AYALSOPLAN: Bitstate Pruning for State-Based Planning on Massively Parallel Compute Clusters

Juhan Ernits, Charles Gretton, Richard Dearden
School of Computer Science
University of Birmingham, UK
{j.ernits,c.gretton,r.w.dearden}@cs.bham.ac.uk

Abstract

Many planning systems operate by performing a heuristic forward search in the problem state space. In large problems that approach fails, exhausting a computer's memory due to the burden of storing problem states. Moreover, it is an open question exactly how that approach should be parallelized to take advantage of modern multiple-processor computers and the proliferation of massively parallel compute clusters. This extended abstract proposes an answer to this second question, while also going some way to addressing the memory problems.

We present AYALSOPLAN, our entry in the Multi-Core Track of the 2011 International Planning Competition (IPC-2011). Our approach is to run many independent and incomplete state-based searches in parallel. Our approach deliberately exploits hashing collisions to limit the set of states an individual search can encounter. Also, none of the parallel searches store all expanded states, each corresponding to a memory efficient state-based reachability procedure, albeit incomplete. As soon as a search determines reachability, the parallel processing ceases, and a single-core computer can efficiently construct the plan.

Because the 2011 IPC evaluation environment of the Sequential Multi-Core Track is *not* a massively parallel computer, and moreover because it imposes a *very limited timeout*, we have limited expectations regarding how AYALSOPLAN might be ranked in that evaluation. Therefore, this extended abstract commits some space to presenting empirical data we collected when evaluating our approach on our local cluster, without any runtime restrictions – i.e., searches can only fail when memory is exhausted. It is in that setting that we demonstrate the positive characteristics of our approach.

1. Introduction

Most of the fastest modern planning systems – including LAMA (Richter and Westphal 2010), the winner of the Sequential Satisficing track of the 2008 IPC – implement a best-first search of the state space. Those searches operate by maintaining an *open list* of states that have been visited but not completely expanded, and a *closed list* of states which have been visited and completely expanded. The storage burden associated with keeping track of visited states is a major hindrance to the scalability of modern systems. This is the case for both *frontier* variants of best-first search (Korf et al. 2005), and more classical implementations. Indeed, we are unaware of any planning system that can solve all IPC benchmark problems, including the big ones, given unlimited processing time and reasonable limitations on available memory.

Bitstate hashing (Courcoubetis et al. 1992; Holzmann 1998) is a memory-efficient technique for keeping track of visited states in state-based searches. The technique tracks the visitation status of a state using a single element, usually a single bit, in an integer indexed array. When a state s is visited during search, the evaluation of a hash function at s maps that state to an index in the array. That s has been visited is recorded by the status of the indexed bit.[1] Bitstate hashing thus trades storage required to record visited states against the probability of *collisions*, which occur when two different states are indexed to the same array entry and therefore cannot be distinguished. Bitstate hashing in a planning context has been explored previously in (Edelkamp 2002; Edelkamp and Jabbar 2005). In this abstract we introduce the related technique of *bitstate pruning*, initially proposed for model checking (Ernits et al. 2006; Ernits 2005), to the planning setting. Bitstate pruning deliberately exploits the collisions of bitstate hashing to dynamically limit the set of states the search can encounter. Bitstate pruning can reduce the memory requirements of search at the expense of completeness. Also in a model checking setting, more-recently a technique that utilises deliberately undersized bitstate hash tables was proposed to alleviate the processing (resp. storage) burden of computing the heuristic value of states (resp. keeping track of states). In detail, Kupferschmid et al. (2006) proposed treating states that hash to the same entry of the undersized array to be of equal heuristic value.

Bitstate pruning as an approach can be applied in any planner that uses state space search to prove goal reachability – constructively or otherwise – while maintaining a collection of visited states. Moreover, the incompleteness of the resulting search can be mitigated by running multiple searches in parallel, each using a different array size. Our competition entry, AYALSOPLAN, is based on an implementation of bitstate pruning in LAMA. In order to help us demonstrate the effectiveness of bitstate pruning

[1] Some variants use multiple bits in multiple arrays according to a set of state hashing functions.

in planning, we have also implemented it in the straightforward satisficing state-based search of AYPLAN (Robinson, Gretton, and Pham 2008). In the remainder of this abstract, when addressing a specific implementation, we write AYALSOPLAN^{lm} if the base search procedure is LAMA, and AYALSOPLAN^{ay} for the AYPLAN implementation.

When exploiting the parallelisation that bitstate pruning makes possible on a cluster of computers, both AYALSOPLAN^{ay} and AYALSOPLAN^{lm}are parallel satisficing planners that consistently solve larger problems than the procedures of their respective base systems, AYPLAN and LAMA. Here, it is important we clarify that our evaluation does not impose a timeout, and therefore when we speak of scalability, we do so where planning failure occurs solely due to memory exhaustion – i.e., given all the time in the world, the parallel incomplete searches of AYALSOPLAN^{ay} can solve larger problems than the serial satisficing procedure of the base planner AYPLAN. Moreover, we also find that the parallel incomplete searches of AYALSOPLAN^{ay} can often solve larger and more difficult problems than LAMA with a 3GB memory limit and unlimited time. In this work we have not exhaustively evaluated AYALSOPLAN^{lm} because the many heuristics and search optimisations employed in LAMA obscure the results, and pose a massive burden on our limited cluster resources, if we have to evaluate all the varieties of LAMA. Finally, we believe that our evaluation using AYALSOPLAN^{ay} already makes a clear empirical case for bitstate pruning for planning on massively parallel architectures.

