Improving Local Search for Resource-Constrained Planning

Hootan Nakhost
University of Alberta
Edmonton, Canada
nakhost@ualberta.ca

Jörg Hoffmann
INRIA
Nancy, France
joerg.hoffmann@inria.fr

Martin Müller
University of Alberta
Edmonton, Canada
mmueller@ualberta.ca

Abstract

A ubiquitous feature of planning problems – problems involving the automatic generation of action sequences for attaining a given goal – is the need to economize limited resources such as fuel or money. While heuristic search, mostly based on standard algorithms such as A*, is currently the superior method for most varieties of planning, its ability to solve critically resource-constrained problems is limited: current planning heuristics are bad at dealing with this kind of structure. To address this, one can try to devise better heuristics. An alternative approach is to change the nature of the search instead. Local search has received some attention in planning, but not with a specific focus on how to deal with limited resources. We herein begin to fill this gap. We highlight the limitations of previous methods, and we devise a new improvement (smart restarts) to the local search method of a previously proposed planner (Arvand). Systematic experiments show how performance depends on problem structure and search parameters. In particular, we show that our new method can outperform previous planners by a large margin.

Introduction

Automated planning, also often referred to as *domain-independent planning*, is concerned with the development of tools that, given high-level descriptions of an initial state, a goal, and a set of available actions, generate a plan: a schedule of actions leading from the initial state to the goal. The planning problem – deciding whether or not there exists a plan – is **PSPACE**-hard even in its simplest (but reasonably expressive) formalizations. Hence we cannot hope to obtain good performance on all possible problem instances. What we can try to do is to develop algorithms that can deal well with particular kinds of *problem structure*, i.e., with subclasses of instances sharing certain properties.

Problem structure that tends to appear in practical applications is, of course, particularly important. We herein are concerned with one such structure: *the need to economize limited resources*. This need is ubiquitous in many of the classical applications of planning, involving the control of autonomous agents requiring to economize resources such as energy, fuel, money, and/or time. Clearly, such economization is the more difficult the more *constrained* the problem is: if C denotes the ratio between the amount of available resources vs. the minimum amount required, then the problem becomes intuively "harder" as C approaches 1.[1]

Although planning with resources is a long-standing topic, e.g. (Koehler 1998; Haslum and Geffner 2001; Long and Fox 2003; Hoffmann 2003), the progress made in solving such problems is limited. Specifically, in almost all known varieties of planning, today heuristic search based on standard algorithms such as A* and greedy best-first search is by far the superior method. The key to success has been the development of heuristic functions that are very informative in many of the traditional benchmark domains, e.g. (Bonet and Geffner 2001; Hoffmann and Nebel 2001; Richter, Helmert, and Westphal 2008; Helmert and Domshlak 2009). Now, almost all of these successful heuristics are what has been termed *relaxation heuristics* (Helmert and Domshlak 2009): they compute estimates relative to a relaxed planning problem in which all "negative effects" of the operators, which can obstruct the rest of the plan, are ignored. Resource consumption falls into this category. Hence *almost all existing heuristic functions for planning completely ignore the need to economize resources*.

It has been observed (Hoffmann et al. 2007) that some otherwise very successful planners perform very poorly when C is close to 1. Somewhat surprisingly, this is the *only* thing known about how planning systems react when changing C, and no attempt has been made to improve on this situation. So what can we do about this dilemma?

One option is to try to develop better planning heuristics. An alternative option is to modify the search. Local search in particular might be better suited than A* for this kind of situation where the heuristic is very error-prone. To motivate this thought with a well-known example, consider random CNF formulas from the phase transition region (Cheeseman, Kanefsky, and Taylor 1991), and the heuristic that counts the number of unsatisfied clauses. Certainly, no-one would entertain the idea of solving this with A*. Hill-climbing, on the other hand, is very successful (Selman, Levesque, and Mitchell 1992; Wei, Li, and Zhang 2008).

Local search has received some attention in planning, e.g.

[1]Obviously, this intuitive "hardness" is connected to the computational complexity of (exact or approximate) optimization in the underlying domain. Note, however, that in domain-*independent* planning there is no general connection between C and computational complexity. It is trivial to construct example domains that are **NP**-hard despite large C, or that are in **P** despite $C = 1$.

(Hoffmann and Nebel 2001; Gerevini, Saetti, and Serina 2003; Nakhost and Müller 2009), but not with a specific focus on how to deal with limited resources. We herein begin to fill this gap. The main empirical basis of our investigation is the simple transportation domain of Hoffmann et al (2007), called *NoMystery* here, whose generator allows to control (fuel) constrainedness C. We can hence control precisely the "problem structure" we are interested in, enabling us to draw clear-cut conclusions.

We first highlight the limitations of previous methods, updating the results of Hoffmann et al (2007) with new planners that appeared in the meantime. As it turns out, the performance with C close to 1 has not improved at all. To address this, we introduce an improvement to the local search planner *Arvand* (Nakhost and Müller 2009). Arvand appears especially suitable since its basic search strategy relies on fast random walks, enabling the planner to explore a larger fragment of the search space even if the heuristic function – as is typically the case in planning – is very costly to compute. The new technique in *ArvandSR* are *smart restarts*: it maintains a pool of the p "best" previous search episodes, that ultimately reached the smallest heuristic values), and restarts from a random state along those search trajectories. Thus ArvandSR balances exploitation of previous runs against the risk of repeating previous mistakes.

