Effective Bidirectional A* with Frontier Search and External-Memory Utilization

Robert Niewiadomski José Nelson Amaral
Robert C. Holte
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: {niewiado, amaral, holte }@cs.ualberta.ca

October 4, 2008

Abstract

We present an advanced Bidirectional A* algorithm featuring an application of Frontier Search and a strategy for the performance-efficient utilization of External Memory. We present the results of an experimental evaluation demonstrating that this algorithm is capable of tackling exceptionally large state spaces while consuming significantly less time and space than its A* counterpart. For instance, in solving difficult instances of the 5-by-5 Sliding-Tile Puzzle and the 4-peg Towers-of-Hanoi problems, using additive pattern-database heuristics, the typical reductions in time- and space-consumption are in the range of one to two orders of magnitude.

0.1 Introduction

A common perception regarding Bidirectional A* (BA*) is that it is inferior to A* in terms of time- and space-consumption. Although the basic BA* algorithm can indeed end-up consuming significantly more time and space than A*, an advanced BA* algorithm can end-up consuming significantly less.

In this paper we investigate the utilization of Frontier Search (FS) and External Memory (EM) in an advanced BA* algorithm. The aim of FS is reducing the space consumption of the search, while the aim of utilization of EM is increasing the amount of space that is available to the search. Although the utilization of FS and EM has been examined in the context of A*, to our knowledge, neither the utilization of FS or EM has been examined in the context of an BA* algorithm.

0.2 Background

In a BA* algorithm a 'forward instance' of A* that searches the state space beginning at s_s, the start state, is executed together with a 'backward instance' of A* that searches the transposed instance of the state space beginning at s_g, the goal state, in order to compute a minimum-cost path from s_s to s_g. This paper refers to the forward and backward instances of A* in a BA* algorithm as search d and search d', respectively, and vice versa, in addition to using d and d' subscripts to distinguish between various values of the searches.

BA* algorithms use either *front-to-back evaluation* or *front-to-front evaluation*. With front-to-back evaluation each search uses a heuristic that guides it towards its goal state [12, 9, 4, 1]. With front-to-front evaluation each search uses a heuristic that guides it towards the states in the frontier of the other search [2, 13]. Front-to-front evaluation is generally perceived as being impractical due to the high overhead of computing heuristic values. This perception may change with further research into multiple-goal pattern-databases [7]. This paper is concerned with BA* algorithms where front-to-back evaluation is used and where the searches do not re-open closed states in the presence of a consistent heuristic. A chronologically ordered listing of such BA* algorithms follows.

In Bidirectional Heuristic Path Algorithm (BHPA) [12], the searches are executed concurrently by way of alternating between the execution of a search step in search d and in search d'. This alternating is governed by a *search-effort allocation policy*, an example of which is executing a search step in the search with smaller number of open states. As search steps are executed, μ is computed as the minimum cost of a path from s_s to s_g through a state in the intersection of the search d and search d' search trees. Execution of search steps continues until either $f\text{-min}_d$ or $f\text{-min}_{d'}$ is at or above μ. BHPA does not improve upon A* because the searches can repeat each others search effort.

In BS* [9], BHPA is modified with the application of four techniques: *nipping*, *pruning*, *screening* and *trimming*. Let s_i be a state that is to be expanded in search d. With nipping, s_i is closed without expansion if s_i is closed in search d'. With screening, each state s_j that is generated in the expansion of s_i is not opened if its f_d-value is at or above μ. With pruning, if s_i is closed in search d then each state s_j that is a descendant of s_i in the search d' search tree is removed from the search d' open list. Let s_i be a state that is open in search d. With trimming, s_i is removed from the search d open list if the f_d-value of s_i is at or above μ. BS* improves upon BHPA by preventing each search from repeating the search effort of the other search and from searching needlessly.

In BS*-Add [4], BS* is modified with the application of the *add* technique. With add, each state that is generated in search d has its f_d-value computed as the sum of its g_d-value, its h_d-value, and an estimate of the error of h_d in estimating the minimum cost of a path from an open state in search d' to the goal state of search d. This error estimate is computed as the minimum difference between the $g_{d'}$-value of an open state in search d' and its h_d-value. BS*-Add improves upon BS* by increasing the accuracy of heuristic estimates in each search using information gleaned from the other search.

