Article

JIS

Journal of Information Science
2014, Vol. 40(3) 363–375

DOI: 10.1177/0165551513519212
jis.sagepub.com
SAGE

Performance evaluation of parallel multithreaded A* heuristic search algorithm

Basel A. Mahafzah
Department of Computer Science, University of Jordan, Jordan

Abstract

Heuristic search is used in many problems and applications, such as the 15 puzzle problem, the travelling salesman problem and web search engines. In this paper, the A* heuristic search algorithm is reconsidered by proposing a parallel generic approach based on multithreading for solving the 15 puzzle problem. Using multithreading, sequential computers are provided with virtual parallelization, yielding faster execution and easy communication. These advantageous features are provided through creating a dynamic number of concurrent threads at the run time of an application. The proposed approach is evaluated analytically and experimentally and compared with its sequential counterpart in terms of various performance metrics. It is revealed by the experimental results that multithreading is a viable approach for parallel A* heuristic search. For instance, it has been found that the parallel multithreaded A* heuristic search algorithm, in particular, outperforms the sequential approach in terms of time complexity and speedup.

Keywords

A* algorithm; 15 puzzle; heuristic search; performance evaluation

1. Introduction

A searching algorithm is a description of the process in which an initial state and a goal state are provided as inputs, and a path sequence in the search space from the initial state to achieve the input goal is returned as output. If the goal is not found, on the other hand, a failed mark is returned as an output [1].

Searching algorithms are widely applied in the artificial and computational intelligence fields, such as game playing, robotics, planning, the travelling salesman problem, network management and discrete optimization problems [1–4], as well as other fields such as information retrieval systems, web search engines and road networks [5–8]. In the artificial and computational intelligence fields, searching algorithms are categorized as blind or heuristic searches. In blind search algorithms, all the paths in the search space have the same probability of leading to the goal state [1], while heuristic search algorithms use information to find the path leading to the goal state [1, 9]. Unless complexity boundaries are reached, sequential searching is reasonable. On the other hand, if performance problems are encountered, then multithreading serves as a solution.

Multithreading is a promising approach owing to its ability to decrease the amount of execution time required by the search process, where multiple threads perform the search process simultaneously (virtual parallelization), thus increasing the speedup of the sequential heuristic search algorithms, which is a significant improvement owing to the great role that speedup plays in performance improvement of heuristic search algorithms [10]. However, some important issues should be taken into consideration when designing a multithreaded heuristic search algorithm. The first issue deals with the synchronization of concurrent threads, which can be achieved by adding constraints on using the shared resources to save the shared data consistency. However, if a complicated synchronization strategy is used, the performance of such algorithms might be degraded [10]. The second issue is concerned with load balancing among threads, which measures the utilization of the concurrent working threads. The significance of such a measurement stems from the negative effect of

Corresponding author:
Basel A. Mahafzah, Department of Computer Science, University of Jordan, Amman 11942, Jordan.
Email: b.mahafzah@ju.edu.jo

the imbalanced load among threads on the performance of such algorithms, where one thread may work on a large task while other threads work on smaller tasks.

In this paper, the A* heuristic search algorithm is reconsidered by using a heuristic estimate to distinguish the path that may lead to the solution through expanding the node with the minimum estimated cost [1, 11]. More specifically, a parallel A* heuristic search algorithm is presented based on multithreading using the POSIX threads (Pthreads) library.

In summary, the following contributions are drawn from this paper:

- A parallel multithreaded A* heuristic search algorithm for solving the 15-puzzle problem is designed and implemented.
- The design and structure of the parallel multithread A* heuristic search algorithm is described in detail.
- The parallel multithreaded A* heuristic search algorithm is evaluated analytically in terms of time complexity, space complexity, completeness and optimality.
- The parallel multithreaded A* heuristic search algorithm is evaluated experimentally in terms of the following performance evaluation metrics: speed up, number of generated nodes, number of expanded nodes, number of duplicate checks, number of duplicates found and search effectiveness.
- A comparison is made between our parallel multithreaded A* heuristic search algorithm and the sequential A* heuristic search algorithm for various problem sizes and various numbers of concurrent threads.

The rest of this paper is organized as follows: Section 2 introduces the background and the related work. The proposed parallel multithreaded A* algorithm is demonstrated in Section 3. The analytical evaluation of the proposed parallel multithreaded A* algorithm is presented in Section 4. The experimental results are then shown and discussed in Section 5. Finally, the conclusions of the paper and the intended future work are presented in Section 6.

2. Background and related work

One type of searching algorithm in the artificial and computational intelligence fields is the blind search. Blind search (uninformed search) algorithms have no information about the states or search space. Searching in such algorithms is performed by generating successors and testing whether each of the generated successors is the goal state or not. Thus, for all the generated nodes, the expansion priority will be the same.

Breadth First Search (BFS) [1, 12] is a well-known blind search algorithm, in which all successors of a node are expanded at one step. The root node is expanded first, followed by its successors. Next, the resulted successors, which are at the same level of the tree, are expanded consecutively. BFS is an optimal algorithm, in which the optimal solution is guaranteed, if a solution exists. Using BFS, all the nodes in one level are expanded before any node in the next level. Therefore, the optimal solution will always be reached. BFS is also a complete algorithm since all the reached nodes are expanded during the search process. This result in an average time of $O(\beta k)$, where β is the branching factor, which defines the average number of successors generated from one node during the search process [9], and k is the depth of the shallowest solution. On the other hand, BFS requires large memory space to handle the large number of generated nodes, which is $O(\beta k)$ on average.

