PNBA*: A Parallel Bidirectional Heuristic Search Algorithm

Luis Henrique Oliveira Rios[1] **, Luiz Chaimowicz**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{lhrios,chaimo}@dcc.ufmg.br

Abstract. *A** (A-star) *is a heuristic search algorithm used in various domains, such as robotics, digital games, DNA alignment, among others. In spite of its large use, A* can be a computationally expensive depending on the characteristics of the state space and heuristics used. Aiming at improving its performance, in this paper we propose a parallel implementation of a bidirectional version of A*. Named PNBA* (Parallel New Bidirectional A*) the proposed algorithm combines the benefits of bidirectional search and parallel execution in the development of an efficient A* based search algorithm. Experiments performed in different domains show that PNBA* is more efficient than the original A* and NBA*, the bidirectional version of A* it is based on.*

1. Introduction

Several problems in artificial intelligence can be solved by searching algorithms. In general, these algorithms explore large state spaces looking for a sequence of steps that will lead from an initial state to a goal state. One of the most studied algorithms in this area is A* [Hart et al. 1968], an *informed search algorithm* that assumes the existence of some problem-specific knowledge beyond the problem definition. This knowledge is represented by a heuristic function that guides the state space exploration. If this heuristic function respects some conditions, A* is proved to find the lowest cost solution.

By using a heuristic function, A* significantly reduces the number of states visited to find a solution without the sacrifice of the solution optimality. In spite of that, depending on the characteristics of the problem and heuristics used, its cost can still be prohibitive: in the worst case scenario, the number of states visited can be exponential in the length of the optimal solution. As discussed in [Rios and Chaimowicz 2010], several extensions and variations have been proposed to modify A*, improving its performance in different application scenarios.

Some of these extensions try to adapt A* for parallel execution. This possibility is gaining attention as multicore machines and computing clusters become more accessible. The main challenge faced by these implementations is how to reduce contention, *i.e.*, how to allow large periods of distributed computation without synchronization, still keeping the algorithm correctness.

The transformation of A* into a bidirectional heuristic search is another possible way to improve performance. On bidirectional search, the state space is simultaneously explored by two search process: one beginning at the start node and moving forward and the other from the goal and exploring the states backwards. A solution is found when the two exploration frontiers meet. The main motivation for the use of A* in a bidirectional setting is the possible reduction of the number of expanded nodes. In fact, recent results show that the combination of A* algorithm with the bidirectional search is able to significantly reduce the search effort [Whangbo 2007, Klunder and Post 2006, Pijls and Post 2009b].

In this paper, we present PNBA* - *Parallel New Bidirectional A**, a parallel shared memory version of NBA* [Pijls and Post 2009b] - a recently proposed bidirectional search algorithm based on A*. Our main objective is to combine the benefits of bidirectional search and parallel execution in the implementation of an efficient A* search algorithm. In the paper, we discuss the parallel implementation of the algorithm and present some arguments regarding its admissibility. We perform a series of experiments in three different scenarios (fifteen puzzle and grid pathfinding on random generated mazes - with uniform and non-uniform costs) and show that PNBA* is more efficient than the original A* and also NBA*.

This paper is organized as follows: next section brings some background on heuristic search, reviewing some key properties of A* and presenting some related work on its parallel and bidirectional implementations. Section 3 discusses NBA*, the bidirectional implementation of A* proposed by [Pijls and Post 2009b]. In Section 4 we present our new algorithm, PNBA*, discussing its implementation and main properties. The experiments performed with the new algorithm and the obtained results are described in Section 5. Finally, Section 6 brings the conclusion and possibilities for future work.

2. Background

A search problem is generally defined by the following basic components [Russell and Norvig 2003, Nilsson 1998]: an initial state, a goal state, a set of operators that modifies/transforms the states (successor function) and an optional step cost function that provides the cost of producing one state from another by applying the successor function. The initial state and the successor function combined implicitly define the state space [Korf 1996]. Its size (number of states) is drastically affected by the problem characteristics. A problem instance is the set of operators together with the initial and the goal states.

The process of searching consists of, from the start state, systematically applying the operators until reaching a goal state. That is, the intention is to transform the initial state into the goal state by employing the available rules. Normally, we are interested on the sequence of steps necessary to reach the goal state or, alternatively, to check if it is possible or not to reach a goal state. The step cost function provides a metric for the solution. Therefore, it is possible to impose restrictions over the solution and to evaluate its quality. For example, the objective may be finding the less expensive sequence of steps.