We run large-scale experiments with *ArvandSR* in NoMystery, modifying three relevant parameters: C, the constrainedness of resources; p, the size of the pool for smart restarts; and w, a parameter controlling the trade-off between Arvand's two search strategies MDA and MHA, which were previously strictly separate. We plot performance as a function of (combinations of) these parameters. In particular, smart restarts can be very useful. They do not have much effect when resources are aplenty. However, as C approaches 1, they become more useful. For $C = 1.0$ and $C = 1.1$, *ArvandSR* drastically outperforms all previous planners. Cross-checking these results on a subset of IPC benchmarks with a resources or puzzle nature, we determine that, also there, smart restarts rarely ever hurt, and sometimes help significantly.

In the next section, we define the planning problem we consider, and give an outline of the literature on resource constrained planning. We then briefly describe the planners considered herrein, and highlight their difficulties as C approaches 1. We explain smart restarts, and give our experimental findings, before concluding the paper.

Resource Constrained Planning

We base our work on the simple wide-spread planning formalism of propositional STRIPS planning, short *STRIPS*. A *planning task* is a tuple (P, I, G, A) where P is a set of propositions, $I \subseteq P$ is the *initial state*, $G \subseteq P$ is the *goal*, and A is a set of *actions*. Each action $a \in A$ is a triple (pre_a, add_a, del_a) of subsets of P, referred to as a's *precondition*, *add list*, and *delete list* respectively; we assume that $add_a \cap del_a = \emptyset$. A *state* s is a subset of P, interpreted as those propositions that are true in s. An action a is *applicable* to s if $pre_a \subseteq s$; the result of executing a is $(s \setminus del_a) \cup add_a$. A *plan* is a sequence of actions in whose iterative execution from I all actions are applicable, and that ends in a result state s so that $G \subseteq s$. For example, a driving action in a transportation domain could be $a = (\{\text{at-A}\}, \{\text{at-B}\}, \{\text{at-A}\})$, I could be $\{\text{at-A}\}$, and G could be $\{\text{at-B}\}$. The sequence $\langle a \rangle$ is then a plan.

What is meant by the term "resource-constrained planning"? At the intuitive level, resources may come in many forms (ranging from fuel and time over robot arms to the individual agents in RoboCup Rescue). Herein, we focus exclusively on the classical meaning of "resource" in the sense of "a quantity that is required for, and consumed by, executing some of the actions". This kind of planning with resources has been formalized, e.g. by Haslum and Geffner (2001). STRIP planning tasks are extended with a set R of resource identifiers as well as functions $i : R \mapsto \mathbb{Q}_{\geq 0}$ and $u : A \times R \mapsto \mathbb{Q}_{\geq 0}$. Here, $i(r)$ gives the initial amount available for each resource $r \in R$, and $u(a, r)$ is the amount of r consumed when executing action a.[2] Sufficient availability of resources becomes an additional condition for action applicability. If $R = \{r\}$ is a singleton, we define *resource constrainedness* $C := \frac{i(r)}{M}$ where M denotes the minimal resource consumption in any plan.[3]

In our implementation, we assume that i and u map into $\mathbb{N}_{\geq 0}$ instead of $\mathbb{Q}_{\geq 0}$. The planning tasks are then described in propositional STRIPS by a straightforward encoding of effects and preconditions on numbers between 0 and $max_{r \in R} i(r)$. The motivation is technical: *ArvandSR* currently cannot handle numeric variables.

As outlined in the introduction, planning with resources has a tradition in planning. To some extent, this is reflected in the benchmarks used in the biennial International Planning Competitions (IPC). In IPC'98, the domains *Mystery* and *Mprime* encoded transportation with fuel consumption.[4] In IPC'08, *Trucks* features time as a resource. Actions take time and there are strict delivery deadlines. In IPC'02, *Satellite* features fuel consumption as a resource; however, the STRIPS version of the domain omits resources so we do not run the domain here. Several other IPC domains feature resource consumption, imposing it not as a hard constraint but only as an optimization criterion. Satisficing planners may choose to ignore it, and most often do so. One domain has a re-fuelling action, which we disallow for the sake of simplicity.

IPC domains are widely used for testing, which is in principle good because it provides a common test base. However, such tests are not suitable for understanding resource constrainedness, i.e., the kind of problem structure we are interested in. The simple reason is that C is not a controlled

[2]Haslum and Geffner (2001) also allow "renewable" resources, like space in a truck, that are only temporarily consumed. We do not consider this case here.

[3]When there are multiple resources, the definition of C is a little tricky because there is no one aggregation method for resource values that is adequate across all possible domains. A simple canonic aggregation method that would make sense is addition. In the NoMystery domain, the only resource is fuel.

[4]Absurd predicate names were chosen so as to disguise this nature from planner developers – hence the name "Mystery".

quantity in the IPC benchmarks. In many cases, C is not even known, and much less do the IPC instances provide a range of instances with different values of C, keeping all other settings the same. We hence use, in most of our experiments, the aforementioned *NoMystery* domain of Hoffmann et al (2007). There, a truck moves in a weighted graph; a set of packages must be transported between nodes; actions move along edges, and load/unload packages; each move consumes the edge weight in fuel. In brief, NoMystery is a straightforward transportation problem similar to the ones contained in many IPC benchmarks. Its key feature is that it comes with a random generator allowing to control C. The generator creates a random connected undirected graph with n nodes, and it adds k packages with random origins and destinations. The edge weights are uniformly drawn between 1 and 25. A domain-specific branch-and-bound procedure computes the minimum required amount of fuel,[5] M, and the initial fuel supply is set to $\lfloor C * M \rfloor$, where $C \geq 1$ is a (float) input parameter of the generator. Our experiments fix n and k and let C range in order to be able to draw clear-cut conclusions about planner performance when resources become more constrained.