In BS*-MaxRedux [1], BS* is modified with the application of the *max-redux* technique.[1] With max-redux, each state that is generated in search d, and each state that is open in search d, is screened and trimmed, respectively, based on the maximum of its f_d-value and its alternative f_d-value. The alternative f_d-value of a state s_i is computed as the sum of $f\text{-min}_{d'}$ and an estimate of the error of $h_{d'}$ in estimating the minimum cost of a path from s_i to the goal state of search d'. This error estimate is computed as the difference between the g_d-value of s_i and the $h_{d'}$-value of s_i. Like

[1]The max-redux technique is not to be confused with the *max* technique [4]. The max technique is also applied to BS* but can lead to the searches re-opening closed states in the presence of a consistent heuristic.

BS*-Add, BS*-MaxRedux also improves upon BS* by increasing the accuracy of heuristic estimates in each search using information gleaned from the other search.

Applying both add and max-redux to BS* has the potential of providing the benefits of both techniques. However, if both techniques are applied without modifying max-redux, the error estimate of max-redux in search d may double count the error estimate of add in search d'. The resulting erroneous screening and trimming may compromise the admissibility of the search. To eliminate this problem we modify max-redux such that the error estimate of add in search d' is subtracted from the error estimate of max-redux in search d. We refer to the combination of add and max-redux as the *add/max-redux* technique, and to the version of BS* that uses this technique as BS*-Add/MaxRedux. Experimental results indicate that although the benefits of add and max-redux in combination do not seem to be additive, combining add and max-redux does seem to provide the maximum benefit provided by either technique in isolation of the other.

0.3 Frontier BS*

Space consumption is a limiting factor in BS* because the searches are executed as instances of A*. An approach for reducing the space consumption of BS* is to execute the searches as instances of a Frontier A* (FA*) algorithm instead of as instances of A*. In the presence of a consistent heuristic, A* does not re-open closed states. A FA* algorithm leverages this property maintain an amount of information about the closed regions of the state space that is reduced but sufficient to prevent the search from leaking into these regions.

There are two mainstream FA* formulations: the Korf formulation (Korf-FA*) and the Zhou/Hansen formulation (Zhou/Hansen-FA*). Korf-FA* [8] maintains an open list but does not maintain a closed list. Each state in the open list has a set of state transitions S that consists of each state transition from that state to one of its closed successors. Each time an open state is expanded its S-set is consulted to prevent the generation of closed successors, the S-sets of open successors are updated, while the S-sets of non-open and non-closed successors are initialized. Zhou/Hansen-FA* [14] maintains an open list and a reduced closed list. The closed list contains all closed states that have non-closed predecessors, but does not contain any closed states that have no non-closed predecessors. Each state in both the open list and the closed list has an integer p that is equal to the number of non-closed predecessors of that state. Each time a state is expanded the p-values of closed successors and open successors are updated, while the p-values of non-open and non-closed successors are initialized. Whenever the p-value of a closed state reaches zero it is removed from the closed list.

In this section we present Frontier BS* (FBS*), a version of BS* where the searches are executed as instances of a FA* algorithm. The techniques that are applied to BS* to obtain BS* derivatives are also applicable to FBS*, as long as the technique does not lead to the searches re-opening closed states in the presence of a consistent heuristic. FBS* executes the searches as instances of Zhou/Hansen-FA* and not as instances of Korf-FA* because the former algorithm is better suited at facilitating the execution of BS* than the latter. In particular, in executing BS*, whenever search d either generates or expands a state that is closed in search d', the $g_{d'}$-value of that state is required. This $g_{d'}$-value is available in using Zhou/Hansen-FA* because the maintained information about closed states includes both the identities of closed states and their g-values. The same $g_{d'}$-value is not available in using Korf-FA*, however, since the maintained information about closed states is sufficient to obtain the identities of closed states, but is insufficient to obtain their g-values. This difference makes Zhou/Hansen-FA* better suited at facilitating the execution of BS*. That is not to say that Korf-FA* is not applicable. The aforementioned $g_{d'}$-value can be made available in using the Korf FA* algorithm with a modification where each state-transition in the S-set of each open state is tagged with the g-value of the corresponding closed state. In this paper, however, we do not pursue this modification.

0.3.1 The algorithm

The definition of the algorithm is literally identical to that of BS* with the exception of two differences.