It is suitable to view the state space as a graph, in which the states are nodes and an edge from state x to state y indicates that the application of a specific operator on x generates y (we say that y is a successor of x). Edges may be labeled with the cost of this operation. Therefore, finding an optimal solution (the lowest cost one) for a search problem is equivalent to compute the shortest path between the start node and the goal node - from now on, denoted as x_{start} and x_{goal} respectively.

2.1. A*

A* is one of the most well known search techniques in Artificial Intelligence. It is a best first heuristic search algorithm, so it uses a heuristic function to guide the node expansion. At each iteration, it selects the most promising node according to an evaluation function f, that includes the real cost of going to that node and an estimate of the cost from that node to the goal.

More formally, let $d(x, y)$ denote the weight/cost associated with edge (x, y). The cost of the shortest path from x to y (in this case, not necessarily two neighboring nodes) will be represented by $d^*(x, y)$. As mentioned, informed search algorithms uses a heuristic function that provides more information to the search process. For a node x, $h(x)$ expresses this heuristic, which is an estimate of $h^*(x) = d^*(x, x_{\text{goal}})$. In most cases, $h^*(x)$ is not available in advance. Function $g(x)$ (an estimate of $g^*(x) = d^*(x_{\text{start}}, x)$) is the cost of the shortest path found so far from x_{start} to x. It is not necessarily equal to $g^*(x)$ because more than one path from x_{start} to x may exist and the algorithm might not have considered all of them at that time. The cost of the shortest path that connects x_{start} to x_{goal} and goes through x is represented by $f^*(x) = g^*(x) + h^*(x)$. The function $f(x) = g(x) + h(x)$ is an estimate of this cost and is used by A* to decide which node should be expanded next.

In the context of A*, the heuristic function may have two important properties. It is said to be *admissible* if $h(x) \leq h^*(x)$ for all nodes x, *i.e.*, the heuristic function never overestimates the real cost. Also, the heuristic function is said to be *consistent* or *monotone* if $h(x) \leq d^*(x, y) + h(y)$ for any nodes x, y (or, alternatively, $h(x) \leq d(x, y) + h(y)$ for any edge (x, y)). This can be viewed as a kind of triangle inequality: each side of a triangle can not be larger than the sum of the two other. Every consistent heuristic function is also admissible [Russell and Norvig 2003]. The importance of this distinction will be explained below.

A* is complete, *i.e*, it always terminates with a solution wherever one exists, if the following conditions are satisfied [Pearl 1984]: (i) nodes must have a finite number of successors and (ii) the weight associated with the edges must be positive. Also, it is admissible (returns the lowest cost solution) if one more condition is satisfied: heuristic admissibility. Imposing a more strict restriction (consistency) on the heuristic function allows a simpler and more efficient version of A*, in which a node needs to be expanded at most once. From now on, the heuristic function is assumed to be consistent.

Two data structures are generally used during the search to manage node expansions: *open* and *closed* lists. The *open* list contains all nodes that are in the frontier of the search (candidates to be expanded). The *closed* list stores the nodes that have already been expanded. The *open* list helps the algorithm during the selection of the most promising node to expand and is commonly implemented as binary heap. The *closed* list structure is used by A* to avoid the expansion of the same state multiple times and is frequently represented as a hash table.

Before starting the node expansion loop, A* initializes $g(x_{\text{start}})$ with 0 and then inserts this node in the *open* list. The loop is executed while there are any nodes in this list and x_{goal} has not been removed from it yet. At each iteration, the node with the lowest f-value is removed from *open* list for expansion. Its successors are generated and those that have not been expanded yet (are not in *closed* list) are inserted in the *open* list. It is possible that one or more of these nodes are already in the *open* list. In this case, a different path from x_{start} to this node has been found. If the new cost is smaller, the g-value of this node will be updated. Finally, the expanded node is inserted into the *closed* list.

There are basically two termination conditions for the algorithm. If there are no more nodes to be expanded (the *open* list is empty), there is no solution for the problem. On the other hand, if the goal state is selected for expansion from the *open* list, a solution has been found (an optimal solution, considering that the heuristic is admissible or consistent). A common way to retrieve the nodes that compose the final path is to keep a pointer from each node to its parent - the node that has been expanded to achieve it with the lowest cost. Hence, A* normally maintains an explicit search tree with all the generated nodes, a (hopefully very small) subset of the search space.

One of the most important consequences of the adoption of a consistent heuristic is that a node x is only expanded if the lowest cost path from x_{start} to x has already been found [Nilsson 1998]. This guarantees the second termination condition just mentioned. Another important consequence is that the sequence of f-values expanded is monotonically non-decreasing.