The Limited-Depth BFS algorithm [1] is a variation of BFS, in which a threshold value is specified on the maximum depth to which the search tree can be constructed. This value must be estimated carefully, such that it is large enough to cover the area of the tree that contains the solution. In addition, the threshold value should be small enough to avoid exhausting a large memory space, as is the case in the BFS algorithm. Conversely, if the threshold value is estimated appropriately, the Limited-Depth BFS algorithm is considered as an optimal algorithm since it finds the optimal solution with less cost because no node is expanded in one level before expanding all the nodes in the predecessor level. It is clear from the above discussion that the Limited-Depth BFS is similar to BFS in being an optimal and complete algorithm. Given the fact that the Limited-Depth BFS algorithm is controlled by the threshold value, the execution time required by this algorithm is $O(\beta^d)$, where β is the branching factor and d is the threshold value. Using this algorithm, the required memory space will also be $O(\beta^d)$.

The depth first search (DFS) [1, 12] is another well-known blind search algorithm. At each step of the DFS algorithm, one successor of the most recently expanded node is generated. When the last recently expanded node has no successors to be generated, or all of its successors are expanded, its predecessor generates the next successor. Unless the tree is exhausted, or the solution is found, the algorithm proceeds with similar next steps. Furthermore, a variation of DFS algorithm is the Limited-Depth DFS search [1], which specifies a threshold value on the maximum depth, to which the tree can be traversed. When a node at a depth equal to a given bound is reached, this node is treated as a leaf in the tree.

A combination of breadth-first and depth-first search that allows a single search algorithm to have the advantages of both is presented by Zhou and Hansen [13]. They showed the benefits of this combination using the treewidth problem.

Another type of search algorithms is the heuristic search (informed search) [1, 11, 12], which has further information about the cost of the path between any state in the search space and the goal state. The objective of heuristic search algorithms is to minimize the total expansion cost. Therefore, information about the cost of the path is obtained during the search process, and used to estimate which node in the search space has the highest probability of leading to the goal node. Consequently, expansion priorities of generated nodes will be varied.

Adaptive random search technique is an example of a heuristic search algorithm, which was proposed by Hamzaçebi and Kutay [14]. This new random search technique facilitates the determination of the global minimum. In addition, the applicability of the algorithm on artificial neural network training is tested with the *XOR* problem.

A-Star (A*) [1, 11, 12, 15] is a well-known heuristic search algorithm that selects the next node to be expanded using an evaluation function $f(n)$. For a node n, $f(n)$ is the sum of two functions, $g(n)$ and $h(n)$, where $g(n)$ is the cost of reaching n from the initial state, and $h(n)$ is the estimated cost to reach the goal node from n. The node that is selected to be expanded is the one with the lowest $f(n)$, since the lowest $f(n)$ value is more promising to lead to the solution. For a search process, the A* algorithm uses two lists, open and closed [15]; the nodes that are generated within the search process, but not expanded, are managed in the open list, while the closed list manages the expanded nodes during the search process. Because the A* algorithm expands the nodes with smaller $f(n)$ values before expanding the nodes with larger $f(n)$ values, it always finds the goal with the cheapest cost. This is why the A* algorithm is considered as an optimal algorithm. One drawback of this algorithm is the large required memory space, which grows exponentially with larger search spaces. This refers to the fact that all of the nodes, whose $f(n)$ values are smaller or sometimes equal to the $f(n)$ value of the goal node, are expanded using the A* algorithm, requiring more memory space to save all of the generated nodes. Moreover, some of the goal node's neighbours may be expanded before selecting the goal node [1]; this occurs when there are some nodes in the open list whose expansion priority is equal to that of the goal node. Therefore, some of these nodes may be expanded before the goal node itself, thus consuming more memory space to be saved, and more time for these nodes to be expanded and for their successors to be added to the open list.

Several parallel heuristic search algorithms were developed in the last few decades to solve the 15 puzzle problem, among which is the A* algorithm and its variations [9, 16–22]. One popular parallelization technique for the A* algorithm is called the distributed tree search approach [9, 21], which is based on dividing the A* search tree into equal-sized partitions which are distributed among processors using a specific tree partitioning technique. Then, each processor performs searching on its partition using the same threshold value.

Szer et al. [23] presented multi-agent A* (MAA*), which is a complete and optimal heuristic search algorithm for solving decentralized partially observable Markov decision problems with finite horizon. The MAA* algorithm can be used for computing optimal plans, such as multirobot coordination, network traffic control or distributed resource allocation, that operate in a stochastic environment.

Various parallel heuristic search algorithms have been developed over the years [9, 16–22, 24], but little work has been done on parallelizing heuristic search algorithms using multithreading [22, 24, 25]. Good performance is achieved by parallel heuristic search algorithms for very large problem sizes, but they are inefficient for small problem sizes, owing to communications overheads and poor processor utilization. The performance of such algorithms is also degraded by the difficulty of achieving balanced load among processors and the need for load balancing techniques within a search process.

The multithreaded approach makes communication between concurrent threads easier since a multithreaded algorithm can be designed and implemented on a machine, with one or more processors, and a shared memory between concurrent threads. A variable number of concurrent threads can be created at execution time since, in some applications, the number of required threads may be determined at the application's run time. Moreover, load balancing is not highly required unless the concurrent threads' loads are highly imbalanced. Such problems may occur in multithreaded heuristic search algorithms, where each thread receives a subtree of the search space, and trees may be of different sizes, hence producing an imbalanced load state between threads. A typical solution here comes from thread scheduler using, for example, a work-stealing policy [26, 27], whereby a thread that finished a subtree in a short time picks up some work from the list of another busy thread at the meantime. Another approach is over-decomposition [28], in which the number of used threads is larger than the number of existing processors (or cores). Other useful approaches, such as dynamic scheduling [29] can also be used. An example of this strategy is the self-scheduling of processors, which has been proven to achieve good load balancing. Another useful feature of multithreading is that, if threads are created once and then recycled to execute new work, the overhead would be reasonable and the utilization would be improved.