This extended abstract is organised as follows: In the next section we provide a brief introduction to the use of search in planning algorithms. In Section 3, we then present the bitstate pruning approach and show how the probability of a search reaching a state can change as the size of the hashtable changes. In Section 4, we discuss the specifics of how we have implemented bitstate pruning in a number of planning systems, and present an empirical evaluation of one of those systems in Section 5. In Section 6, we summarise our contribution, making some concluding remarks.

2. Best-First Forward Search

In the early 90s implementations of best-first search exhausted "the available memory on most machines in a matter of minutes" (Korf 1992). Nowadays it continues to be considered a memory intensive approach (Korf et al. 2005), nonetheless the approach underlies a majority of good planning systems. Indeed, the dominant satisficing planning systems are based exactly on a best-first search of the state space, the more successful approaches typically employing a variant of A^* (Hart, Nilsson, and Raphael 1972). This is evidenced by the successes at recent IPCs of such systems. These include LAMA, SGPLAN versions 4 and 5 (Hsu et al. 2006; Chen, Hsu, and Wah 2004),[2] FASTDOWNWARD (Helmert 2006), FASTFORWARD (Hoffmann and Nebel 2001), and HSP (Bonet and Geffner 2001). Overall, LAMA, the base procedure for our competition entry, is one of the more recent and scalable procedures in this vein.

Although the recent success of best-first search in a planning setting might partly be attributed to the relatively vast quantities of memory available on modern computers, it can mostly be attributed to three important developments. First, that of heuristics for the satisficing case, such as the FF-heuristic $h^{\mathtt{ff}}$ (Hoffmann and Nebel 2001), the causal-graph heuristic $h^{\mathtt{cg}}$ (Helmert 2006), and the landmark-counting pseudo heuristic $h^{\mathtt{lm}}$ (Richter and Westphal 2010);[3] second, the development of planning-specific preprocessing algorithms, such as relevance analysis methods (e.g., (Haslum and Jonsson 2000; Bacchus and Teh 1998)) and plangraph analysis (Blum and Furst 1997), and representational optimisations, such as compilations to *multi-valued* setting and the related hierarchical decompositions of planning tasks (Helmert 2006); third, search tricks, such as *deferred(/lazy) heuristic evaluation* and *preferred operators* that have been shown to drastically improve the efficiency of some heuristic search techniques in many planning benchmarks (Richter and Helmert 2009).

We provide a brief sketch of state-based best-first forward search as it typically appears in the discussed planning procedures, since we will use the terms later. The search can be described in terms of a bound ranking function from states to numbers $f : \mathcal{S} \to \mathbb{N}$, two container data-structures *open* and *closed*, and a search graph. Here, the evaluation of the ranking function at a state, $f(s)$, maps each state to a numeric value, thereby ranking states. Structure *open* contains states encountered by the search, forward from the starting-state s_0, whose successors have not all been evaluated – i.e, there are actions whose effects on states in *open* have not been evaluated. The *closed* structure contains states that were previously in *open*, and for which all successor states have been evaluated. Typically we say that states in *closed* have been "expanded", and that states in *open* are "unexpanded" (or in some searches "partially expanded"). The search graph has one vertex for each state occurring in either *open* or *closed*, and a directed edge (s,s') labelled with actions whose expansion at s induced a state transition to s'. Here, we use the notation s to refer both to the state $s \in \mathcal{S}$ and its corresponding vertex.

At the commencement of search *open* is a singleton containing s_0 and *closed* is empty. The search proceeds interleaving the selection of a state from *open* to expand and its expansion. This process executes until a goal state is reached during an expansion, or otherwise until all promising states have been expanded – e.g., in the case that the goal is not reachable according to the search constraints. State selection is done greedily according to the given evaluation function $f(s) = c(s) + \beta h(s)$. Here, $c(s)$ is the (sometimes approximated) length of the shortest path from s_0 to s, while $h(s)$ estimates the value of expanding s. The factor β, usually 1, determines how greedy the search is with respect to h.

[2]For the underlying search procedure SGPLAN uses: *metric*-FASTFORWARD, MCDC (a variant of *Metric*-FASTFORWARD), and LPG. Only the first two can be characterised as a best-first forward search. LPG is a local search procedure for planning inspired by the Boolean SAT(isfiability) procedure WALKSAT.

[3]Nowadays these heuristics are often used in combination in a so-called *multi-heuristic* setting (Helmert 2006).