As mentioned, the main drawback of A* is that the number of nodes expanded can be exponential in the length of the optimal path. More formally, let b denote the node branching factor - the average number of successors over the search tree. Note that b is deeply related with the problem's successor function. Let d be the length - number of operators/edges employed in the transformation of x_{start} into x_{goal} - of the optimal solution. The worst case time and space complexity is $O(b^d)$ [Korf 1985].

2.2. Parallel A*

There are several reports on the adoption of parallel heuristic search algorithms based on A* to solve search problems. Grama and Kumar [Grama and Kumar 1993] provide a survey of these algorithms and presents a discussion of the challenges related with A* parallelization with feasible solutions for them. It is possible to classify these methods according to the memory architecture they assume available: shared memory or distributed memory, but most algorithms were designed for distributed memory architectures.

One example is Parallel Local A* (PLA*) [Dutt and Mahapatra 1993]. Basically, each processor has its local *open* and *closed* lists and the processors interact to inform the best solution found, to redistribute the work, to send or receive cost updates and to detect the algorithm termination. A new parallel start up phase and distribution strategy are proposed. The former has shorter execution time compared with previous works. The latter combines anticipatory work request, quantitative and qualitative load balance with a duplicate pruning mechanism to improve scalability. Different versions of this algorithm have been evaluated on a distributed memory environment to solve the Traveling Salesman Problem (TSP), showing a notable speed up in an environment with 256 processors.

Another A* based algorithm [Kishimoto et al. 2009] exploits distributed memory computing clusters using a hash function to distribute and schedule work among processors. Named Hash Distributed A* (HDA*), this algorithm maintains separated *open*/*closed* lists for each processor. The communication is performed through a message exchange protocol. Each processor executes the following steps while the computation has not finished. It process the received states using the *closed* list to avoid duplications and the *open* list to store the nodes it will expand. If there are no more nodes to be processed, it selects the node with the lowest f-value from its *open* list, expands and sends asynchronously those that do not belong to it to the proper processors. When a goal is found, a message is broadcast and the computation continues until all processors prove that is not possible to improve the solution quality. Although the authors concentrated the effort on parallelization for distribute memory architectures, an evaluation on a single, multicore machine has also shown good results.

Parallel Best-NBlock-First (PBNF) algorithm [Burns et al. 2009] is an example of a competitive shared memory parallel heuristic search algorithm. Its implementation employs a technique called Parallel Structured Duplicate Detection (PSDD) to deal with the synchronization. The idea is to create an abstract function (specific for each kind of problem) that maps several nodes of the original graph to a unique node of the abstract graph (denoted as nblock). PBNF maintains an *open* list and a *closed* list for each nblock. The abstract graph is used to help the selection of nblocks that can be explored in parallel without synchronization. As there is no restriction in the order nodes are explored, it will continue the search while there are nodes with f-values smaller than the cost of the current solution.

2.3. Bidirectional A*

As mentioned, bidirectional search executes two searches simultaneously: one forward from the initial state and the other backward from the goal [Russell and Norvig 2003].

Some of the early works to deal with bidirectional heuristic search did not show good results. [Pohl 1971], for example, argued that bidirectional heuristic search was not faster than its unidirectional version. A kind of consensus that the search frontiers pass each other (without meeting in the middle) has been established. In these cases, the number of expansions and the time of execution would be larger if compared with same values of unidirectional versions. A metaphor of two missiles separately directed at each other base desiring that they collide has been created to illustrate this problem [Pohl 1969]. The belief on this motivated the creation of front-to-front algorithms [Politowski and Pohl 1984, Davis et al. 1984] where each search aims at the opposite search tree frontier instead of its root. However, this approach did not solve the problem, resulting in computationally expensive algorithms or in algorithms that did not guarantee solution quality, as discussed in [Kaindl and Kainz 1997].

The conjecture of searching frontiers passing each other has been shown to be wrong [Kaindl and Kainz 1997] because the major computational effort is done in the post processing phase, after the two search frontiers meet. The authors have also presented evidences that bidirectional heuristic search is more appropriate for solving certain problems than the corresponding unidirectional search versions. These results have revived the interest on bidirectional heuristic.