Mahafzah [22] presents a parallel multithreaded IDA* (PMIDA*) heuristic search algorithm using POSIX threads (Pthreads) and message-passing interface libraries running on 16 dual-core processors. The PMIDA* algorithm has been

evaluated analytically in terms of various performance metrics, including time complexity, space complexity, completeness and optimality. Moreover, the PMIDA* algorithm has been evaluated experimentally and compared with other parallel heuristic search algorithms, in terms of various performance metrics, including efficiency, number of generated nodes, search effectiveness, etc., on various applications, such as the asymmetric travelling salesman problem and vertex cover problem on a cluster of 16 dual-core processors.

Zhang and Hansen [24] proposed a parallel breadth-first heuristic search on a shared-memory architecture, where the parallelization is achieved using a multithreaded approach, in which layer synchronization has been used where threads expand all nodes in one layer of the breadth-first search graph before they are expanded in the next layer. However, nodes in any layer of the search graph are not prioritized; thus simplifying dynamic load balancing.

Zabatta [25] presented the Multithreaded Best First With Backtracking Search (MT-BFWBS) algorithm. In Zabatta's algorithm, a priority queue is shared among concurrent threads to save the nodes to be expanded. Nodes with maximum probability of leading to the solution are given higher priority. Using this algorithm, the root node is positioned in the priority queue before creating any thread. One thread is then created to pick the root node from the queue, and using a branching strategy, it expands the root node's successors, each of which is appended to the queue based on its priority, which is estimated through the heuristic function. Whenever the queue contains more than one node, the active thread creates another equivalent thread, which is assigned half the number of creator thread's nodes in the queue. The created thread executes the same instructions and can create other threads, taking into account that the number of concurrent threads should not exceed a predetermined limit. Since multiple threads work simultaneously, the number of expanded nodes increases and the queue becomes full quickly. Therefore, a backtracking strategy is used to remove the nodes with lower priorities from the queue. One advantage of this algorithm is the dynamic thread creation ability to create more threads as the problem size increases. Dynamic thread creation provides a control on the number of concurrently working threads. That is, it creates a large number of concurrent threads when the search process requires a large number of workers. Otherwise, a smaller number is created. Another advantage of this algorithm is that the allocated memory space is controlled using a backtracking strategy which removes the least priority nodes from the queue. The drawback of this algorithm, on the other hand, is the time wasted by the backtracking strategy, which is used many times during searching. In addition, the longer execution time is exhausted while expanding and generating more useless nodes during the search process [25].

3. Parallel multithreaded A* algorithm

The proposed parallel multithreaded A* heuristic search algorithm applies the sequential A* algorithm using multiple threads. A graphical representation of the proposed parallel multithreaded A* algorithm is shown in Fig. 1, in which two phases are developed. During the first phase, the tree is expanded to reach a specific level by executing the Limited-Depth BFS algorithm. The level's depth may be determined before starting the algorithm or dynamically at run time. A number of threads are created based on the number of generated nodes within each level and the number of required threads. This number should be large enough to ensure good parallelization and small enough to achieve efficient CPU utilization.

The target of the first phase is to create the *SubRoots* queue, which contains the generated nodes that are located in a predefined depth, determined by the Limited-Depth BFS algorithm. Two global lists, *OpenPhase1* and *ClosedPhase1*, are used in this phase, where *OpenPhase1* holds the nodes in the tree that are generated but has not been expanded, while *ClosedPhase1* holds the generated and expanded nodes in the tree. *ClosedPhase1* represents a table structure, which keeps the reversed path from each node in the *SubRoots* queue to the initial state. The main output of the first phase is the *SubRoots* queue which will be used as an input to the second phase.

The objective of the second phase is to divide the tree, built in phase one, into equal partitions. To achieve this purpose, multiple equivalent threads are created. For each node in the *SubRoots* queue, one thread is used to execute the sequential A* algorithm on its partition concurrently with other threads. Each thread has its own *Open* and *Closed* lists. Once any thread reaches the goal node, it instructs other threads to terminate by using the global flag *StopAll*, which is set to true if the goal is found. Otherwise, the search process ends when the tree is exhausted without achieving the goal by any of the working threads, in which case, the value of *StopAll* will be set to false.

The second phase of the proposed parallel multithreaded A* algorithm is shown in Fig. 2, in which, the loop in lines 1 and 2 represents the process of concurrent working threads' dynamic creation. For each node n_x in the *SubRoots* queue, a thread is created dynamically, at run time. Each created thread x is assigned a thread handler th_x, which represents the identifier of thread number x. Each thread executes the sequential A* algorithm concurrently with other threads and terminates when it completes its work. In other words, a thread completes its work when its subtree is exhausted, or when it finds the goal.

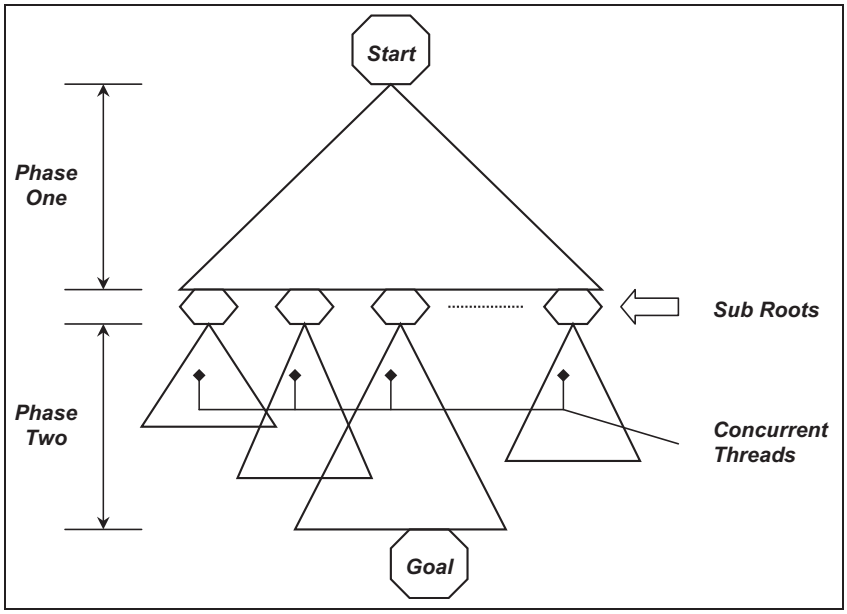

Figure 1. A graphical representation of the parallel multithreaded A* algorithm.