Current Algorithms and Limitations

A major performance breakthrough for satisficing planning – by many orders of magnitude in most benchmarks, and by a reduction from exponential to low-order polynomial behavior in some – was achieved around the year 2000. Instrumental for this was the development of what Helmert and Domshlak (2009) term *relaxation heuristics*: heuristics that estimate goal distance relative to a relaxed planning problem in which all delete lists are ignored. This idea was first voiced by Bonet and Geffner (2001). It underlies FF (Hoffmann and Nebel 2001) which outperformed all other automatic planners in IPC'00. The same is true of the best performer at IPC'02, LPG (Gerevini, Saetti, and Serina 2003), and at IPC'04 and IPC'06, SGPlan.[6] LAMA (Richter, Helmert, and Westphal 2008), the best performer in IPC'08, uses FF's heuristic plus a new landmarks-based relaxation heuristic.

As for optimal planning, while this expectedly is generally much less performant than satisficing planning, recently the invention of an admissible heuristic called *LM-cut* (Helmert and Domshlak 2009) yielded a major advance. LM-cut is also a relaxation heuristic.

Despite their enormous successes on IPC benchmarks, and despite the fact that resource-constrained planning has been a relevant topic for a long time, none of these successful planners is well suited for solving critically resource-constrained problems. This was observed for FF and LPG by Hoffmann et al (2007). We confirm this here for all the planners listed above.

We created 5 different NoMystery instances with $n = 12$ nodes in the graph and $m = 12$ packages. Their minimal fuel consumption was computed by the generator. Seven values for C were tested: $\{1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 2.0\}$, leading to a total of 35 test cases, by setting the initial amount of fuel to C times the minimum amount in each instance. This setup assures that results across different values of C are exclusively due to the level of resource constrainedness. The same 35 test instances underly all the NoMystery experiments reported throughout the rest of this paper. Running all the planners, we obtained the data depicted in Figure 1.[7] All the satisficing planners excel when C is large, but degrade to be rather useless as C approaches 1. The optimal planner, A* with LM-cut, solves one instance for each $C > 1$, but not for $C = 1$.

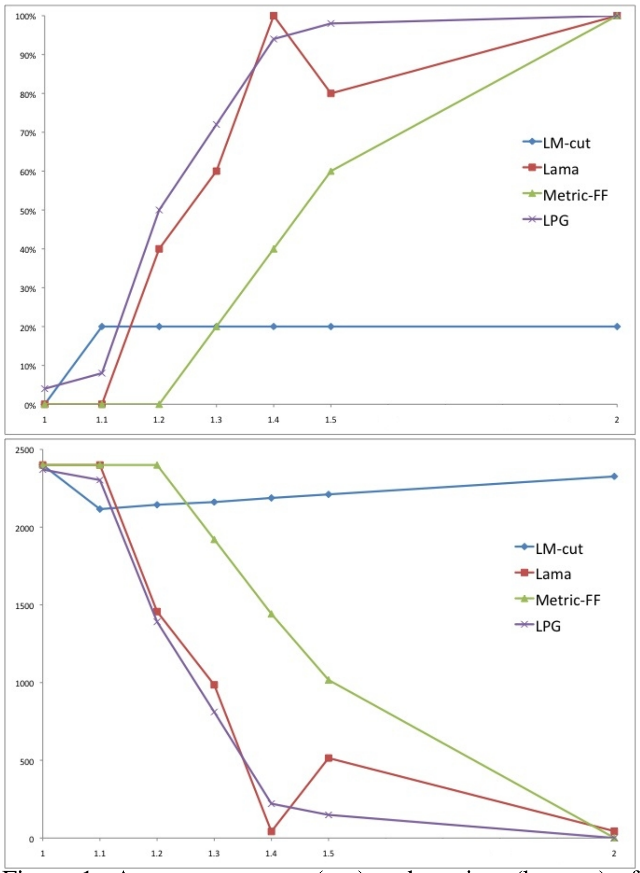


Figure 1: Average coverage (top) and runtime (bottom) of state-of-the-art planners when varying (only) resource constrainedness C. 5 test instances per value of C, only difference across C is initial fuel level. Memory limit 2 GB. Runtime limit 40 minutes which is inserted into the average computation in case of a time-out.

As previously hinted, there is a simple explanation to this profound lack of performance, pertaining to relaxation-based heuristics. These heuristics ignore the effect statements erasing the resource levels *prior* to execution of an

[5] Note here the difference between domain-*specific* and domain-*independent* solvers. Instances easy for the former (the generator) can already be very challenging for the latter (the planners we test).

[6] We do not run SGPlan because, as became known in 2008, it changes its configuration based on parsing IPC domain names.

[7] We also ran the "num2sat" planner of Hoffmann et al (2007), who reported this to beat FF and LPG when C is close to 1. However, those results were obtained on $n = m = 8$, and num2sat does not scale well with n and m. It does not solve a single instance in our experiment here, where $n = m = 12$.

action. The next action can hence reuse these same prior resource levels, rendering the heuristic completely unaware of resource consumption.

To address the observed difficulties, one can try to develop better-suited heuristics. We herein consider the alternative of developing better-suited search methods. All shown planners employ variants of A* or greedy best-first search – except LPG that uses a variant of local search.[8] Thus Figure 1 strongly suggests to consider local search in more depth: LPG is clearly the best planner for small C. In our work, we focus on the more recent local search planner Arvand.

Smart Restarts in ArvandSR

Arvand (Nakhost and Müller 2009) is a stochastic planner that mixes random sampling with heuristic evaluation. This combination enables Arvand to both benefit from good heuristic values and, to some extent, overcome misleading ones thanks to the speed of random sampling.