More recently, bidirectional heuristic search algorithms that execute faster than their corresponding unidirectional version have been proposed. Whangbo presents an efficient modified bidirectional A* [Whangbo 2007] that improves previous versions in two aspects: by avoiding repetitive searches and through the use of a new stop condition that does preserve the admissibility. Other relevant work [Ikeda et al. 1994] shows ways of combining A* with bidirectional version of Dijkstra's algorithm without losing the guarantee of finding the optimal solution. This result has been employed with some improvements in an empirical evaluation of various shortest path algorithms [Klunder and Post 2006]. It concludes that bidirectional heuristic search is the most efficient alternative in the context of large real road networks.

One of the latest bidirectional heuristic search algorithms is NBA*, proposed in 2009 [Pijls and Post 2009b]. This algorithm is the base for our parallel implementation and will be discussed in the next section.

3. New Bidirectional A* (NBA*)

New Bidirectional A* (NBA*) [Pijls and Post 2009b] is a bidirectional heuristic search algorithm based on A*. Unlike the methods that inspired the creation of a preliminary version [Pijls and Post 2009a], NBA* just imposes one constraint on the heuristic function: consistency.

As any bidirectional search algorithm, two search process are managed simultaneously but not necessary in parallel. A common technique to accomplish the simultaneous execution in this context is to alternate the execution of each search process. This approach has been employed by NBA* authors. The search process number 1 operates in the original graph and process number 2 handles the reverse graph where every original edge (x, y) is replaced by an edge (y, x) with the same weight. Moreover, on the second process, the original start node (x_{start}) becomes the goal node and the original goal (x_{goal}) changes into start node. We will use the notation s_1, t_1 and s_2, t_2 for the start and goal nodes of each search process.

Before we start explaining the NBA* algorithm, the notation that has been employed so far should be enhanced. Let $d_p(x, y)$ and $d_p^*(x, y)$ denote, respectively, the same as $d(x, y)$ and $d^*(x, y)$ but for search process $p \in \{1, 2\}$. The subscript will also be added on the other elements resulting in $g_p(x)$, $h_p(x)$ and $f_p(x) = g_p(x)+h_p(x)$ estimates of $g_p^*(x) = d_p^*(s_p, x)$, $h_p^*(x) = d_p^*(x, t_p)$ and $f_p^*(x) = g_p^*(x) + h_p^*(x)$ respectively. Hence, the semantic is equivalent but now it is restricted to the search process expressed by p.

Like A*, NBA* uses a structure to keep control of node expansions. Using the original notation, $\mathcal{M}$ contains the nodes that are in the "middle", *i.e.*, between the two searches. Initially, all nodes are inside it. The nodes in the search frontiers are those that belongs to $\mathcal{M}$ and have been *labeled*. A node x is *labeled* if $g_1(x)$ or $g_2(x)$ is finite and the node has not been expanded. Besides $\mathcal{M}$, $\mathcal{L}$ is a shared variable that is read and written by both search process. It contains the cost of the best solution found by the algorithm so far and is initialized with an infinity value ($\mathcal{L} = \infty$). $\mathcal{L}$ is employed in the pruning criteria together with F_p, the lowest f_p-value on the frontier of search process p. Variables f_p, g_p and F_p are only written by one side but read on both.

The operations executed by each process will now be detailed. The search process number 1 has been chosen as reference but the operations are analogous for the other side. Initially, the g_1-value of all graph nodes is set to ∞. Then, the algorithm makes $g_1(s_1) = 0$ and $F_1 = f_1(s_1)$. At each iteration, the node $x \in \mathcal{M}$ with the smallest finite f_1-value is selected for expansion. It is removed from $\mathcal{M}$ and pruned (not expanded) if

$f_1(x) \geq \mathcal{L}$ [1] or $g_1(x) + F_2 - h_2(x) \geq \mathcal{L}$. Otherwise, all its successors y are generated. In the first case, it is classified as *rejected* and in the other situation as *stabilized* because $g_1(x)$ will not be changed anymore. For each y, $g_1(y)$ and $\mathcal{L}$ are updated, respectively, by the following statements values: $\min(g_1(y), g_1(x) + d_1(x, y))$ and $\min(\mathcal{L}, g_1(y) + g_2(y))$. At the end of the iteration, F_1 is updated with the lowest f_1-value inside the frontier. The execution stops when there are no more candidates to be expanded in one of the search sides. At this moment, $\mathcal{L}$ holds the cost of the optimal solution or an infinity value if no solution exits.

4. Parallel New Bidirectional A* (PNBA*)

The Parallel New Bidirectional A* (PNBA*), proposed in this work, is a parallel implementation of NBA*. It is similar to NBA* but tailored for parallel execution on shared memory architectures. The main idea is to run both search process in parallel instead of alternating their execution. As the majority of the variables involved in the algorithm are written by only one side of the search, the necessity of mutual exclusion sections is small, which tends to yield a good performance.