PhaseTwo(*SubRoots*):
For each node n_x in *SubRoots*
Do Create Thread (th_x, Sequential-A* (n_x))

Figure 2. Phase two of the parallel multithreaded A* algorithm.

4. Analytical evaluation

In this section, an analytical evaluation of the proposed parallel multithreaded A* heuristic search algorithm is presented using various evaluation metrics, including time complexity, space complexity, completeness and optimality, as shown in Theorems 1–4.

Theorem 1. Given that β is the branching factor, k is the depth of the shallowest solution and t is the number of threads, the time complexity of the parallel multithreaded A* algorithm is $O(\beta^k/t)$.

Proof 1. The time complexity of the proposed parallel multithreaded A* algorithm can be computed as the total time required by both the first and second phases. The first phase of the parallel multithreaded A* algorithm requires β^d, which is the time consumed by the Limited-Depth BFS algorithm performed during this phase, where d is the threshold value of the Limited-Depth BFS algorithm [1]. In the second phase, the sequential A* algorithm is executed by independent threads. Thus, the execution time required by phase two is the sequential A* algorithm's execution time divided by the number of concurrent threads. Given that the time required by the sequential A* algorithm is β^k [1], then the time of the second phase is $((\beta^k - \beta^d)/t)$. Therefore, the total time required by the proposed parallel multithreaded A* algorithm is equal to $\beta^d + ((\beta^k - \beta^d)/t)$. However, since d is very small, β^d is considered as a constant and thus ignored. As a result, the time complexity of the parallel multithreaded A* algorithm becomes $O(\beta^k/t)$. This is better than the time complexity of the sequential A* algorithm, which is $O(\beta^k)$.

Theorem 2. Given that β is the branching factor, k is the depth of the shallowest solution, and t is the number of threads, the space complexity of the parallel multithreaded A* algorithm is $O(t\beta^k)$.

Proof 2. The space complexity of the proposed parallel multithreaded A* algorithm can be expressed as the sum of the space complexities of both the first and second phases. The first phase of the parallel multithreaded A* algorithm, in

which the Limited-Depth BFS algorithm is executed, requires β^d memory space which is the required memory space for the Limited-Depth BFS algorithm, where d is the threshold value of the Limited-Depth BFS algorithm [1]. Moreover, the memory space required by phase two is that required by the sequential A* algorithm multiplied by the number of concurrent threads. This is because the sequential A* algorithm is executed independently by the created threads. Therefore, given that the memory space required by the sequential A* algorithm equals β^k [1], then the memory space for the second phase is equal to $t(\beta^k - \beta^d)$. Thus, a total of $\beta^d + t(\beta^k - \beta^d)$ of memory space is required by the proposed parallel multithreaded A* algorithm. However, since d is very small, β^d is considered as a constant and thus ignored. As a result, the space complexity of the parallel multithreaded A* algorithm becomes $O(t\beta^k)$. This is not as good as the space complexity of the sequential A* algorithm, which equals $O(\beta^k)$. However, both algorithms, the sequential and parallel multithreaded A*, keep all generated nodes in memory, but the parallel multithreaded A* algorithm generates more nodes than the sequential A* algorithm.

Theorem 3. The parallel multithreaded A* algorithm is a complete algorithm.

Proof 3. A search algorithm is complete if it always finds a solution, if the solution exists. The Limited-Depth BFS and the sequential A* algorithms are executed, respectively, during the first and second phases of the parallel multithreaded A* algorithm. Therefore, since the Limited-Depth BFS and the sequential A* algorithms are complete [1], then the proposed parallel multithreaded A* algorithm is also a complete algorithm.

Theorem 4. The parallel multithreaded A* algorithm is an optimal algorithm for admissible heuristics.

Proof 4. A search algorithm is optimal if it always finds the best solution, if the solution exists. In the first and second phases of the parallel multithreaded A* algorithm, the Limited-Depth BFS algorithm and the sequential A* algorithm are performed, respectively. Given that the Limited-Depth BFS algorithm is optimal and the sequential A* algorithm is optimal for admissible heuristics [1], then the proposed parallel multithreaded A* algorithm is also an optimal algorithm for admissible heuristics because all concurrent threads work under the same threshold value, which means that, if there exist more than one solution under a specific threshold value, then all these solutions are optimal since the solution was not found under the predecessor threshold value.

5. Experimental results and performance evaluation

The proposed parallel multithreaded A* algorithm uses the 15 puzzle problem [30] as a search domain, which is used by many heuristic search algorithms [9, 16–22, 24]. The 15 puzzle problem has a board consisting of 16 squares. Fifteen squares, numbered from 1 to 15, are scattered randomly, while the remainder square is a blank one. The goal of this problem is to place each numbered square in its correct position. In order to achieve this goal, the blank square is exchanged with one of its neighbouring squares – upper, lower, left or right squares – provided that only one movement is allowed per each step. The objective of heuristic search algorithms is to achieve this goal with the minimum number of steps.

The proposed parallel multithreaded A* algorithm is implemented using the multithreading library, POSIX threads (Pthreads). The Pthreads standard, specified by IEEE, is called 1003.1c – 1995 POSIX API [10]. It has become a popular standard for writing parallel programs owing to some attractive features, including inexpensive thread creation and termination, a large number of created threads for parallel programs and the ability to run a program written using Pthreads on both sequential and multiprocessor computers without changing the code. However, some issues should be taken into account when using Pthreads, such as Pthreads scheduling and synchronization to ensure the consistency of shared data [10, 31, 32].