Once a state s is selected it undergoes expansion. For one or more actions available at s, the search evaluates the successor states of s, adds them to *open* if they are not already contained in either *open* or *closed*, and updates the search graph to reflect node additions and/or altered connectivity. If this step exhausts the action possibilities at s, then s is moved from *open* to *closed*. If h is inadmissible, when expanding a state it might be that a better, less costly path to a *visited* successor s is discovered, and therefore that a better approximation of $c(s)$ is discovered. Many searches propagate that better estimate through the search graph. For example, in order to emphasise a laziness in computation, in some descriptions a state is said to be *reopened* – e.g. see (Hansen and Zhou 2007) – in the sense that when a better approximation of $c(s)$ is found, s is added again to *open*, and the c-value change starts to propagate when the reopened state s is again chosen from *open* for expansion.

During the expansion of a state the search establishes whether or not evaluated successor states are new. The details of how this is achieved are very important to our contribution. In practice, all encountered states (i.e., elements in *open* and *closed*) are stored in a sorted associative container,[4] or more commonly a hash table. When a successor state is considered during expansion, its membership in that container is tested to decide if a new state should be added to *open*, and how the details of the search graph should be altered. In many planning benchmarks solvers fail on large problem instances because storage of encountered states exhausts system memory.

Nowadays it remains an open question how best to exploit modern distributed computing environments to achieve better scalability and efficiency in state-based searches. In particular, how best to trade-off available processing and memory resources, avoiding exhaustion of system memory before search yields a solution.

3. Bitstate Pruning

Let us consider a best-first search that utilises bitstate hashing to distinguish between new and already visited states. At the beginning of search an array of bits H is initialised to contain zeros. Whenever a state s is added to *open* a bit in the hash array at the address of $\mathtt{hh}(s)$ is set, $H[\mathtt{hh}(s)] := 1$. Here, let $\mathtt{hh}$ be a state hashing function that satisfies the standard *uniform hashing assumption* for analysis purposes. When expanding the actions of a state s in *open*, a successor s' is added to *open* iff $H[\mathtt{hh}(s')] \neq 1$.

A hash collision occurs when several states hash into the same address in H. In the limit as $|H|$, the size of the hash array, goes to infinity, we can invoke a special $\mathtt{hh}$ that hashes each distinct state to a distinct bit in that array. However, since in practice we are constrained by available memory resources, $|H|$ is small and hash collisions occur. In previous applications of bitstate hashing several measures are taken to reduce such effects. For example, in Bloom filters (Bloom 1970), each state uses several bits in the hash table, and the addresses for a state are calculated by multiple hash functions. That approach decreases the probability of collisions if the hash table contains a small number of entries. Other analyses of bitstate hashing (Holzmann 1998; Dillinger and Manolios 2004; Kuntz and Lampka 2004) have also been concerned with reducing the collision/omission probability. (Holzmann 1998) proposes a sequential multihash principle that performs hashing repeatedly using independent hash functions. The overall effect is to reduce the probability of collisions occurring at the same places and thus avoiding omissions.

[4]This is the case in LAMA.

In our setting hash collisions are valuable, as we use them to reduce the number of states explored by a given instance of best-first search. In more detail, suppose the number of reachable states n in the search graph is much larger than $|H|$. Then the probability of collisions is 1 and states are dropped by a search. That problem can be mitigated by using sequential multihashing, but rather than reducing the state drop-rate to increase the exhaustiveness of a search, we use a sequential multihash idea to increase the probability of reaching a goal state early in multiple independent searches. We call the overall approach *bitstate pruning*.

To increase the probability of finding a goal according to bitstate pruning, it is necessary to either repeat the search with a different hash function, change the search policy,[5] or change the size of H. As such repetitions are independent and involve no communication, they can be performed in parallel. Essentially, we can leverage an abundance of independent processing units to quickly (wall time) find a good hash function and the corresponding plan.

In our approach $\mathtt{hh}(s)$ is a combination of two things: a hash function that takes the bitvector of a state as input and produces a hash value of some size, typically 32 or 64 bits, and the use of the modulo function ($\mathtt{mod}$) to calculate the address of the bit in H. Thus, changing the size of $|H|$ provides an easy way of changing the hash function.

A desirable side effect of bitstate pruning is that it imposes a limit to the *open* data structure: there can never be more states in *open* than there are bits in H. The exhaustion of memory due to a large number of states in *open* is one of the reasons why search with very large bitstate hash tables fails. Thus, by pruning the search graph slightly differently under each instantiation of bitstate pruning, we trade memory for CPU, admittedly with some repeated exploration of problem states.

Example

The Three Integer Problem has states consisting of the values of three integer variables and has three actions a_1, a_2 and a_3 that each increment one of those variables. The search graph for this problem is shown in Figure 1 expanded up to search depth 2.

We index states s_i in a sequence that corresponds to depth-first exploration of the state space.

Let us assume that we start exploring the state graph in Figure 1 using depth-first search, a depth limit of 2 actions,

[5]In iterative deepending, we can change the search depth, and in A^* we can alter β, h, and even alter how c is (approximately) evaluated.