The first difference is that in executing search d, the algorithm maintains p_d-values for states in the open and closed lists of search d, updates the p_d-values of states that are open or closed in search d, initializes the p_d-values of states that are opened in search d, removes states from the closed list of search d when their p_d-values become zero, and does not add states to the closed list of search d when their p_d-values are zero.

PDB	Time consumption		Space consumption	
	FA*	FBS*	FA*	FBS*
3-3-3-3-3	2.94+07	9.62+06	4.05+07	8.05+06
4-4-4-3	1.12+07	9.40+06	1.57+07	8.15+06
5-5-5	3.64+06	2.83+06	5.24+06	2.13+06
6-6-3	1.84+06	1.42+06	2.80+06	1.13+06
7-7-1	3.86+05	5.60+05	6.29+05	6.99+05
8-7	1.43+05	1.57+05	2.46+05	1.85+05

(a) STP.

PDB	Time consumption		Space consumption	
	FA*	FBS*	FA*	FBS*
5-10	6.71+07	1.09+07	2.14+07	2.88+06
6-9	1.09+08	1.93+07	3.06+07	6.00+06
7-8	1.33+08	2.83+07	3.60+07	8.31+06
8-7	1.37+08	5.80+07	3.76+07	1.42+07
9-6	1.21+08	8.38+07	3.20+07	2.02+07
10-5	8.83+07	9.23+07	2.23+07	2.11+07

(b) ToH.

Table 1: Time- and space-consumption of FA* and FBS* with add/max-redux on the 100-th instance in Korf's 4-by-4 STP problem instance set, and the standard instance the 15-disk ToH problem.

The second difference concerns pruning. Whereas in BS*, given a state s_i that is nipped in search d, the states that are pruned in search d' are all the descendants of s_i in the search d' search tree, in the algorithm the states that are pruned in search d' are all the descendants of s_i in the search d' search tree that are also successors of s_i. More specifically, the algorithm prunes in search d' all successors of s_i whose $g_{d'}$-value is equal to the sum of the $g_{d'}$-value of s_i and of the cost of the state transition to them from s_i. In principle, BS* performs *complete pruning*, while the algorithm performs *partial pruning*. The use of partial pruning in the algorithm is a necessity because the algorithm removes states from the search d' closed list. In doing so, the algorithm loses the ability to identify the descendants of states in the search d' search tree, with the exception of those that are also successors. Fortunately, by the time that the algorithm begins pruning it also begins trimming. Unless the heuristic is particularly inaccurate, the states that the algorithm fails to prune are likely to be good candidates for trimming.

Adhering to the policies of nipping, pruning, screening and trimming, gives rise to a *stale p-value problem*: the p-value of a state can be larger than it should be, an effect of which is bigger closed lists. For example, when a state s_i is removed from the open list because of trimming, the p-value of s_i is lost. When s_i is generated again and placed into the open list, the p-value of s_i is larger than it should be and, more importantly, will not reach zero when all predecessors of s_i are closed, which is the prerequisite for the removal of s_i from the closed list. A fix to the stale p-value problem is as follows. When nipping, expand the nipped state such that the f-value of each generated state is ∞. When screening, generate the screened state with g- and f-values equal to ∞. When pruning or trimming, retain the pruned or trimmed state while setting its g- and f-values to ∞. Unfortunately, the fix does more harm than good. Although using the fix does indeed lead to smaller closed lists, using it also leads to bigger open lists, with the increase in the size of the open lists being larger than the decrease in the size of the closed lists. Accordingly, we did not utilize the fix in obtained any of the results reported in this paper.

0.3.2 Experimental Evaluation

We performed an experimental evaluation of FA* and FBS* with add/max-redux. Table 1 highlights the results of this experimental evaluation: Table 1(a) presents the time- and space-consumption of the algorithms on the 100-th instance in Korf's 4-by-4 Sliding-Tile Puzzle (STP) problem set for various additive pattern-database heuristic configurations, while Table 1(a) does likewise on standard instances of the 15-disk 4-peg Towers-of-Hanoi (ToH) problem. Time consumption is measured in terms of the total number of state expansions, while space consumption is measured in terms of the maximum number of open or closed states at any given time during execution.

In general, FBS* consumes less time and space than FA*. The more accurate the heuristic the smaller the difference between the time- and space-consumption of FA* and FBS*.