Starting at the initial state, at each "search step" Arvand choses its next state from a number of states sampled in a local neighborhood. Each sample is obtained by running a bounded random walk, i.e., a sequence of randomly selected actions. The sampled states are the endpoints of the walks. Arvand transitions to an endpoint whose heuristic value under FF's heuristic (Hoffmann and Nebel 2001), henceforth referred to as h^{FF}, is minimal (ties are broken randomly). Since this method computes heuristic values only at the endpoints, it can perform a large number of random walks even if computing FF's heuristic is costly (which it typically is). Arvand continues performing search steps in this fashion, until either a goal state is reached or the search gets stuck: either the heuristic value does not improve over several search steps, or all random walk endpoints s are dead-ends - either no actions are applicable, or $h^{FF}(s) = \infty$. If the search is stuck, it restarts from the initial state, beginning a new "search episode".

Nakhost and Müller (2009) devise two different methods, *Monte-carlo with Helpful Actions (MHA)* and *Monte-carlo Deadlock Avoidance (MDA)*, that bias action selection during the random walks based on prior experience. In MDA, actions that frequently appeared in walks that get stuck are penalized. In MHA, the bias rewards actions that are often considered as "helpful actions" by FF (Hoffmann and Nebel 2001). This is a subset of actions that are deemed relevant, and can be determined quickly as a byproduct of computing h^{FF}. The intuition formulated by Nakhost and Müller (2009) is that MDA is better on domains with dead-ends, while MHA is better on domains with large branching factors. Although it appears natural to combine both methods, Nakhost and Müller (2009) did not do that. We report on a straightforward combination in our experiments further below.

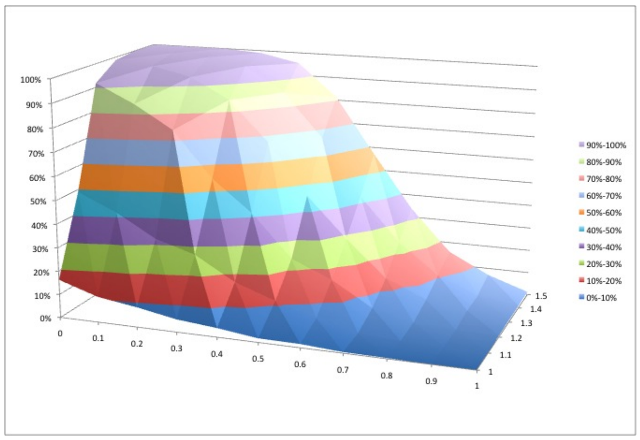
Our main modification of Arvand pertains to its restarting strategy. Consider Arvand's search after N restarts, i.e., after N search episodes all of which eventually got stuck. Arvand discards all these episodes, and restarts from scratch. This appears sensible because, after all, the previous episodes all appear to contain a wrong decision. Still, it does not entail that *all* their decisions were bad. In particular, in a resource-constrained planning task with C close to 1, it may easily happen that several good decisions at the start were finally turned into a failure by a single bad decision at the end. It is then a waste to discard the whole search episode. This motivates the new method of *smart restarts* that is implemented in ArvandSR.

The basic idea is simple: maintain a pool of "most promising episodes" performed so far. When an episode gets stuck, restart from a state visited in such an episode, instead of always restarting from the initial state. An episode is considered "more promising" than another one if it reaches a more promising state. As for states, not having a better handle on what is promising, we simply rely on the (same) heuristic function: state s is "more promising" than state s' iff $h^{FF}(s) < h^{FF}(s')$. Clearly, this definition inherits the difficulties of the heuristic – it does capture to some extent "how much work have we managed to accomplish", but it

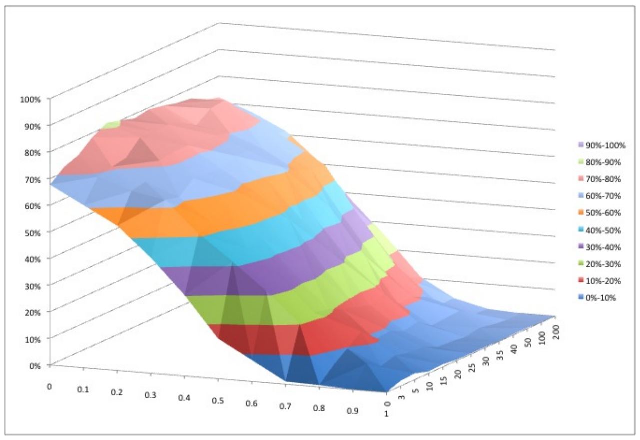
Algorithm 1 ArvandSR

```
Input Planning task $(P, I, G, A)$, pool size $p$, threshold $N$
Output A plan for $(P, I, G, A)$

  $s \leftarrow I$; $num\_restarts \leftarrow 0$; $\mathcal{P} \leftarrow \{\}$
  while TRUE do
    $\langle s_0, \dots, s_k \rangle \leftarrow ArvandSearchEpisode(s)$
    if $G \subseteq s_k$ then
      break
    end if
    $min \leftarrow argmin_{0 \leq i \leq k}(h^{FF}(s_i), i)$
    $add\_trace(\mathcal{P}, \langle s_0, \dots, s_{min} \rangle)$
    if $num\_restarts < N$ then
      $s \leftarrow s_0$ {restart from initial state}
    else
      $s \leftarrow get\_restart\_state(\mathcal{P})$ {restart from pool}
    end if
    $num\_restarts \leftarrow num\_restarts + 1$
  end while
  return the actions yielding $\langle s_0, \dots, s_k \rangle$

Procedure $add\_trace(\mathcal{P}, \langle s_0, \dots, s_k \rangle)$
  if $|\mathcal{P}| < p$ then
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle s_0, \dots, s_k \rangle\}$
  else
    if $h^{FF}(s_k) < max_{\langle s'_0, \dots, s'_l \rangle \in \mathcal{P}} h^{FF}(s'_l)$ then
      $\mathcal{P} \leftarrow \mathcal{P} \setminus \{argmax_{\langle s'_0, \dots, s'_l \rangle \in \mathcal{P}} h^{FF}(s'_l)\}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle s_0, \dots, s_k \rangle\}$
    end if
  end if

Procedure $get\_restart\_state(\mathcal{P})$
  $\langle s_0, \dots, s_k \rangle \leftarrow$ a random element of $\mathcal{P}$
  $s \leftarrow$ a random state on $\langle s_0, \dots, s_k \rangle$
  return $s$
```

[8]FF also performs a variant of Hill-climbing, however this variant is a hybrid with complete search, is not randomized, no restarts are made, and if it fails then FF defaults to greedy best-first search.