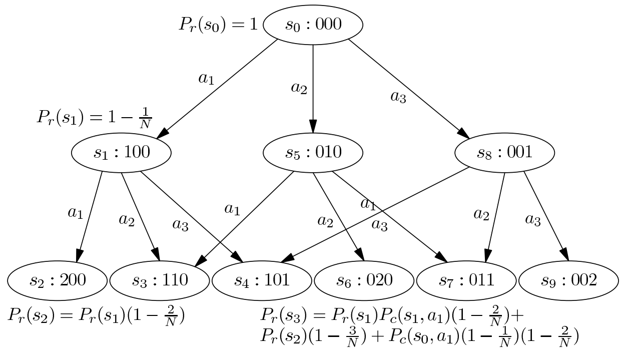

$P_r(s_0) = 1$
$s_0 : 000$
a_1
a_2
a_3
$P_r(s_1) = 1 - \frac{1}{N}$
$s_1 : 100$
$s_5 : 010$
$s_8 : 001$
$s_2 : 200$
$s_3 : 110$
$s_4 : 101$
$s_6 : 020$
$s_7 : 011$
$s_9 : 002$
$P_r(s_2) = P_r(s_1)(1 - \frac{2}{N})$
$P_r(s_3) = P_r(s_1)P_c(s_1, a_1)(1 - \frac{2}{N}) +$
$P_r(s_2)(1 - \frac{3}{N}) + P_c(s_0, a_1)(1 - \frac{1}{N})(1 - \frac{2}{N})$

Figure 1: Search graph for the three integer problem. $P_r(s_i)$ denotes the probability of reaching state s_i, and $P_c(s_i, a_i)$ denotes the probability of hash collision when executing action a_i at state s_i. N is the size of the bitstate hash table.

and a bitstate hash table of 2 bits. The probability of reaching the initial state, written $P_r(s_0)$, is 1. The probability of reaching s_1 is $P_r(s_0)(1 - 1/|H|) = 1/2$. When the size of the bitstate hash table is only 2, s_2 is not reachable, because to reach s_2, s_1 must be reached and then the hash table is full. As a result all actions from s_1 yield a collision. Although s_2 is not reachable, it is possible to reach s_5 with probability $P_r(s_0)P_c(s_0, a_1)(1 - 1/|H|) = 1/4$ – i.e. $P_c(s_0, a_1)$ is the probability of a collision when expanding a_1 at s_0, therefore overall the expression gives the probability of reaching s_0 times the probability of action a_1 yielding a collision, times the probability of a collision not occurring in s_5. Using a similar reasoning it is possible to reach s_8 with the probability of $P_r(s_0)P_c(s_0, a_1)P_c(s_0, a_2)(1 - 1/2) = 1/8$. It should be noted that in the given example there is also a $1 - 1/2 - 1/4 - 1/8 = 1/8$ probability that neither s_1, s_5 or s_8 is reached with a bitstate hash table size 2. With the bitstate hash table size 3 the probability of reaching any node in the search graph in Figure 1 using a depth-first search policy becomes nonzero.

4. Our Approach

We develop a satisficing planning approach, AYALSOPLAN, that can leverage the processing resources of cluster computing environments to obtain better scalability according to the bitstate pruning scheme just described. Our competition entry is based on AYALSOPLAN^{lm}, an implementation of this approach using LAMA, and our experimental evaluation is predominantly based on an implementation using AYPLAN. AYALSOPLAN uses multiple searches, each of which implement bitstate pruning for a distinct array size in what is otherwise a *frontier search* procedure (Korf et al. 2005). In order to construct a plan AYALSOPLAN operates in two phases: the first performs parallel plan existence searches, then if a plan exists, the second constructs a plan by repeating a successful incomplete search on a single core, this time storing all the encountered states. It is worth noting that because the frontier searches are independent, they can be run in parallel, in sequence, or a combination thereof. Therefore, in practice one can use several different cluster resources for planning.

Each executed search in the first phase corresponds to a *frontier search* (FS), a best-first search that uses only a fraction of the memory used by ordinary best-first searches of a state space. In detail, FS deletes states designated to *closed*, implicitly removing these from the search graph. Consequently FS is a sound approach to obtaining a proof of plan existence,however it is not constructive, because there is no data from which to extract a plan directly once a goal state is reached. Existing varieties of FS-based systems are rendered constructive by using them according to a divide-and-conquer query strategy, or otherwise by keeping *closed* on a secondary (slow) storage device. In this respect AYALSOPLAN differs from existing FS variants. AYALSOPLAN *deletes* states designated for *closed*, but also, according to bitstate pruning, an instance of search forbids multiple states which hash to the same entry of a bitarray from being considered. This implies two important consequences beyond the scalability obtained by exploiting bitstate pruning in multiple parallel instantiations of AYALSOPLAN. First, as with any FS variant, AYALSOPLAN uses relatively little memory when performing plan existence proofs. Second, a plan can be extracted relatively quickly during our post-processing phase using relatively little memory, by using the $|H|$ and search depth limit from a successful bitstate pruning FS search.

