformed by at least DAE_YAHSP at the 7th IPC. We also briefly described YAHSP2-MT, the multi-threaded version of YAHSP2 that aims at exploiting multi-core processors, which very often obtains super-linear speedups in comparison with the sequential version.

IPC	domain	#pbs	#solved (difference with best)			
			LPG-td	SGPlan6	TFD	YAHSP2
3	depots	22	**22**	21 (1)	2 (20)	**22**
	driverlog	20	**20**	18 (2)	10 (10)	19 (1)
	rovers	20	**20**	**20**	19 (1)	**20**
	satellite	20	**20**	**20**	**20**	**20**
	zenotravel	20	**20**	**20**	14 (6)	**20**
	total	102	**102**	99 (3)	65 (37)	101 (1)
	% solved		**100.0%**	97.1%	63.7%	99.0%
4	airport	50	42 (3)	43 (2)	10 (35)	**45**
	airport-timewindows	50	0 (46)	0 (46)	6 (40)	**46**
	pipesworld-notankage-deadlines	30	0 (30)	**30**	11 (19)	**30**
	pipesworld-notankage	50	43 (1)	0 (44)	20 (24)	**44**
	pipesworld-tankage	50	28 (15)	10 (33)	6 (37)	**43**
	satellite	36	**36**	35 (1)	7 (29)	**36**
	satellite-timewindows	36	0 (21)	0 (21)	3 (18)	**21**
	total	302	149 (116)	118 (147)	63 (202)	**265**
	% solved		49.3%	39.1%	20.9%	**87.7%**
5	openstacks	20	18 (2)	**20**	4 (16)	**20**
	storage	30	**30**	**30**	8 (22)	19 (11)
	trucks	30	24 (6)	24 (6)	18 (12)	**30**
	total	80	72 (2)	**74**	30 (44)	69 (5)
	% solved		90.0%	**92.5%**	37.5%	86.2%
6	crewplanning	30	11 (19)	**30**	29 (1)	**30**
	elevators	30	0 (30)	**30**	17 (13)	**30**
	openstacks	30	**30**	**30**	**30**	**30**
	parcprinter	30	20 (5)	**25**	13 (12)	18 (7)
	pegsol	30	17 (13)	18 (12)	28 (2)	**30**
	sokoban	30	2 (19)	10 (11)	12 (9)	**21**
	total	180	80 (79)	143 (16)	129 (30)	**159**
	% solved		44.4%	79.4%	71.7%	**88.3%**
total		664	403 (191)	434 (160)	287 (307)	**594**
% solved			60.7%	65.4%	43.2%	**89.5%**

Table 2: Number and percentage of solved problems in all temporal domains of the IPCs from 2002 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

Acknowledgments

This work has been supported by the French National Research Agency (ANR) through COSINUS program (project DESCARWIN n°ANR-09-COSI-002). Many thanks to my colleagues of the DAE$_{\text{YAHSP}}$ team for their enthusiasm and numerous insightful discussions: Johann Dréo, Pierre Savéant and Marc Schoenauer; as well as to Lucas Bordeaux and Youssef Hamadi for their multi-core expertise.

References

Bäckström, C. 1998. Computational aspects of reordering plans. *JAIR* 9:99–137.

Baier, J. A., and Botea, A. 2009. Improving planning performance using low-conflict relaxed plans. In *Proc. ICAPS*, 10–17.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proc. ICAPS*, 18–25.

Chen, Y.; Wah, B. W.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR* 26:323–369.

Dréo, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2011. Divide-and-evolve: the marriage of descartes and darwin. In *Booklet of the 7th IPC*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. ICAPS*, 130–137.

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: K-best-first search. *AMAI* 39(1-2):19–39.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in LPG. *JAIR* 20:239–290.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proc. ICAPS*.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS*, 273–280.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Rintanen, J. 2010. Heuristic planning with sat: Beyond uninformed depth-first search. In *Proc. Australasian Conf. on AI*, 415–424.

Teichteil-Königsbuch, F.; Kuter, U.; and Infantes, G. 2010. Incremental plan aggregation for generating policies in MDPs. In *Proc. AAMAS*, 1231–1238.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *Proc. SOCS*, 100–107.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proc. ICAPS*, 150–160.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *Proc. ICAPS*, 352–359.