The code executed by one of the search processes is shown in Algorithm 1. The other process is analogous and can be obtained by properly replacing the subscripts (from 1 to 2 and vice-versa). The algorithm is very similar to the version presented on NBA* paper [Pijls and Post 2009b] but is more focused on implementation. We opt to explicitly represent the priority queue, denoting it by *open*$_p$ and adopting f_p as the sort criterion. Each side has its own priority queue that is able to execute the following operations: *insert*, *peek*, *pop*, *remove* and *size*. The variable *finished* indicates when the computation must stop and is common for both sides. The structure $\mathcal{M}$ is also shared and need to cope with concurrent access. Some initializations are made before each process start: on each side, $g_p(s_p)$ is set to 0 and s_p is inserted into *open*$_p$. Therefore, at the beginning, all nodes are on $\mathcal{M}$ and have their g_p-values equal ∞, except for s_1 and s_2 as $g_1(s_1) = 0$ and $g_2(s_2) = 0$.

Besides the inclusion of the mutual exclusion sections, one important change was made in the original algorithm: line 17, originally placed after line 3, was moved to its current position. This line is responsible for removing node x from $\mathcal{M}$. Without this change, PNBA* could possibly ignore some nodes depending on the execution sequence of some parallel statements. For example, suppose there is a graph with only two nodes a and b that are the start and goal respectively. Considering that there is an edge connecting a to b and line 17 is just after line 3 (like in NBA*), search process 1 removes a from its *open* list, then from $\mathcal{M}$ and the execution of this process is suspended. Search process 2 does the same steps with b, expands b generating a but is not able to update $g_2(a)$ and $\mathcal{L}$ as $a \notin \mathcal{M}$. Search process 1 continues its execution, expands a generating b but is not able to refresh $g_1(b)$ and $\mathcal{L}$ as $b \notin \mathcal{M}$ as well. The displacement of statement $\mathcal{M} \leftarrow \mathcal{M} - \{x\}$ is sufficient to overcome this problem with the drawback of possible unnecessary expansions that do not violate the solution optimality as will be discussed below. Another difference in the algorithm regards the update of $\mathcal{L}$ (lines 12-16). First, a verification is done outside the mutual exclusion section. Then, if there is a chance of a

[1]The original condition should be $f_1(x) - h_1(t_1) \geq \mathcal{L}$. However, since in most heuristics (including those used in this work) $h_1(t_1) = h_2(t_2) = 0$, this term was simplified to $f_1(x) \geq \mathcal{L}$.

better value to $\mathcal{L}$, the verification and update are done inside the mutual exclusion section to ensure the operation will be atomic.

Variables F_p, $\mathcal{L}$ and $\mathcal{M}$ are checked by the algorithm to decide if parts of the search tree may be pruned. As they are updated by both processes, their values may be outdated at the time they are read. In spite of that, we can argue that this does not affect PNBA* admissibility. It is important to notice that F_p and $\mathcal{L}$ are respectively monotonically non-decreasing and monotonically non-increasing. The former is guaranteed by the characteristics of the heuristic that is also monotonic and the latter by the lines that update $\mathcal{L}$ (lines 14-15). The g_p-values are also monotonically non-increasing (lines 6-7). All these conditions assure that the algorithm will not prune a node that should not be pruned, guaranteeing admissibility. It is worth mentioning that the opposite can happen: a node that should be pruned will be not. For example, a node that should not be expanded (because it is not more on $\mathcal{M}$) might be expanded due to the use of outdated information on line 3. But this will not affect the solution optimality as the algorithm verifies if the new values are smaller than the old values before updating $\mathcal{L}$ or g_p-values. It will only cause unnecessary computation that is compensated by the parallel execution, as shown in our experimental evaluation.

The g_p-values of a node can also be outdated on $\mathcal{L}$ actualization. However, after a write on $g_p(x)$ there is always an update on $\mathcal{L}$. Therefore, if the $g_1(x)$ change is followed by an update of $\mathcal{L}$ using $g_2(x)$ that is just about to be refreshed (with a lower value), search process number 2 will store the proper value on $\mathcal{L}$ because it will have the newest values of $g_1(x)$ and $g_2(x)$ available.