The experimental runs were conducted on a Dual-Core Intel Processor (CPU 2.26 GHz), with 14 pipeline stages supported with Hyper-Threading Technology based on Simultaneous Multi-Threading (SMT), in which each core has dual SMT hardware threads. Therefore, the experimental runs were performed on a total of four SMT hardware threads. The experimental system is supported with 2 GB RAM, and 3 MB L2 Cache. Both algorithms, the sequential and parallel multithreaded A*, are implemented using C programming language, GNU C library version 2.3.x (glibc 2.3.x), running under SUSE® Linux operating system version 10.

The number of concurrent software-defined threads created during the execution of the proposed algorithm varies from seven to 120 sharing the four SMT hardware threads. Using the multithreaded approach, the number of instructions issued by independent software-defined threads, which can be executed simultaneously in a pipelined processor, is limited by the number of that processor's pipeline stages, denoted as *p_stages*. If the number of simultaneous instructions issued by the software-defined threads is not greater than *p_stages* [31, 33, 34], then all of the concurrent instructions

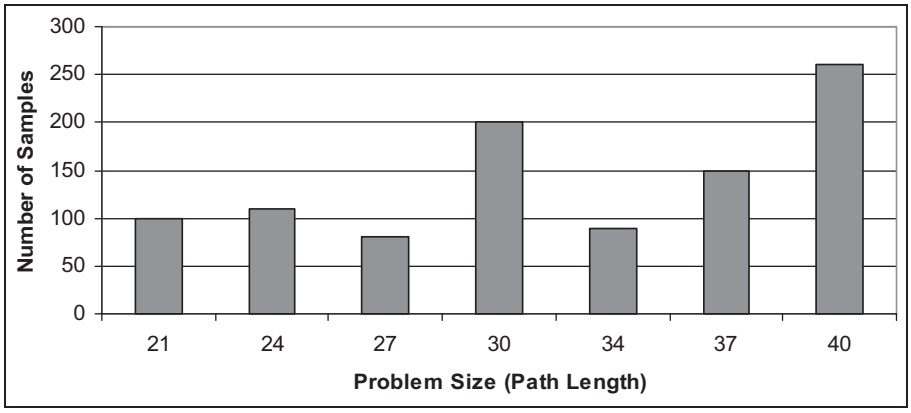

Figure 3. Data samples of the 15 puzzle problem.

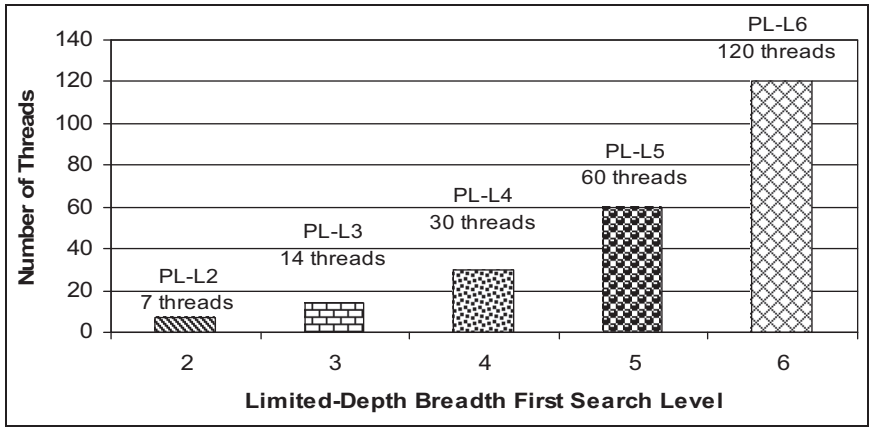

Figure 4. Run cases of the parallel multithreaded A* algorithm.

can be simultaneously executed in distinct stages of the pipelined processor. Otherwise, the *p_stages* instructions can be concurrently executed. Thus, all the concurrent instructions are scheduled to use the pipeline stages, such that, at a certain point of execution time, *p_stages* instructions may be executed, while the rest of instructions will be waiting to use the pipelined processor. A multithreaded architecture provides an optimized throughput of multiprogramming workloads rather than a single-thread performance [32, 35]; that is, the multithreaded architecture enhances instruction throughput by allowing instructions to be issued by several independent threads to multiple functional units in a single cycle [36].

In this work, about 1000 data samples with different data sizes (i.e. path length) were used for executing the experimental runs, as shown in Fig. 3. The number of concurrent threads for each run in the parallel multithreaded A* algorithm was specified by executing the Limited-Depth BFS algorithm in the first phase of the parallel multithreaded A* algorithm, as discussed in Section 3. In this work, for each sample, one run of the sequential A* algorithm was compared with five runs (PL-L2, PL-L3, PL-L4, PL-L5 and PL-L6) of the parallel multithreaded A* algorithm, where each run of the parallel algorithm was performed with five different levels (2–6), respectively, and each level had an average number of threads, as shown in Fig. 4. For example, the run case PL-L6, where PL stands for parallel multithreaded A* heuristic search algorithm and L6 stands for level six of the Limited-Depth BFS algorithm, had an average of 120 concurrently working threads, as shown in Fig. 4.

The experimental results of the sequential and parallel multithreaded A* algorithms on the 15 puzzle problem were evaluated using the following performance metrics: speedup, number of generated nodes, number of computed heuristic distances, number of expanded nodes, number of duplicate checks, number of duplicates found and search effectiveness for small and large problem sizes.