Overall, our contribution is in a similar vein to systems such as HDA* (Kishimoto, Fukunaga, and Botea 2009), which also tackle the A^* memory consumption problem in a parallel setting. HDA*-like systems use a more-or-less brute-force approach, distributing state storage over a cluster of multiple independent networked machines. A key advantage of our approach is that we require no inter-process communication. Also, because we perform a frontier search, we have diminish memory requirements, and improved scalability with the number of processors.[6]

5. Experimental Results

To demonstrate the efficacy of bitstate pruning, we compare AYALSOPLAN^{ay} with AYPLAN, a straightforward implementation of best-first search that stores all the visited states in *open* and *closed* explicitly. In this evaluation we also include the performance results of a August 2010 version of LAMA using the IPC-6 run script (without WA* iteration) as a reference. That system represents a state-of-the-art domain independent system for most of the benchmark problems we have considered. Finally, we also present preliminary experimental results using AYALSOPLAN^{lm}.

It remains to discuss a few planning specific details of AYALSOPLAN^{ay} that reduce the burden on search. Based on AYPLAN, AYALSOPLAN^{ay} incorporates a preprocessing phase that employs a number of computationally cheap problem analysis techniques. In particular, following (Haslum and Jonsson 2000), when grounding domain operators we omit from consideration actions whose precondi-

[6]Clarifying, this comment is not to be interpreted in the language of "speedup factors", rather, it is a comment about being able to solve larger problems given many CPU-cores, each with limited memory and unlimited time in which to solve a problem.

Table 1: Number of problems solved by of AYPLAN, AYALSOPLAN^{ay}, and LAMA. For some problems AYALSOPLAN^{ay} proves plan existence, however a plan cannot be extracted in phase-2 given the 2GB memory limit. In that case we report two figures: (a) outside parenthesis, the number of problems for which a plan could be extracted, and (b) in parenthesis, the number of problems for which the existence problem was solved.

Domain	AYPLAN	AYALSOPLAN^{ay}	LAMA
transport	9	30	30
pipes-tankage	23	43(44)	39
elevators	16	30	26
peg solitaire	30	30	30
scanalyzer	27	27	30
openstacks	14	17(18)	30
sokoban	12	15 (25)	25

tions are statically false. We further reduce the size of the set of ground operators and state propositions by performing *static relevance* testing as described in (Bacchus and Teh 1998). Although recent versions of AYPLAN implement a number of useful planning heuristics, in AYALSOPLAN^{ay} and AYPLAN we rank states in *open* according to how many goal propositions they satisfy. In many cases we find that other heuristics can be detrimental to performance of bitstate pruning on massively parallel machines in the benchmarks we have considered – This usually occurs because a heuristic encourages many of the parallel searches to be uniform, therefore the processing resources are not exploited for coverage. In Figure 4 this problem is indicated for the case of PIPES-TANKAGE P23, where as the bitarray becomes large the probability of AYALSOPLAN^{lm} searches failing increases.

Our evaluation compares the planners on several domains: The IPC 2004 PIPES-TANKAGE domain that poses an NP-Hard satisficing problem (Helmert 2006), and several of the IPC 2008 domains. In evaluation all AYPLAN-based processes were limited to use maximum 2GB of memory. LAMA was run on a single core of a computer with Intel quad core CPU Q9650 and 3GB of RAM. AYPLAN-based processes were run on a compute cluster with 20 quad-core Xeon E5345 CPUs totalling 80 CPU cores with 2GB memory per core. In no case do we impose any time limit. The default search depth for AYALSOPLAN^{ay} in all cases was limited to the minimum of 100000 and $|H|$.

Table 1 gives a summary of the results across the domains. Each row shows the number of problems solved (i.e. satisficing plans extracted) in the domain by each planner. The numbers in parenthesis in the AYALSOPLAN^{ay} column indicate the number of plan existences found as in some cases the plan extraction step exceeded the 2GB memory limit. Figure 2 summarizes the hash table sizes that were required to find plans. In many of the domains the $|H|$ values were in the order of tens or hundreds of thousands. In those cases, it is even feasible to run AYALSOPLAN^{ay} serially on a single core. Figure 3 reports plan costs that were obtained by AYPLAN, AYALSOPLAN^{ay} and LAMA. Here, we are reporting the costs of the first solution found by each planner.[7] In the PIPES-TANKAGE case the plan cost is equivalent to plan length. For that domain AYALSOPLAN^{ay} usually produces better initial plans than AYPLAN. In the PEG SOLITAIRE case, the plan qualities of first plans obtained are also of good quality. We should note that it would be a simple matter to implement a cost cutoff for AYALSOPLAN^{ay} in a manner similar to our maximum depth cutoff.

The results in Table 1 show that in all cases AYALSOPLAN^{ay} made it possible to solve more problems than AYPLAN, and in some cases, like PIPES-TANKAGE and ELEVATORS even more than LAMA. In SOKOBAN AYALSOPLAN^{ay} was able to detect plan existence in more cases than AYPLAN and LAMA, but we were only able to extract plans for 15 problem instances during phase-2 processing. In the SCANALYZER domain AYALSOPLAN^{ay} and AYPLAN found solutions to all problems that could get through AYPLAN preprocessing.