(a)



(b)

Figure 2: Average coverage of ArvandSR as a function (on the z-axis) of: the trade-off w between MDA and MHA; the pool size p for smart restarts; and resource constrainedness C. (a) scales w (x-axis) against C (y-axis) averaging over p, (b) scales w (x-axis) against p (y-axis) averaging over C. Detailed results for scaling p against C are in Figure 3.

does not actually take into account the resources. Hence the need to maintain *a pool* of episodes to restart from: this increases our chances to not repeat a previous mistake. This also makes apparent the need for a search parameter, namely the size of the pool, that we denote with p. If p is small, then the search is more greedy and concentrates on the hitherto best states (by $p = 0$ we will denote the original Arvand). If p is large, then the search is more explorative, and can solve some harder cases given more runtime. With limited time, more exploration can be detrimental since each individual state in the pool is visited less frequently. We will see later that these intuitions are reflected in our results on NoMystery.

Algorithm 1 gives pseudo-code detailing our changes to Arvand. We maintain a pool $\mathcal{P}$ that, given the definition of $add\_trace(\mathcal{P}, \langle s_0, \dots, s_k\rangle)$, contains up to p execution traces. The more tricky bit lies in which traces qualify for being included in $\mathcal{P}$. The procedure $ArvandSearchEpisode(s)$ executes one episode of (unchanged) Arvand. The trace $\langle s_0, \dots, s_k\rangle$ hence contains the state s_0 restarted from, followed by the endpoint states of the search steps. The expression $argmin_{0\leq i\leq k}(h^{FF}(s_i), i)$ denotes lexicographic minimization, i.e., s_{min} is taken to be the earliest state among those states with minimum heuristic value. The motivation for taking the earliest state is that, with growing i, the amount of resources consumed can only grow. Thus, s_{min} is the most promising state reached by the episode.

The procedure $get\_restart\_state(\mathcal{P})$ selects a random trace from $\mathcal{P}$. It considers not only the end state of the trace, but rather returns *any* state on it. This further decreases the chance to select a faulty state: with limited resources, states further up the trace have more resources so are potentially less dangerous. Note also that, if s_i is a state from which we can reach s_k with very small $h^{FF}(s_k)$, then this indicates that in s_i we have not wasted too many resources yet – otherwise we couldn't "complete the work" to this extent.

The rest of the algorithm should be self-explanatory, the only relevant additional detail being that the smart restarts method is not invoked until a threshold number N of search episodes has been completed. The motivation for this is that, if we begin smart restarting right away, then search is heavily biased towards the initial runs. In the experiments reported herein, we fixed $N = 50$.

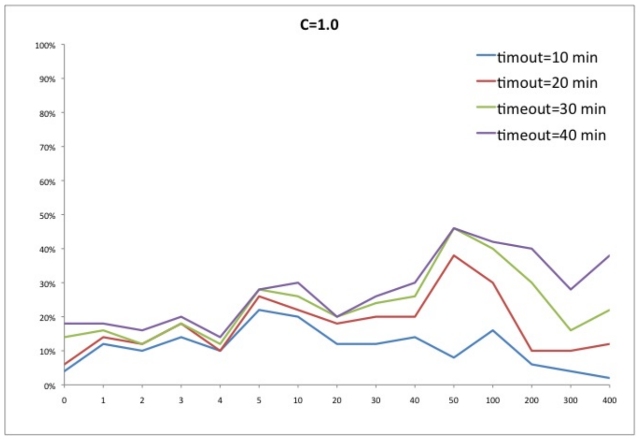
The reader might have the impression that some of our algorithm are hand-tailored to NoMystery. We emphasize that this is not the case. *The underlying intuitions apply to resource-constrained planning in general.* We also did not do any fine-tuning to NoMystery. We fixed N simply because, otherwise, there would have been too many free parameters in our experiments (as it is, we have 3 of them, see below). The setting $N = 50$ was obtained based on limited tests performed prior to the detailed runs reported below. The setting of all "old" search parameters of ArvandSR, i.e., of those parameters present already in the original version of Arvand (2009), were kept exactly the same.

Experiments

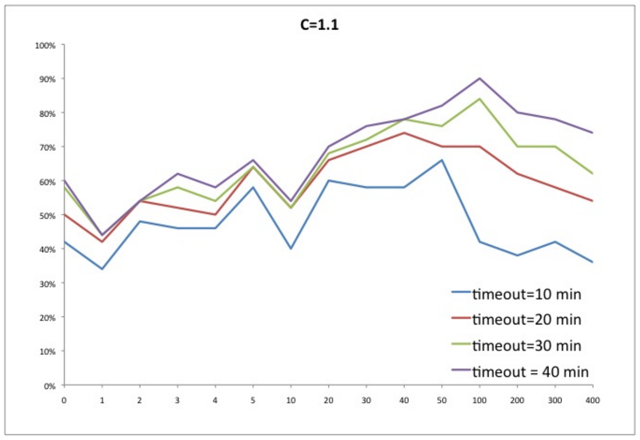
We ran large-scale experiments on NoMystery, as well as more limited tests on relevant IPC benchmarks. Experiments on NoMystery were run on a 3.00 GHz machine, and a 2.50 GHz machine is used for tests on IPC benchmarks. For all experiments the memory limit was set to 2 GB. The runtime cut-off varies depending on the experiment, and will be individually given below.