The speedup is defined as the ratio of the required execution time to solve a problem using the sequential A* algorithm to the execution time that is required to solve the same problem through the parallel multithreaded A* algorithm,

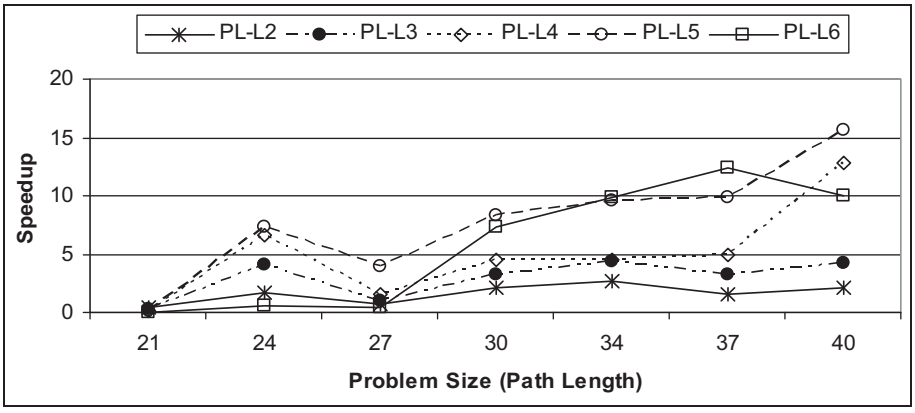

Figure 5. Speedup of the parallel multithreaded A* algorithm for all run cases.

as shown in equation (1). The execution time, in general, is the amount of time required by an algorithm to solve a problem. The sequential A* algorithm's execution time is the time consumed from the start of the search process until a solution is found. This metric is expressed in the parallel multithreaded A* algorithm as the amount of time required by the search process since the beginning of the first phase until the second phase of the algorithm is completed. Thus, the execution time includes the time that is required for creating and terminating concurrent threads in addition to the time of serializing the algorithm and synchronizing shared data.

$$\text{speedup} = \text{sequential time}/\text{parallel multithreaded time} \tag{1}$$

The speedup provides an evaluation of the performance improvement of the parallel multithreaded A* algorithm over the sequential A* algorithm [33]. Figure 5 presents the speedup results of the parallel multithreaded A* algorithm. It is clear from Fig. 5 that most run cases of the parallel algorithm achieve better performance than the sequential algorithm for most problem sizes and most run cases. For example, the speedup of the parallel multithreaded A* algorithm, for problem size 40, is more than the sequential A* algorithm by about 16 times for the run case PL-L5. The reason behind this increased speedup is that the sequential A* algorithm requires a lot of time because of not taking advantage of the multithreaded architecture, which enhances instruction throughput by issuing multiple instructions from multiple threads within one clock cycle. Moreover, the speedup of the parallel multithreaded A* algorithm is enhanced by an increased number of concurrent threads until reaching a problem of size 37, where the speedup of the case PL-L6 becomes larger than the speedup of all other run cases. Consequently, for large problem sizes, increasing the number of concurrent threads to more than 60 does not improve the performance of the search process. Thus, such an increase adds an overhead for threads management. However, for PL-L4 and PL-L5 run cases, the speedup increases for large problem sizes such as 40 since the number of threads between 30 and 60 does not add an overhead compared with that suffered by a number of threads over 60.

Generated nodes are those that are generated through the search process, including both generated and ignored nodes. A node is ignored if it is a duplicate. The number of nodes generated by a search algorithm is a measurement of the searching throughput, which is defined as the number of nodes generated during a period of time [34]. The number of generated nodes depends on the branching factor, given that the branching factor of the 15 puzzle problem is 2.13 for both the sequential A* and the parallel multithreaded A* algorithms. The number of generated nodes is also affected by the problem's execution time, where the number of generated nodes increases for an increasing execution time.

Figure 6 shows the number of generated nodes for both the sequential and parallel multithreaded A* algorithms. The figure reveals that the parallel multithreaded A* algorithm generates larger numbers of nodes than the sequential A* algorithm for most problem sizes. For example, for a problem of size 37, the parallel algorithm generates about 1.3 million nodes for the PL-L6 run case, whereas the sequential algorithm generates about 0.25 million nodes for the same problem size. It is also shown that, for the parallel multithreaded A* algorithm, the number of generated nodes increases for a larger number of concurrent threads. The reason for that is that each thread works independently, so each thread generates its own nodes. Therefore, more nodes are generated for an increasing number of concurrent threads. However, the search space partition is not completely disjoint. Therefore, generating more nodes does not always mean that more of the state space was searched; it could mean that the same nodes were visited by different threads.

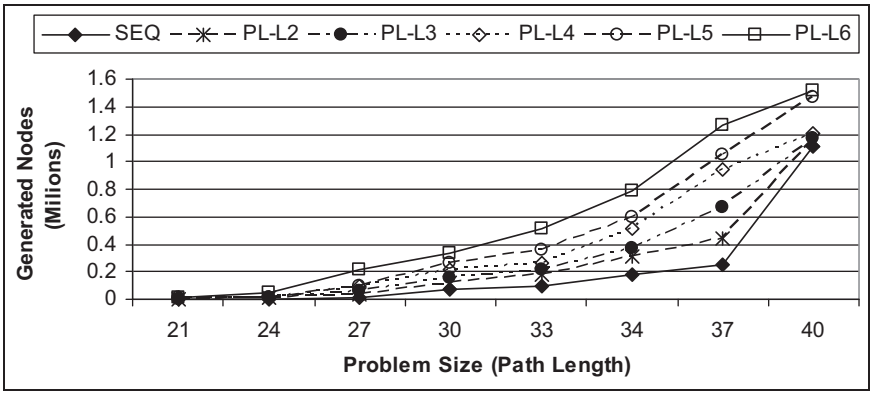

Figure 6. Number of generated nodes for sequential and parallel multithreaded A* algorithms.

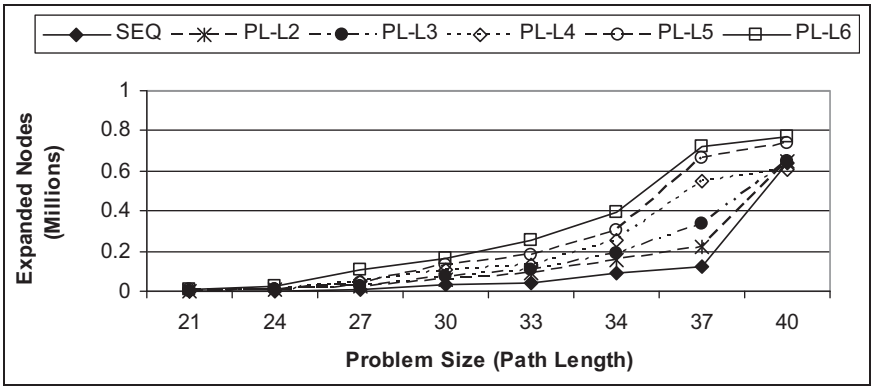

Figure 7. Number of expanded nodes for sequential and parallel multithreaded A* algorithms.