A selection of the results (easier problems solved by all planners are omitted) for PIPES-TANKAGE domain are given in more detail in Table 2. As we described above, AYALSOPLAN^{ay} uses frontier search to further decrease its memory requirements. On termination a plan is constructed using a second phase, that performs an ordinary search parametrised by the array size that yields a solution. For results presented in Table 2, the time and memory recorded for AYALSOPLAN^{ay} represent the time (memory) to run AYALSOPLAN^{ay} in FS mode to prove the existence of a plan, in addition to the time (memory) required to extract a plan in the second phase – by running a plan extraction instance of AYALSOPLAN^{ay} with the size of the hash table discovered by a successful instance of FS search. Space limitations mean that we cannot include results from all the domains we tested in Table 2, however we present PIPES-TANKAGE to highlight some of the interesting findings. The first four and last two columns show respectively the performance of AYPLAN, AYALSOPLAN^{ay} and LAMA in terms of time and memory requirements. A dash in the memory requirements column indicates that after the reported amount of time the planner ran out of memory without producing a plan. Thus, AYPLAN failed on ten of the 14 problems in the table, LAMA failed on three, and AYALSOPLAN^{ay} solved all of them. The three columns following AYALSOPLAN^{ay} show the performance of the plan existence (frontier search) part of AYALSOPLAN^{ay}. The hashtable size column shows the bit length of the hash array required to find a plan, the memory column is the total memory required by AYALSOPLAN^{ay} to detect plan existence on this size array, and the time column shows the average runtime for runs of AYALSOPLAN^{ay} with array sizes close to the presented bitstate hash table size that led to a plan. The reason we choose this measure is that the runtime is typically much faster when a plan is found than when one is not found, so these times represent the time for an exhausive search given a hashtable of the largest size that needs to

[7]None of the costs are guaranteed to be optimal.

Table 2: Memory and time requirements for finding satisficing solutions for some of the problems in the PIPES-TANKAGE domain using AYPLAN, AYALSOPLAN^{ay} and LAMA. AYALSOPLAN^{lm}could find solutions to problems P23, P26 and P28.

Problem	AYPLAN		AYALSOPLAN		AYALSOPLAN existence			LAMA	
	time (s)	memory (MB)	time (s)	memory (MB)	time (s/iter)	memory (MB)	hashtable size (bits)	time (s)	memory (MB)
P16_NET2_B14_G6_T80	1747	-	10.78	89	2.69	81	31441	6	32
P17_NET2_B16_G5_T20	26.4	191	2.44	38	.1	38	3066	5	20
P18_NET2_B16_G7_T60	786	-	5.25	78	.1	78	1356	3.24	23
P19_NET2_B18_G6_T60	23.65	165	7.06	113	.17	112	2169	5.70	14
P20_NET2_B18_G8_T90	1698	-	14.49	166	.13	166	4843	12.60	26
P21_NET3_B12_G2_T60	596	-	2.74	42	0.03	42	832	2.86	6
P22_NET3_B12_G4_T60	15.2	94	3.59	42	0.323	42	5607	4.2	23
P23_NET3_B14_G3_T60	1451	-	9.69	115	1.16	115	9410	800	-
P24_NET3_B14_G5_T60	1451	-	13.0	115	2.54	115	12711	9.0	43
P25_NET3_B16_G5_T60	1717	-	407	574	151	326	510642	46.7	77
P26_NET3_B16_G7_T70	1928	-	191	506	97.9	308	384214	2280	-
P27_NET3_B18_G6_T70	698	-	112	301	81.4	233	223881	11.6	82
P28_NET3_B18_G7_T70	698	-	371	879	184	505	904270	1150	-
P29_NET3_B20_G6_T70	1204	-	1065	1560	517	827	1428472	22.1	117

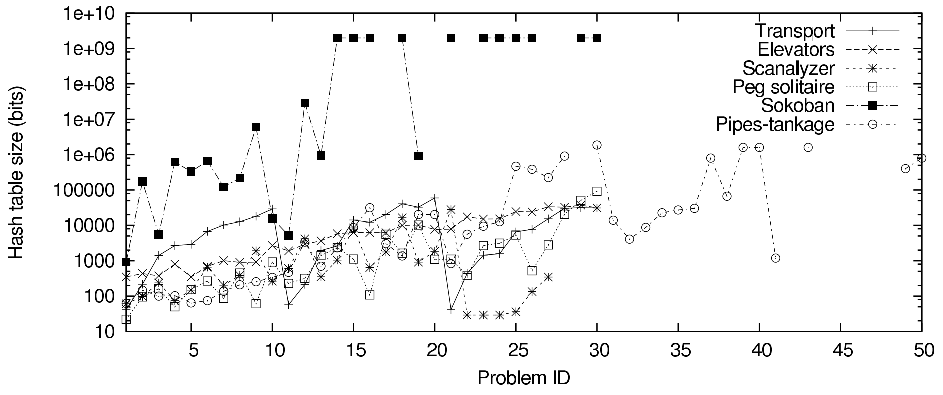
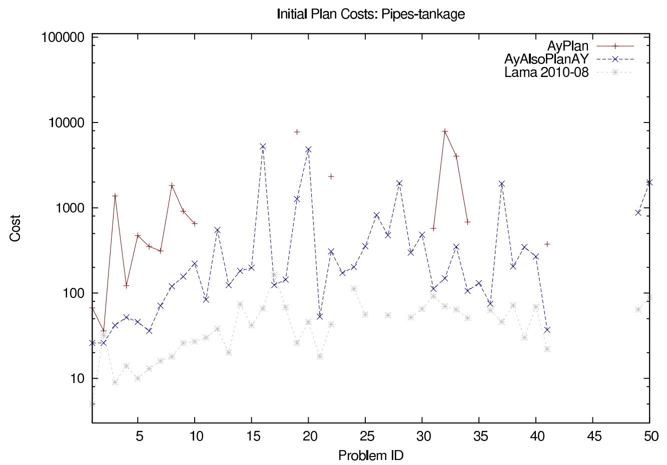