Algorithm 1 PNBA*, Parallel New Bidirectional A*

```
1:  while ¬finished do
2:    $x \leftarrow$ open$_1$.pop()
3:    if $x \in \mathcal{M}$ then
4:      if $(f_1(x) < \mathcal{L}) \wedge (g_1(x) + F_2 - h_2(x) < \mathcal{L})$ then
5:        for all edges $(x, y)$ of the graph being explored do
6:          if $(y \in \mathcal{M}) \wedge (g_1(y) > g_1(x) + d_1(x, y))$ then
7:            $g_1(y) \leftarrow g_1(x) + d_1(x, y)$
8:            $f_1(y) \leftarrow g_1(y) + h_1(y)$
9:            if $y \in$ open$_1$ then
10:             open$_1$.remove($y$)
11:           open$_1$.insert($y$)
12:           if $g_1(y) + g_2(y) < \mathcal{L}$ then
13:             lock
14:             if $g_1(y) + g_2(y) < \mathcal{L}$ then
15:               $\mathcal{L} \leftarrow g_1(y) + g_2(y)$
16:             unlock
17:     $\mathcal{M} \leftarrow \mathcal{M} - \{x\}$
18:   if open$_1$.size() $> 0$ then
19:     $F_1 \leftarrow f_1$(open$_1$.peek())
20:   else
21:     finished $\leftarrow$ true
```

5. Experiments

In this section, we present the experimental evaluation of the proposed algorithm. PNBA* is compared with A* and NBA* in three different scenarios: fifteen puzzle and grid pathfinding on random generated mazes with uniform and non-uniform costs. The former is a type of problem whose state space does not fit entirely on memory, thus it is not represented explicitly. During the computation, only the part needed is generated. The latter is assumed to fit entirely on memory and has an explicit graph associated with it. This distinction is important because it directly impacts implementation.

For both problems, a flag is kept to indicate wherever a node is on $\mathcal{M}$ (NBA* and PNBA*) or on the *closed* list (A*). As the state space is not explicit for fifteen-puzzle, a hash table is employed to guarantee that during the execution there will be an unique representation per state. For PNBA*, this structure implements atomically the operations *has* and *insertIfAbsent*. The first queries about the existence of an item and gets it while the second inserts an item if it is not there already. As the hash table implementation is based on buckets, just the affected one needs to be locked during these operations. For the other algorithms, the same implementation is used but without the overhead associated with the synchronization mechanisms demanded by PNBA*. A binary heap is used to implement the priority queue in the algorithms.

The implementations were done using the C++ programming language (g++ 4.4.3 compiler) with the *pthreads* library. The experiments were performed on an Intel(R) Core(TM) i5 3.20GHz CPU and with 4 gigabytes of memory, running Ubuntu with Linux kernel version 2.6.32. To evaluate the algorithms on fifteen-puzzle, an instance database was created. In this database, the instances are separated by the length (number of moves) of their optimal solution. The algorithms were executed on instances which solution length varied from 46 to 58 on increments of two and, for each length, we computed the arithmetic mean of 25 executions on different instances. On grid pathfinding, a series of square mazes were generated randomly. On these mazes, a cell was blocked with a probability of 33% and start and finish cells were positioned on the top left and bottom right corner cells respectively. The costs of moving between cells on non-uniform maze were randomly selected from 1 to 8. The maze size was varied from 1000 to 4000 in steps of 300. The data collected for each size was the result of 25 executions - mazes without solution are ignored until this number is achieved. The Manhattan distance was used as the heuristic on all domains. The ties (nodes with the same f-value) were broken in favor of those with the largest g-values. In all experimental scenarios the number of expanded nodes and the clock time spent were collected.

Figure 1 shows two graphs with the execution time and the number of node expansions as the fifteen-puzzle instances get harder. NBA* clearly beats A* on both criteria. It is able to expand fewer nodes because it has two search trees that together are smaller than A* search tree. NBA* pruning mechanism is important to achieve this result. The number of expansions of PNBA* and NBA* are very similar. However, PNBA* has a shorter execution time since it explores both search trees in parallel.