Problem Structure vs. Search Parameters

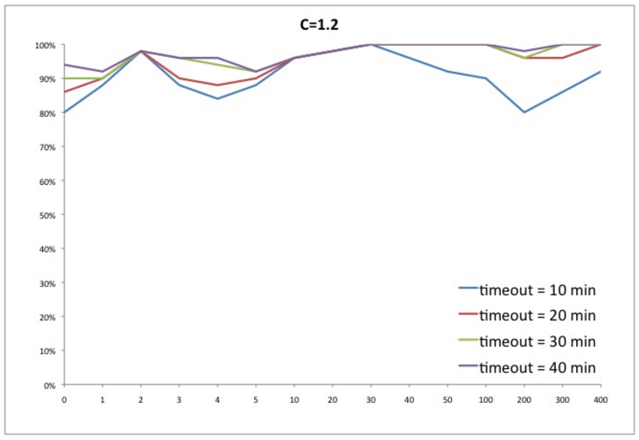
We now investigate the performance of ArvandSR as a function of problem structure and search parameters. Problem structure is captured in terms of resource constrainedness C, which motivates our choice of NoMystery because it is the only planning benchmark in existence whose generator allows to control that parameter. Throughout the section, we use exactly the same test instances as underly Figure 1. Recall that these instances each have 12 graph nodes and 12 packages; 5 random instances were generated, then they were ported to each $C \in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 2.0\}$ by



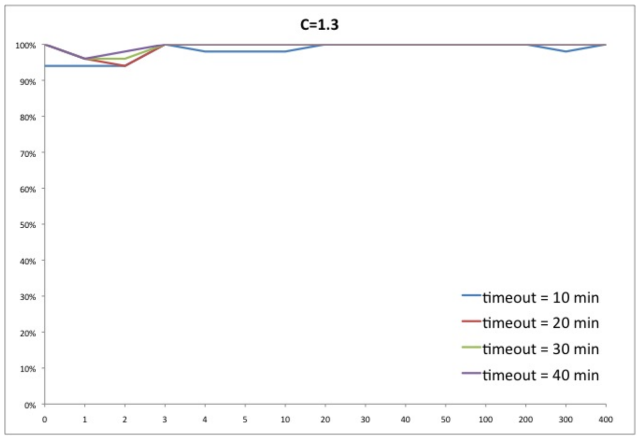
(a)



(b)



(c)



(d)

Figure 3: Coverage of ArvandSR when varying: the pool size p for smart restarts [shown on the x-axis in all of (a)–(d)]; the runtime cut-off [10, 20, 30, 40 minutes in each of (a)–(d)]; and resource constrainedness C [1.0 in (a), 1.1 in (b), 1.2 in (c), 1.3 in (d)].

setting the initial fuel level. We omit $C = 2.0$ here because, for Arvand, this yields the same behavior as $C = 1.5$.

We vary the pool size p for smart restarts. In addition, we experiment with a trade-off between MDA and MHA, governed by a parameter w with $0 \leq w \leq 1$. Every time Arvand samples an action within a random walk, we generate a random number $0 < r \leq 1$. If $r \leq w$, then we chose the next action according to MDA; if $r > w$, then we chose the next action according to MHA. Hence we have MHA for $w = 0$, MDA for $w = 1$, and a mixture of the two in between.

We run ArvandSR 10 times on each instance. Varying all three parameters simultaneously, the number of test runs required is large ($11 * 30 * 13$ instances, 10 runs each) so we set a relatively small time-out of 10 minutes. Data presentation is also tricky since in total it would require a 4-dimensional plot. Figure 2 shows two of the three possible 3-dimensional plots where two of the parameters are fixed, and results are averaged over the third one. The most apparent observation – apart from the known fact that small C is more challenging – is that MDA is completely useless here. Coverage increases monotonically the closer the w trade-off gets to MHA. Especially for small C, where dead-end states (too much fuel consumed) are omnipresent, this contradicts the intuition of Nakhost and Müller (2009) who suggested that MDA is more suitable in the presence of dead-ends. It is not clear to us what causes this behavior. We note that, as Figure 2 clearly shows, the behavior is independent from pool size and resources constrainedness.

One unresolved issue in *ArvandSR* is how to set the schedule for adapting the length of the random walks in this combination, since MDA and MHA use different schedules by default (Nakhost and Müller 2009). For NoMystery, pure MHA was always superior in our tests no matter which schedule for MDA was chosen.

It can be seen in Figure 2 (a) that, for small C, ArvandSR tends to be significantly more effective than the state-of-the-art planners tested in Figure 1. For $C = 1.0$, the best of those planners, LPG, has 4% coverage. In Figure 2 (a), the best coverage value for this is 16% at $w = 0$. For $C = 1.0$, these numbers are 8% for LPG, but 50% for *ArvandSR* despite the 4 times shorter time.