Figure 2: Hash table sizes in bits required by AYALSOPLAN^{ay} to find a plan in the corresponding domain.



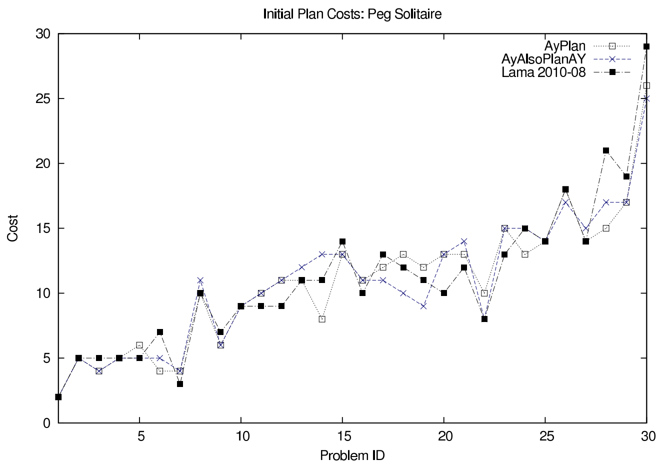
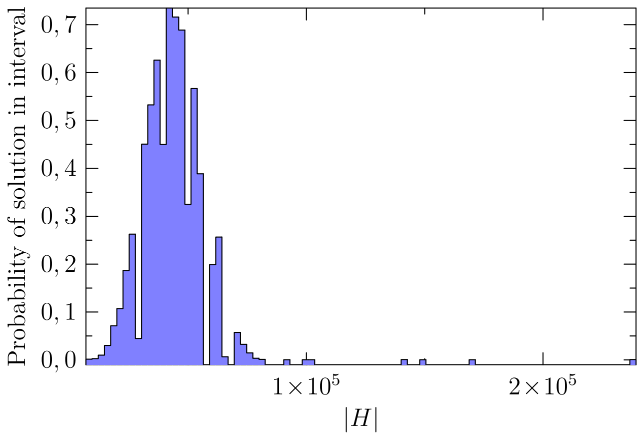


Figure 3: Initial plan costs of plans found by AYPLAN, AYALSOPLAN^{ay}, and LAMA.

be evaluated. A reasonable approximation to the total time required to find a plan (distributed over all the parallel processors) is the time per iteration multiplied by the hashtable size and divided by two and divided by the number of processors we have available. In addition, it is easy to see that the smaller the $|H|$ the faster the search.

Though very preliminary, in Figure 4 we explore the behaviour of AYALSOPLAN^{lm} in PIPES-TANKAGE problem P23, a problem that LAMA is not able to solve and that AYALSOPLAN^{lm} solves very quickly – in the best case in 8 seconds while using 20 MB of memory at $|H| = 6834$. Not surprisingly, on the lower graph, we show that the maximum memory consumed by search instances grows linearly with the bitarray size $|H|$. The data also demonstrates the difficulty of predicting the time a search at some $|H|$ will take. Most importantly, from the data-points we have, it is clear that LAMA's search guidance has a negative impact in this problem instance, stopping goal states from being discovered at larger $|H|$ values.



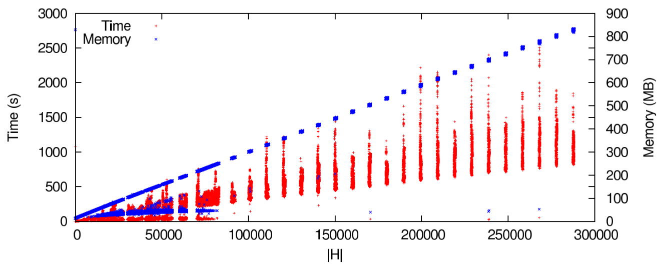


Figure 4: Above: Probability of finding a plan at a given hash table size $|H|$ in P23_NET3_B14_G3_T60 by AYALSOPLAN^{lm}. Below: Time and memory consumption for either finding a plan at a $|H|$ value or for exhausting the reachable state space in problem P23_NET3_B14_G3_T60 by AYALSOPLAN^{lm}.