The results for the uniform grid pathfinding scenario can be seen on the graphs on Figure 2. The results are very similar, although PNBA* surprisingly expands a little bit less nodes than NBA* especially for large instances. A possible explanation is the non-synchronized manner through which the search space is explored. This can randomly

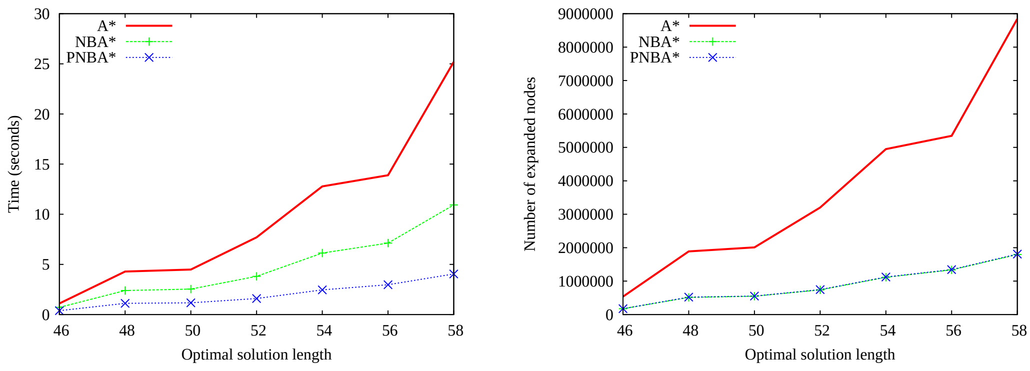

Figure 1. Results on fifteen-puzzle domain.

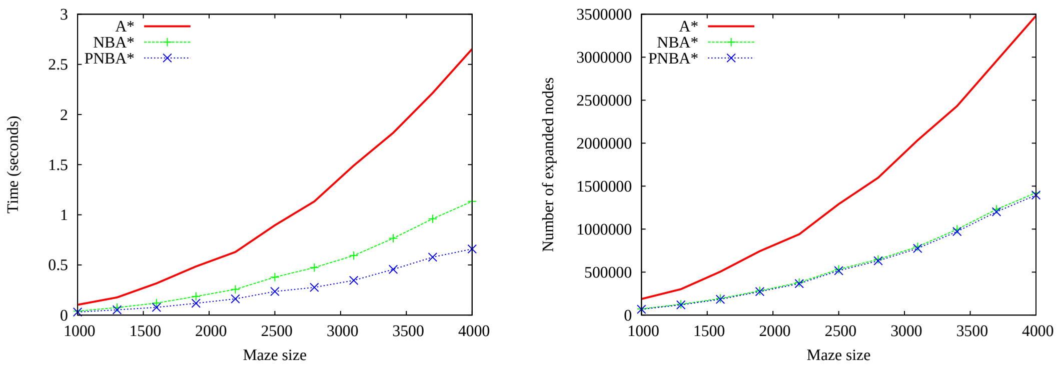

Figure 2. Results on uniform cost grid pathfinding domain.

favor the computation of a good initial first solution (very close to the optimal solution), enabling the pruning of more nodes during the rest of the computation.

Finally, the results for the non-uniform cost grid pathfinding are shown in Figure 3. The difference between figures 2 and 3 (uniform and non-uniform respectively) is remarkable. The problem with non-uniform cost is clearly harder as the heuristic estimates the lowest cost considering that all cells have the smallest weight: 1. This is done in order to preserve the admissibility property, since it is always possible that one path from start to goal passing on cells which cost is 1 exists. The consequence is that the heuristic may underestimate the real cost by a large factor, mainly when compared to the uniform cost domain. Therefore, the number of expanded nodes for the three algorithms increased, but as in the other scenarios, PNBA* and NBA* have similar values. In terms of execution time, the difference between A* and NBA* reduced significantly, but PNBA* was again the faster among the compared algorithms. Based on this experiment, it seems that bidirectional heuristic search is more affected by a poor heuristic than unidirectional versions, but a further investigation is necessary in order to evaluate this.

These results clearly demonstrate the advantages of PNBA*, the parallel implementation of NBA*. It leverages the benefits of a bidirectional heuristic search (reduction in the search space) by exploring both searches in parallel, reducing the total execution time.

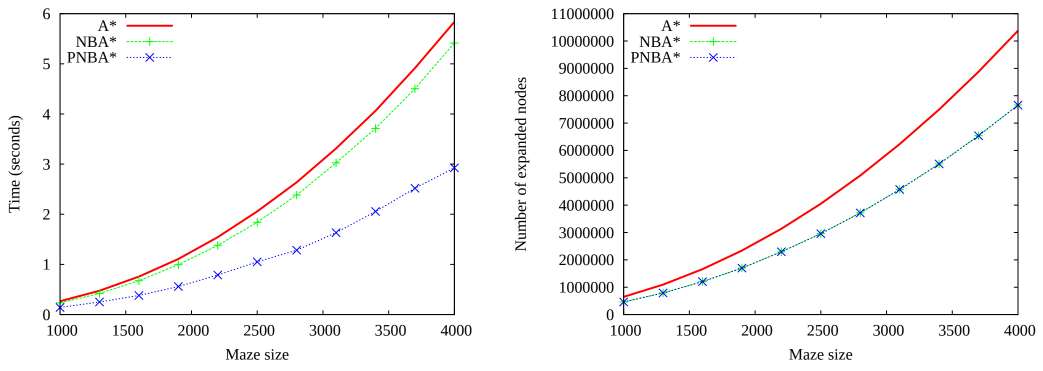

Figure 3. Results on non-uniform cost grid pathfinding domain.