0.4 External-Memory Frontier BS*

Even though FBS* is designed to consume less space than BS*, space consumption remains a problem. The utilization of External Memory (EM) in addition to Internal Memory (IM) alleviates this problem. The access latency and, to a lesser extent, the access bandwidth of EM is significantly worse than in the case of IM. Therefore, ensuring that EM is utilized in a performance-efficient manner requires the use of special techniques, such as sorting-based delayed-duplicate-detection [5, 11], hashing-based delayed-duplicate-detection [5, 3, 8] and hashing-based structured-duplicate-detection [15, 16].

In this section we present External-Memory Frontier BS* (EM-FBS*), a version of FBS* that utilizes EM in addition to IM, and External-Memory FA* (EM-FA*), a version of FA*, that utilizes EM in addition to IM. To utilize EM in a performance-efficient manner, both EM-FA* and EM-BS* use a sorting-based delayed-duplicate-detection strategy. The execution of EM-FA* forms the basis of the execution of EM-FBS*.

0.4.1 Runs and reductions

Both algorithms use *runs*. A run is a list of states where there are no duplicate copies of states and where states appear in the increasing order of the magnitudes of the binary-encodings of their descriptions. Both state-insertion and state-removal operations on a run must preserve the run property.

Both algorithms also use *reductions*. A reduction is an operation whose input is one or more copies of a state s_i and whose output is one copy of s_i, also called the *reduced copy*, such that the g-value of the output copy of s_i is the minimum of the g-values of the input copies of s_i, the f-value of the output copy of s_i is the minimum of the f-values of the input copies of s_i, and the p-value of the output copy of s_i is the difference between the number of predecessors of s_i and the sum of the differences between the number of predecessors of s_i and the p-value of each input copy of s_i.[2]

0.4.2 The unidirectional algorithm

The algorithm maintains EM runs $Open$ and $Closed$ and an IM list of EM runs ***Open***, such that $Open$ consists of the open states, $Closed$ consists of the closed states, and ***Open*** consists of the open states by way of consisting of as many EM runs as there are distinct f-values of the open states, such that each EM run in ***Open*** consists of all the open states with a given f-value and the EM runs in ***Open*** are ordered in the increasing order of their f-values, *i.e.* the f-values of their states.

While $Open$ is not empty and s_g is not in $Open$ with an f-value of f-min — which is computed as the f-value of the leading EM run in ***Open*** — the algorithm executes a search step. In each search step, the algorithm executes an *expansion phase* followed by either a *reconstruction-reconciliation phase*, or a *refinement-reconciliation phase*.

In the expansion phase, the algorithm removes the first EM run from ***Open***, and names it $Expand$. Next, the algorithm creates an IM list of EM runs ***Generate*** and an IM list of states $Generate$ with a fixed capacity that does not exceed the available IM capacity of the machine. Next, the algorithm executes a scan of $Expand$. In executing the scan, the algorithm expands each state in $Expand$ while appending each generated state to $Generate$. Each time $Generate$ becomes full or the expansions are finished, the algorithm sorts and reduces $Generate$ such that all copies of each state in $Generate$ are replaced with their reduced copy — thereby making $Generate$ an IM run — writes $Generate$ to EM to create a new EM run, appends this EM run to ***Generate***, and, if the expansions are not finished, resets $Generate$ to empty. When the expansions are finished, the algorithm destroys $Generate$.

In the reconstruction-reconciliation phase, the algorithm creates EM runs $tOpen$ and $tClosed$ and an IM list of EM runs ***tOpen***. Next, the algorithm executes a merge of $Open$, $Closed$ and each EM run in ***Generate***. While executing the merge, the algorithm computes s_i as the reduced copy of all copies of the current state in the merge output. Each time the next state in the merge output differs from the current state or the merge is finished, the algorithm processes s_i, such that at the end of the reconstruction-reconciliation phase, $Open$ and $Closed$ and ***Open*** are consistent with respect to the state expansions and generations in the expansion phase, should $Open$ and $Closed$ and ***Open*** be replaced with $tOpen$ and $tClosed$ and ***tOpen***, respectively. This processing of s_i involves append operations on $tOpen$, $tClosed$ and the EM runs in ***tOpen***, and find and insert operations on ***tOpen***. When it is finished executing the merge, the algorithm destroys $Open$, $Closed$, each EM run in ***Open***, ***Open***, $Expand$, each EM run in ***Generate***, and ***Generate***, and, subsequently, renames $tOpen$ and $tClosed$ and ***tOpen*** to $Open$ and $Closed$ and ***Open***, respectively.