The number of computed heuristics is defined as the number of times a heuristic distance is computed. For each new generated node, the heuristic distance between the new generated node and the goal node is computed. Also, for each generated node, the value of the evaluation function $f(n)$ is computed before the duplicate check process is performed. As discussed in Section 2, the heuristic distance $h(n)$ from a node to the goal node is one of the evaluation function $f(n)$ parameters. Thus, the heuristic distance $h(n)$ is required to be computed for each new generated node. This means that, for each problem, the number of heuristic distances that are computed is equal to the number of generated nodes. Therefore, the experimental results for the number of heuristic distances computed are equivalent to the experimental results for the number of generated nodes, which is depicted in Fig. 6.

Figure 6 also shows that the parallel multithreaded A* algorithm computes a larger number of heuristic distances than the sequential A* algorithm because a large number of nodes are generated in the case of the parallel multithreaded A* algorithm. For instance, for a problem of size 37, the number of computed heuristic distances for the parallel algorithm, PL-L6 run case, is about 1.3 million, whereas for the sequential algorithm it is about 0.25 million.

Expanded nodes are those that are generated and expanded during the search process. Thus, the number of expanded nodes is closely related to the number of generated nodes, where the number of expanded nodes increases as more nodes are generated.

Figure 7 shows the number of expanded nodes for both sequential and parallel multithreaded A* algorithms. It is shown by this figure that the parallel multithreaded A* algorithm expands a larger number of nodes than the sequential A* algorithm for most problem sizes. For example, for problem size 37, the parallel algorithm expands about 0.7 million nodes for the PL-L6 run case, whereas the sequential algorithm expands about 0.1 million nodes for the same problem size. It is also shown by the figure that, for the parallel multithreaded A* algorithm, an increasing number of concurrent threads causes an incremented number of expanded nodes. This is clarified by the independent threads' work, so each thread generates and expands its own nodes.

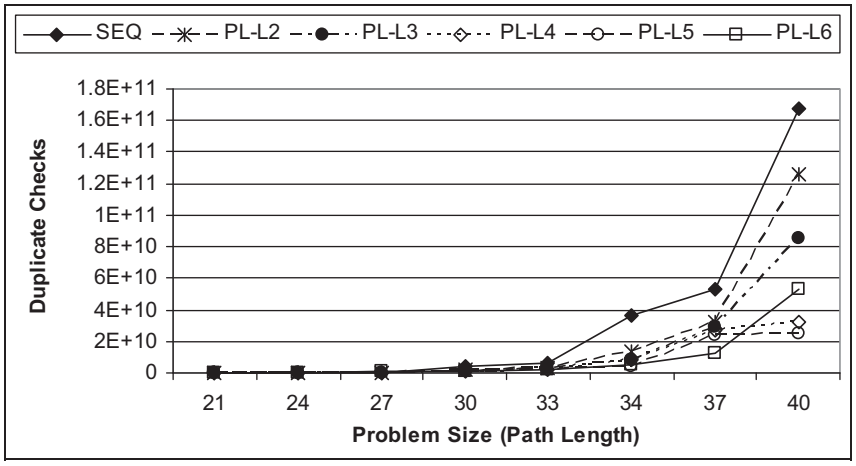

Figure 8. Number of duplicate checks for sequential and parallel multithreaded A* algorithms.

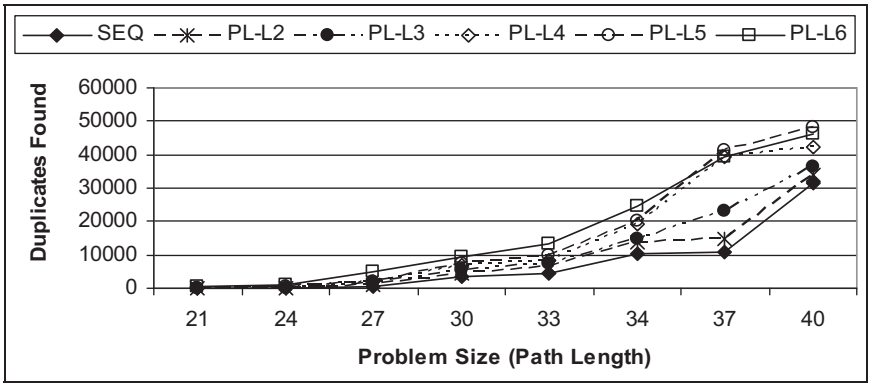

Figure 9. Number of duplicates found for sequential and parallel multithreaded A* algorithms.

A duplicated node is a node that is regenerated by the search process. Thus, a duplicate check process is required for each generated node. In this process, the new generated node is compared with all the nodes in the *Open* list, in which the nodes that are located in the search path are saved with their neighbour nodes. If a similar node is found, the new generated node is considered as a duplicated node. The duplicate check process, in general, is not complete in the algorithms that use the DFS strategy since a duplicated node may be generated, but not found in the *Open* list because the equivalent node is located in another path in the tree, which either has been pruned or has not been generated yet.

In the parallel multithreaded A* algorithm, each thread performs the duplicate check process only on its own *Open* list, while the *Open* lists of other threads are not searched to avoid the overhead of the concurrent lists access synchronization. The number of performed duplicate checks is affected by two factors, one of which is the number of generated nodes, where more duplicate checks are performed for an increasing number of generated nodes. The other affecting factor is the length of the search path, such that, as the search path becomes longer, more duplicate checks are required.