ArvandSR's coverage in Figure 2 (a) is averaged over all different values of pool size. Figure 2 (b) shows an interesting behavior across these p values. The behavior is most pronounced at $w = 0$ where the underlying search is most successful. Coverage improves from $p = 0$ up to around the middle of the p range, but then gets worse again, yielding a kind of sweet-spot behavior. That behavior is diluted, however, by averaging over C. Hence, in our next experiment, we consider this issue in more detail. We fix w to 0, try a broader range of p values, and vary the runtime cut-off. The data in Figure 3 clearly shows that:

1. **Original Arvand already significantly outperforms the state of the art for tightly constrained resources.** With $p = 0$, at the runtime cut-off 40 minutes used in Figure 1, coverage for $C = 1.0$ is 18% compared to 4% for LPG, and for $C = 1.1$ it is 42% compared to 8% for LPG.
2. **Smart restarts can significantly improve Arvand for tightly constrained resources.** With $p > 0$, coverage for $C = 1.0$ improves to 46% at $p = 50$, and for $C = 1.1$ it reaches 90% at $p = 100$. Note the total difference to the previously best planner LPG: a factor > 11 improvement in coverage. Of course, this comparison is a little unfair since, as yet, we do not automatically determine a good value of p (this is an open problem). Still, the amount of improvement that *can* be obtained is impressive.
3. **The benefit of smart restarts tends to grow as C tends to 1.** This is obvious from how the curves develop from (a) to (d). The phenomenon nicely matches our derivation of this algorithm enhancement, which was entirely motivated by considering the economization of resources.

Looking at Figure 3 (a) and Figure 3 (b), we clearly see the sweet-spot behavior across p that was already somewhat visible in Figure 2 (b). Interestingly, the position of the sweet-spot depends on the runtime cut-off. For example, for $C = 1.0$ and time-out 10 minutes, the sweet spot is reached already at $p = 5$; for the 40 minute time-out, the sweet-spot appears to be at $p = 50$. Similarly for $C = 1.1$. An intuitive explanation of this has been given previously when describing Algorithm 1. Larger p yields a more explorative search, which may pay off by solving more examples, but only provided there is enough runtime to do so. If runtime is limited then a more greedy search may "get lucky" more often.

IPC Benchmarks

For cross-checking how our new methods perform on standard benchmark instances, we now perform experiments on IPC domains and instances. As explained previously, there are only three IPC STRIPS domains that are "resource-constrained" in the sense explored here: Mystery and Mprime of the 1998 IPC, Trucks of the 2008 IPC. There are several other IPC domains involving resources, however in most of them resource consumption appears only in the (numeric) optimization criterion, which is ignored by most existing planners,[9] and which, anyway, does not *force* a satisficing planner to economize the resources. In the Satellite and Rovers domains of IPC 2002, resources are a hard constraint in our sense, however in Rovers there is a refuelling action that we don't allow, and anyhow resources are present only in the numeric domain versions, in both cases.

[9]To our knowledge, the only exceptions to this rule are LPG and a non-default version of Metric-FF.

Instance	FF	LPG	LAMA	LM-cut	MDA	MDA50	MHA	MHA50
Mystery02	0.0	122.1	0.1	26.2	8.3	0.5	0.3	0.3
Mystery06	–	–	317.7	–	148.6	39.1	272.5	129.0
Mystery10	–	1015.6	136.1	–	14.6	14.6	26.4	102.7
Mystery13	–	8.4	2.7	–	34.6	37.6	77.5	4.4
Mystery14	4.3	–	2.9	–	14.0	20.3	69.1	12.0
Mprime10	17.7	2.5	3.3	–	60.6	25.1	8.9	10.3
Mprime13	45.6	57.4	3.1	–	218.1	58.4	15.4	14.2
Mprime14	191.4	–	2.8	–	69.4	63.0	20.1	16.5
Mprime18	–	4.5	3.3	–	47.5	28.4	7.3	7.1
Mprime22	322.7	6.3	8.0	–	33.8	28.2	10.9	12.1
Trucks05	0.0		0.0	94.53	0.4	0.5	365.3	265.5
Trucks06	0.0		0.0	–	0.7	0.5	–	746.3
Trucks07	26.6		3.8	261.75	16.7	5.0	–	–
Trucks08	1.4		0.6	641.3	7.6	1.8	–	–
Trucks09	0.0		64.9	–	0.8	20.3	–	–
Trucks10	–		24.84	–	58.5	12.2	–	–
Trucks11	0.0		28.5	–	1.3	3.8	–	–
Trucks12	–		–	–	198.3	91.2	–	–
Trucks13	–		–	–	170.3	164.5	–	–
Trucks14	12.7		22.4	–	22.7	1.6	–	–
Trucks15	–		–	–	159.6	278.8	–	–
Trucks16	–		–	–	–	488.6	–	–
Trucks17	–		–	–	46.5	234.7	–	–
Trucks18	–		–	–	–	582.1	–	–
Trucks21	–		–	–	–	744.5	–	–
Trucks23	–		–	–	–	1012.7	–	–
Tankage07	0.0	0.4	0.2	398.4	0.9	2.0	0.3	0.5
Tankage17	70.2	–	3.7	–	–	–	–	–
Tankage23	–	–	–	–	5.9	5.83	16.59	42.0
Tankage26	–	–	–	–	270.1	32.6	73.0	72.1
Tankage28	–	–	–	–	208.0	–	–	–
Tankage31	29.2	–	249.7	–	202.4	250.1	–	–
Tankage33	–	–	22.9	–	246.3	12.18	802.6	–
Tankage35	–	–	–	–	165.1	354.1	103.8	143.0
Tankage38	–	–	538.4	–	–	464.4	600.1	136.4
Tankage49	–	–	358.8	–	–	–	735.2	1113.9
NoTankage37	–	–	4.9	–	3.5	9.4	39.6	11.2
NoTankage38	–	–	10.9	–	4.8	63.3	39.1	9.9
NoTankage39	–	–	0.6	–	6.0	10.5	15.1	13.7
NoTankage40	–	–	3.7	–	7.6	57.5	40.0	50.4
NoTankage41	6.1	37.5	1.7	–	2.0	2.8	0.8	8.9
NoTankage42	–	–	–	–	10.8	169.9	–	–
NoTankage43	–	–	–	–	20.2	85.9	–	–
NoTankage44	–	–	–	–	171.5	88.1	–	–
NoTankage45	–	–	17.9	–	62.6	32.9	–	1298.27
NoTankage46	–	–	–	–	364.7	900.0	–	–
NoTankage47	–	–	–	–	502.2	313.5	–	–
NoTankage48	–	–	–	–	608.5	668.3	–	–
NoTankage49	0.8	–	16.7	–	12.6	104.6	65.5	24.7
NoTankage50	1.4	–	103.4	–	27.81	106.2	30.1	31.8
Freecell-2000-13-1	204.0	–	47.0	–	68.1	299.6	609.9	112.6
Freecell-2000-13-2	2.9	–	498.8	–	16.2	121.5	169.4	287.9
Freecell-2000-13-3	12.4	–	11.6	–	44.8	140.3	1052.3	78.7
Freecell-2000-13-4	985.1	–	126.0	–	398.8	169.7	–	1060.8
Freecell-2000-13-5	585.4	–	297.5	–	248.8	281.5	–	–