In the refinement-reconciliation phase, the algorithm executes a merge of $Expand$ and each EM run in ***Generate***. While executing the merge, the algorithm computes s_i as the reduced copy of all copies of the current state in the

[2]For example, if there are three input copies of s_i such that the p-values of these three input copies are 3, 3 and 4, and the number of predecessors of s_i is 5, then the p-value of the output copy of s_i is 0, because we compute it as $5 - ((5-3) + (5-3) + (5-4))$.

merge output. Each time the next state in the merge output differs from the current state or the merge is finished, the algorithm processes s_i, such that at the end of the refinement-reconciliation phase, $Open$ and $Closed$ and ***Open*** are consistent with respect to the state expansions and generations in the expansion phase. This processing of s_i involves find, insert, remove and update operations on $Open$, $Closed$ and the EM runs in ***Open***, and find, insert and remove operations on ***Open***, as well as additional reduction operations involving s_i and copies of s_i in $Open$ or $Closed$. When it is finished executing the merge, the algorithm destroys $Expand$, each EM run in ***Generate***, and ***Generate***.

Instance Number	Time consumption		Space consumption	
	EM-FA*	EM-FBS*	EM-FA*	EM-FBS*
38	3.06+09	3.24+08	6.61+09	5.92+08
40	3.76+08	1.25+07	8.78+08	1.98+07
25	1.38+09	3.45+08	3.30+09	5.20+08
46	...	1.34+10	...	1.74+10
45	...	4.90+09	...	7.88+09
6	...	3.84+09	...	5.55+09
11	...	...	...	...
9	...	4.88+10	...	7.38+10
50	...	6.43+10	...	8.33+10

(a) STP.

PDB	Disks	Time consumption		Space consumption	
		EM-FA*	EM-FBS*	EM-FA*	EM-FBS*
15	14-1	2.23+05	4.91+04	3.40+05	7.01+04
16	14-2	2.00+07	3.50+06	1.23+07	2.40+06
17	14-3	2.99+08	3.27+07	1.71+08	1.69+07
18	14-4	2.29+09	1.77+08	1.21+09	7.85+07
19	14-5	1.24+10	6.65+08	6.23+09	2.95+08
20	14-6	...	1.89+09	...	7.95+08
21	14-7	...	8.79+09	...	3.32+09
16	15-1	3.01+06	9.65+05	3.17+06	8.21+05
17	15-2	7.45+07	1.57+07	5.16+07	9.12+06
18	15-3	8.12+08	9.88+07	4.78+08	5.14+07
19	15-4	5.23+09	4.14+08	2.57+09	1.79+08
20	15-5	2.50+10	1.25+09	1.11+10	4.91+08
21	15-6	...	4.13+09	...	1.69+09

(b) ToH.

Table 2: Time- and space-consumption of EM-FA* and EM-FBS* with add/max-redux on instances in Korf and Felner's 5-by-5 STP problem set, and on standard instances of various versions of the ToH problem.

0.4.3 The bidirectional algorithm

The algorithm maintains EM runs $Open_d$ and $Closed_d$ and an IM list of EM runs ***Open***$_d$, such that $Open_d$ and $Closed_d$ and ***Open***$_d$ are equivalent to $Open$ and $Closed$ and ***Open***, respectively, for search d in the unidirectional algorithm. In addition to maintaining the g_d-, f_d- and p_d-values of states in search d, as is done in the unidirectional algorithm, the algorithm maintains their $g_{d'}$- and $f_{d'}$-values.

While both $Open_d$ and $Open_{d'}$ are not empty and neither s_g is in $Open_d$ with an f_d-value of $f\text{-min}_d$ — which is computed as the f_d-value of the leading EM run in ***Open***$_d$ — or s_s is in $Open_{d'}$ with an $f_{d'}$-value of $f\text{-min}_{d'}$ — which is computed as the $f_{d'}$-value of the leading EM run in ***Open***$_{d'}$ — the algorithm executes a search step. In each search step the algorithm executes an expansion phase followed by either a reconstruction-reconciliation phase or a refinement-reconciliation phase.