6. Final Remarks

We described an approach to state-based planning in massively parallel systems that corresponds to an application of bitstate pruning in domain independent satisficing planners. Our approach instantiates multiple independent and incomplete searches in parallel on separate CPU-cores, each of which has limited memory resources. Each individual search can be characterised as an *incomplete* variant of the *one bit per state* approach described in (Korf et al. 2005). Given sufficient processors, in the limit as the number of processes goes to infinity the overall search is sound and exhaustive, one of the searches eventually finding a goal state (if reachable).

We have implemented our approach using both LAMA and AYPLAN as base planners, the former corresponding to our entry, AYALSOPLAN, at IPC-2011. In this extended abstract we have performed an empirical evaluation of AYALSOPLAN^{ay}, our implementation of bitstate pruning using AYPLAN. That evaluation is over several important IPC benchmarks, and demonstrates that given an abundance of processor resources each with limited memory resources, the technique of bitstate pruning can outperform the same planner without it by a significant margin where planning failures only occur due to memory exhaustion (resp. a timeout). The relative memory efficiency of AYALSOPLAN allows it to solve very large planning problems, some of which cannot be solved by good serial systems, such as LAMA.

AYALSOPLAN can be used in the same way as AYPLAN and LAMA for iterative plan refinement after the first plan for a problem has been discovered. Indeed, our competition entry AYALSOPLAN^{lm} uses the entire evaluation period to iteratively improve the plan prescribed in finality. Our experiments suggest that the costs of plans produced by AYALSOPLAN^{ay} are not significantly worse than those produced AYPLAN, in fact, in the PIPES-TANKAGE domain the initial plans by AYALSOPLAN were better.

Finally, it is worth noting that when LAMA's search guidance is very useful, the performance of AYALSOPLAN^{lm} can be worse than that of the base planner. Therefore, our entry in the Multi-Core Track runs the August 2010 of LAMA in one thread.

References

Bacchus, F., and Teh, Y. W. 1998. Making forward chaining relevant. In Simmons, R. G.; Veloso, M. M.; and Smith, S., eds., *AIPS*, 54–61. AAAI.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7):422–426.

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

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

Chen, Y. X.; Hsu, C. W.; and Wah, B. W. 2004. SGPlan: Subgoal partitioning and resolution in planning. In Proc. *IPC4*.

Courcoubetis, C.; Vardi, M. Y.; Wolper, P.; and Yannakakis, M. 1992. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1(2/3):275–288.

Dillinger, P. C., and Manolios, P. 2004. Bloom filters in probabilistic verification. In Hu, A. J., and Martin, A. K., eds., *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, 367–381. Springer.

Edelkamp, S., and Jabbar, S. 2005. Accelerating external search with bitstate hashing. In *KI05 (German Conference on AI) 19. Workshop on New Results in Planning, Scheduling and Design*, 7pp.

Edelkamp, S. 2002. Memory limitations in artificial intelligence. In Meyer, U.; Sanders, P.; and Sibeyn, J. F., eds., *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, 233–250. Springer.

Ernits, J.-P.; Kull, A.; Raiend, K.; and Vain, J. 2006. Generating tests from EFSM models using guided model checking and iterated search refinement. In Havelund, K.; Núñez, M.; Rosu, G.; and Wolff, B., eds., *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 85–99.

Ernits, J. 2005. Memory arbiter synthesis and verification for a radar memory interface card. *Nordic Journal of Computing* 12(2):68–88.

Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds. 2009. *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *J. Artif. Intell. Res. (JAIR)* 28:267–297.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. *Correction* to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.* (37):28–29.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In *In AIPS*, 150–158. AAAI Press.

Helmert, M. 2006. New complexity results for classical planning benchmarks. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 52–62. AAAI.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.* 14(1):253–302.

Holzmann, G. J. 1998. An analysis of bitstate hashing. *Form. Methods Syst. Des.* 13(3):289–307.

Hsu, C. W.; Wah, B. W.; Huang, R.; and Chen, Y. X. 2006. New features in SGPlan for handling preferences and constraints in pddl3.0. In Proc. *IPC5*.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In Gerevini et al. (2009).

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. E. 1992. Linear-space best-first search: Summary of results. In *AAAI*, 533–538.

Kuntz, M., and Lampka, K. 2004. Probabilistic methods in state space analysis. In Baier, C.; Haverkort, B. R.; Hermanns, H.; Katoen, J.-P.; and Siegle, M., eds., *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, 339–383. Springer.

Kupferschmid, S.; Hoffmann, J.; Dierks, H.; and Behrmann, G. 2006. Adapting an AI planning heuristic for directed model checking. In Valmari, A., ed., *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 35–52.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini et al. (2009).

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

Robinson, N.; Gretton, C.; and Pham, D.-N. 2008. Co-plan: Combining SAT-based planning with forward-search. In Proc. *IPC6*.