6. Conclusion

This paper presented PNBA*, a parallel bidirectional heuristic search algorithm. Based on NBA*, a novel bidirectional version of A*, PNBA* improves A* performance by combining the benefits of bidirectional search and parallel execution. The algorithm was empirically evaluated in three different scenarios: fifteen puzzle and grid pathfinding on random generated mazes - with uniform and non-uniform costs and the results showed a significant reduction of execution time without the sacrifice of solution optimality. We also discussed the main characteristics of the algorithm and presented arguments that the parallel implementation does not violate its admissibility properties.

Continuing this work, we intend to extend the comparisons considering different application scenarios - maybe from the classical planning instances of the IPC (International Planning Competition) benchmark. They will also contemplate more parallel heuristic search algorithms, including bidirectional based parallelizations as [Sohn et al. 1994]. Additionally, together with a deeper analysis of these results, the development of formal proofs of PNBA* admissibility will be an important contribution. Finally, in order to better explore the current available multicore machines, we want to adapt PNBA* to include the ability of employing more than 2 two processors in its execution, allowing a greater performance improvement.

Acknowledgments

This work is partially supported by Fapemig and CNPq.

References

Burns, E., Lemons, S., Zhou, R., and Ruml, W. (2009). Best-first heuristic search for multi-core machines. In *Proceedings of IJCAI*, pages 449–455.

Davis, H. W., Pollack, R. B., and Sudkamp, T. (1984). Towards a better understanding of bidirectional search. In *AAAI*, pages 68–72.

Dutt, S. and Mahapatra, N. R. (1993). Parallel a* algorithms and their performance on hypercube multiprocessors. In *IPPS*, pages 797–803.

Grama, A. Y. and Kumar, V. (1993). A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing*, 7.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

Ikeda, T., Hsu, M.-Y., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K., and Mitoh, K. (1994). A fast algorithm for finding better routes by ai search techniques. In *VNIS*, pages 291–296.

Kaindl, H. and Kainz, G. (1997). Bidirectional heuristic search reconsidered. *J. Artif. Intell. Res. (JAIR)*, 7:283–317.

Kishimoto, A., Fukunaga, A. S., and Botea, A. (2009). Scalable, parallel best-first search for optimal sequential planning. In *ICAPS*. AAAI.

Klunder, G. A. and Post, H. N. (2006). The shortest path problem on large-scale real-road networks. *Networks*, 48(4):182–194.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

Korf, R. E. (1996). Artificial intelligence search algorithms. Technical report, University of California.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, San Francisco.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pijls, W. and Post, H. (2009a). A new bidirectional search algorithm with shortened postprocessing. *European Journal of Operational Research*, 198(2):363–369.

Pijls, W. and Post, H. (2009b). Yet another bidirectional algorithm for shortest paths. Technical Report EI 2009-10, Erasmus University Rotterdam, Econometric Institute.

Pohl, I. (1969). Bi-directional and heuristic search in path problems. Technical Report 104, SLAC (Stanford Linear Accelerator Center), Stanford, California.

Pohl, I. (1971). Bi-directional search. In Meltzer, B. and Michie, D., editors, *Machine Intelligence*, volume 6, pages 124–140, Edinburgh. Edinburgh University Press.

Politowski, G. and Pohl, I. (1984). D-node retargeting in bidirectional heuristic search. In *AAAI*, pages 274–277.

Rios, L. and Chaimowicz, L. (2010). A survey and classification of a* based best-first heuristic search algorithms. In *SBIA 2010*, volume 6404, pages 253–262. Springer.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education.

Sohn, A., Sato, M., Sakai, S., Kodama, Y., and Yamaguchi, Y. (1994). Parallel bidirectional heuristic search on the em-4 multiprocessor. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, pages 100–109.

Whangbo, T. K. (2007). Efficient modified bidirectional a* algorithm for optimal route-finding. In *IEA/AIE*, volume 4570, pages 344–353. Springer.