In the expansion phase, the algorithm executes a unidirectional algorithm expansion phase for either search d or search d'. The algorithm modifies the execution of the unidirectional algorithm expansion phase for search d as follows. Let s_i be a state in $Expand_d$ to be expanded in the expansion phase of search d. If the $f_{d'}$-value of s_i is less-than or equal-to $f\text{-min}_{d'}$, then the algorithm modifies the expansion of s_i such that, for each s_j generated in the expansion of s_i, s_j is marked as a pruning state for search d', the g_d- and f_d-values of s_j are both equal to ∞, the $f_{d'}$-value of s_j is equal to ∞, and the $g_{d'}$-value of s_j is equal to the sum of the g_d-value of s_i and of the cost of the state transition from s_i to s_j. Otherwise, the algorithm modifies the expansion of s_i such that, for each s_j generated in the expansion of s_i, the algorithm does not append s_j to $Generate_d$ if the f_d-value of s_j is greater-than or equal-to μ.

In the reconstruction-reconciliation phase, the algorithm fuses the execution of the unidirectional algorithm reconstruction-reconciliation phase for both searches with the execution of a merge of $Open_d$, $Open_{d'}$, $Closed_d$, $Closed_{d'}$, each EM run in ***Generate***$_d$, and each EM run in ***Generate***$_{d'}$, while in the refinement-reconciliation phase, the algorithm fuses the execution of the unidirectional algorithm refinement-reconciliation phase for both searches with the execution of a merge of $Expand_d$, $Expand_{d'}$, each EM run in ***Generate***$_d$, and each EM run in ***Generate***$_{d'}$. In executing these merges, two versions of the reduced copy of the current state in the merge output are computed: a search d version and a search d' version. The algorithm processes these versions in the same manner as the unidirectional algorithm while also performing additional processing. This additional processing involves: the updating of μ using the g_d-values of states that are open in search d and the $g_{d'}$-values of their duplicates that are either open or closed in search d'; the updating of the $g_{d'}$- and $f_{d'}$-values of states that are open in search d using the $g_{d'}$- and $f_{d'}$-values of their duplicates that are either open or closed in search d'; the elimination of states that are open in search d with f_d-values that are greater-than or equal-to μ; the elimination of states that are open in search d with duplicates generated in search d' that are marked as pruning states for search d and have $g_{d'}$-values equal to the g_d-values of the states that are open in search d; and the ignoring of states in search d that were generated in search d but are marked as pruning states for search d'. In the case of the reconstruction-reconciliation phase, the additional processing is accomplished merely with logic operations on the two versions of the reduced instance of the current state in the merge output. In contrast, in the case of the refinement-reconciliation phase, the additional processing involves a combination of logic operations on the two versions of the reduced state, along with find, remove and update operations on $Open_d$, $Open'_d$, $Closed_d$, $Closed'_d$, ***Open***$_d$, ***Open***$_d$, along with find and remove operations on ***Open***$_d$ and ***Open***$_{d'}$.

0.4.4 Reconstruction versus refinement

The refinement-reconciliation phase is designed to be efficient when the ratio of the number of states expanded and generated in the expansion phase over the number of states that are open and closed is small, while the reconstruction-reconciliation phase is designed to be efficient when that ratio is not small. In practice, this ratio tends to be large in the first iteration with a given value of $f\text{-min}$, but then tends to grow smaller, eventually approaching zero, with each subsequent iteration with that value of $f\text{-min}$. After each expansion phase the algorithm chooses the reconciliation phase that is likely to yield the best performance. This decision should take into account both computational efficiency and data-reference locality. We note that the reconstruction-reconciliation phase can consume a significantly larger amount of space than the refinement-reconciliation phase unless the algorithm recovers space at incrementally while executing the merge.

0.4.5 Experimental evaluation

We performed an experimental evaluation of EM-FA* and EM-FBS* with add/max-redux. Table 2 highlights the results of this evaluation: Table 2(a) presents the time- and space-consumption of EM-FA* and EM-FBS* on nine instances in Korf and Felner's 5-by-5 Sliding-Tile-Puzzle (STP) problem set, using the additive pattern-database heuristic configurations described by Korf and Felner [6], while Table 2(b) presents the time- and space-consumption of EM-FA* and EM-FBS* on standard instances of various versions of the Towers-of-Hanoi (ToH) problem, using various additive pattern-databse heuristic configurations.[3] Time consumption is measured in terms of the total number of state expansions, while space consumption is measured in terms of the maximum number of open, closed or generated states at any given time during execution. The nine STP instances are arranged in the increasing order of difficulty as measured by the state generation totals reported by Korf and Felner, and correspond to the three least-, median-, and most-difficult instances. Because of resource limitations we were unable to solve all STP or ToH instances using EM-FA* that we were able to solve using EM-FBS*.