Table 1: Runtime (seconds) in selected instances of IPC STRIPS domains (see text). MDA, MHA are the original search strategies of Arvand. MDA50, MHA50 denote ArvandSR with the respective search strategy and smart restarts with pool size $p = 50$. Dashes indicate time-outs (30 minutes), empty cells mean that the planner could not be run.

We hence consider, in what follows, Mystery, Mprime, and Trucks. To examine the effect of our changes to Arvand in other domains, we also consider some IPC domains that have a puzzle-like nature – while this kind of problem structure is not the focus of our work here, we generally consider it interesting, and perhaps related to the intricacies of economizing a scarce resource. We run all planners tested previously. The data is in Table 1.

Table 1 shows, for each domain, a varying number of instances depending on how interesting the observations to be made are.[10] In each case, the instances selected are the most challenging ones that at least one of the tested planners could solve. In Mystery and Mprime, not much is to be observed. All versions of Arvand are as reliable in coverage as LAMA (although often slower), whereas FF and LPG exhibit a rather erratic behavior. Like in NoMystery (but not without exceptions), MHA tends to be better than MDA. Smart restarts most often make no big difference, although sometimes they do significantly improve runtime, and in rare cases deteriorate it. The main conclusion to be drawn, specifically in comparison to NoMystery, is that these benchmarks do not allow to evaluate resource-constrained planning in an insightful way. The results in Trucks are more interesting. MHA here is so bad that it lags significantly behind even the optimal planner, A* with LM-cut. MDA, on the other hand, vastly outperforms the state of the art – as represented by FF and LAMA – especially when using the smart restars technique proposed herein.

Turning our attention to the puzzle domains, in Tankage and NoTankage smart restarts have no significant effect. Due to slow heuristic computation in these domains, the total number of search episodes in bigger tasks, which are the most interesting ones, barely exceeds 10. Therefore, smart restarting can not be used so often. In Freecell, MDA solves all the problems without needing any restarts. Therefore, it does not really matter what mechanism is used for restarting. However, smart restarts improve both runtime and coverage of MHA in Freecell.

Conclusion

Economizing limited resources is, clearly, an important situation to be considered for automated planning. Just as clearly, our results here (extending those of Hoffmann et al (2007)) show that current state-of-the-art planners are very bad at doing so. We propose local search as a potential way out, and we have shown that a simple improvement to Arvand can already make a big difference. This result is encouraging, yet is only the first step in comprehensively addressing this issue. Some important open points are:

- Experiment with additional resource-constrained planning domains, equipped with to-be-developed generators allowing to control C like in NoMystery.
- Develop strategies for automatically configuring the search, such as finding a good static or dynamic scheme for controlling the pool size p.
- Explore in a similar fashion other enhancements to local search. For example, more intelligent methods for prioritizing states in the pool, perhaps inspired by UCT (Kocsis and Szepesvári 2006), may be useful.

We hope that this line of work will inspire other researchers as well, so that with joined forces we can make up leeway on this glaring open problem.

[10]Note that A* with LM-cut is the only optimal planner here. Such planners are normally not compared with satisficing ones, since this comparison is unfair. We include LM-cut only for completeness, and do not wish to entail such a comparison.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the *really* hard problems are. In *Proc. 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, 331–337.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *J. Artificial Intelligence Research* 20:239–290.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. 6th European Conference on Planning (ECP-01)*, 121–132.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, 162–169.

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

Hoffmann, J.; Kautz, H.; Gomes, C.; and Selman, B. 2007. SAT encodings of state-space reachability problems in numeric domains. In *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 1918–1923.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artificial Intelligence Research* 20:291–341.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Proc. 17th European Conference on Machine Learning (ECML-06)*, 282–293.

Koehler, J. 1998. Planning under resource constraints. In *Proc. 13th European Conference on Artificial Intelligence (ECAI-98)*, 489–493.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *J. Artificial Intelligence Research* 20:1–59.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In *Proc. 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, 1766–1771.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. 23rd National Conference of the American Association for Artificial Intelligence (AAAI-08)*, 975–982.

Selman, B.; Levesque, H. J.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In *Proc. 10th National Conference of the American Association for Artificial Intelligence (AAAI-92)*, 440–446.

Wei, W.; Li, C. M.; and Zhang, H. 2008. A switching criterion for intensification and diversification in local search for SAT. *J. Satisfiability, Boolean Modeling and Computation* 4(2-4):219–237.