The results presented in Table 2 are in line with those presented in Table 1 with the exception that the margins by which EM-FA* beats EM-FBS* are larger than those by which FBS* beats FA*. In most cases, the margin of difference in favour of EM-FBS* is approximately an order of magnitude, with the extreme being approximately two orders of magnitude. Indeed, considering that we were unable to solve the most difficult instances of STP and ToH using EM-FA* while being able to do so using EM-FBS* is testament to the superiority of EM-FBS*.

The results presented in Table 2 were obtained using parallel implementations of EM-FA* and EM-FBS* that target distributed-memory systems. The parallel implementations are equivalent to their sequential counterparts in terms of work and storage, and were developed using the ideas of Niewiadomski *et al.* [11, 10]. Because the parallel implementations are undergoing performance tuning at the time of writing of this paper, and because we obtained results using different sets of machines, we are unable to provide meaningful execution-times. We do note, however, that the longest execution time was approximately a week using a dozen machines on the most difficult instance of STP, and that the decreases achieved by EM-FBS* over EM-FA* in the number of state expansions did indeed translate to similar decreases in execution time.

[3]In STP experiments, we did not utilize the blank-compression technique of Korf and Felner, which increases heuristic accuracy, since it compromises heuristic consistency

0.5 Conclusion

The principal contribution of this paper is an investigation of Bidirectional A* (BA*) with respect to the utilization of Frontier Search (FS) and External Memory (EM) in an advanced BA* algorithm. We presented BA* algorithms featuring an application of FS and a strategy for the performance-efficient utilization of EM, along with experimental results demonstrating that these BA* algorithms consume significantly less time and space than their A* counterparts. In general, we hope that this work will promote an increased awareness and appreciation of BA*. We plan on further exploring the issues addressed in this paper in addition to presenting an in-depth examination of the algorithms and their parallelization.

Bibliography

[1] Andreas Auer and Hermann Kaindl. A Case Study of Revisiting Best-First vs. Depth-First Search. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004*, pages 141–145, 2004.

[2] Dennis de Champeaux. Bidirectional Heuristic Search Again. *Journal of the ACM*, 30(1):22–32, 1983.

[3] Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl. External A*. In *Advances in Artificial Intelligence, 27th Annual German Conference on AI*, pages 226–240, 2004.

[4] Hermann Kaindl and Gerhard Kainz. Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[5] Richard E. Korf. Best-First Frontier Search with Delayed Duplicate Detection. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 650–657, 2004.

[6] Richard E. Korf and Ariel Felner. Disjoint Pattern Database Heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[7] Richard E. Korf and Ariel Felner. Recent Progress in Heuristic Search: A Case Study of the Four-Peg Towers of Hanoi Problem. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2324–2329, 2007.

[8] Richard E. Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier Search. *Journal of the ACM*, 52(5):715–748, 2005.

[9] James B.H. Kwa. BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artificial Intelligence*, 38(1):95–109, 1989.

[10] Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte. A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *Proceedings of the Thirty-Fifth International Conference on Parallel Processing*, pages 531–538, 2006.

[11] Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte. Sequential and Parallel Algorithms for Frontier A* with Delayed Duplicate Detection. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 1039–1044, 2006.

[12] Ira Sheldon Pohl. *Bi-Directional and Heuristic Search in Path Problems.* PhD thesis, Stanford University, 1969.

[13] George Politowski and Ira Pohl. D-Node Retargeting in Bidirectional Heuristic Search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 274–277, 1984.

[14] Rong Zhou and Eric A. Hansen. Sparse-Memory Graph Search. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1259–1268, 2003.

[15] Rong Zhou and Eric A. Hansen. Structured Duplicate Detection in External-Memory Graph Search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 683–689, 2004.

[16] Rong Zhou and Eric A. Hansen. Edge Partitioning in External-Memory Graph Search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2410–2417, 2007.