Figure 8 shows the number of duplicate checks for the sequential and the parallel multithreaded A* algorithms. As shown in the figure, the parallel multithreaded A* algorithm computes a fewer number of duplicate checks than the sequential A* algorithm for most problem sizes. For example, the number of duplicate checks in the sequential algorithm for a problem of size 40 is about 1.7×10^{11}, whereas for the parallel algorithm, PL-L6 run case, it is about 5×10^{10}. Also, for the parallel multithreaded A* algorithm, the number of duplicate checks is inversely related to the number of concurrent threads, where for an increasing number of concurrent threads, the subtree which is generated by a thread becomes smaller, and thus the number of duplicate checks is decreased within each subtree.

A duplicated node that is generated and found in the search path by the duplicate check process is more likely to be found as the length of the search path increases since for long paths more nodes will be in the *Open* list. The number of duplicated nodes found by the sequential and the parallel multithreaded A* algorithms is presented in Fig. 9. As can be

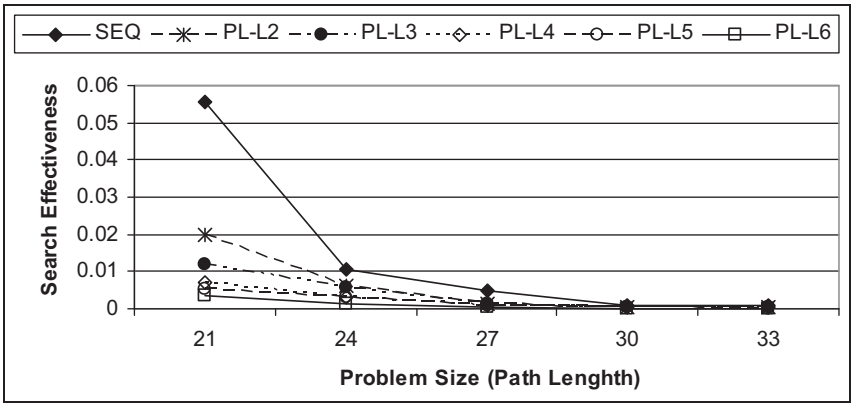

Figure 10. Search effectiveness for small problem sizes.

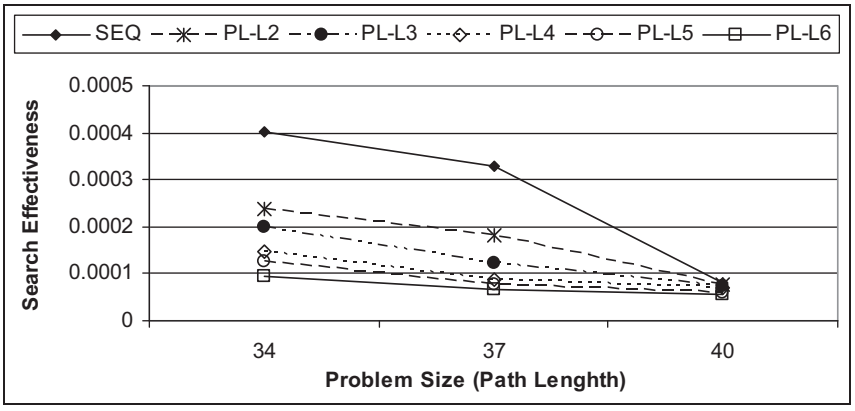

Figure 11. Search effectiveness for large problem sizes.

seen from this figure, the parallel multithreaded A* algorithm finds larger numbers of duplicated nodes than the sequential A* algorithm for most problem sizes. For instance, the number of duplicate nodes found for a problem of size 37 using the parallel algorithm PL-L6 run case is about 40,000, whereas for the sequential algorithm it is about 11,000. Regarding the parallel multithreaded A* algorithm, the number of duplicated nodes found is augmented for an increasing number of concurrent threads, which raises the possibility of generating duplicated nodes among concurrent threads.

The search effectiveness evaluation metric measures the closeness of the search effort of a searching algorithm towards the goal node. For a given problem x, with branching factor β, the search effectiveness *Effectiveness*(x) is defined in equation (2) [34], where l is the length of the path obtained by the search process, and η is the number of generated nodes.

$$\text{effectiveness}(x) = \beta * l/\eta \tag{2}$$

The search effectiveness of both the sequential and the parallel multithreaded A* algorithms, for small problem sizes is shown in Fig. 10, whereas Fig. 11 shows the search effectiveness of both algorithms for large problem sizes. Both figures show that the search effectiveness of the parallel multithreaded A* algorithm is less than the search effectiveness of the sequential A* algorithm, since more nodes are generated using the parallel multithreaded A* algorithm. Also, the search effectiveness of the parallel multithreaded A* algorithm becomes lower for a larger number of concurrent threads, since more nodes are generated as the number of concurrent threads increases.

6. Conclusions and future work

Several research efforts have been dedicated to the design and development of efficient heuristic search algorithms, and have produced various algorithms applied on both sequential and parallel computers. However, the limited throughput

produced by sequential computers and the expensive processor communication cost in parallel computers have motivated the use of the multithreaded approach combining the low cost of sequential computers and the high throughput of parallel computers.

Multithreading improves the CPU throughput by creating a variable number of concurrent threads at the application's run time. In addition, a variable number of threads can be executed simultaneously on one shared memory, thus improving the execution speed and making the communication between concurrent threads easier.

In this paper, the parallel multithreaded A* heuristic search algorithm has been presented and implemented using the POSIX threads (Pthreads) library. An analytical evaluation of the proposed algorithm has been shown in terms of time complexity, space complexity, completeness and optimality. The experimental results have been demonstrated and discussed. The parallel multithreaded A* heuristic search algorithm has been evaluated and compared against its sequential counterpart on the 15 puzzle problem as a search domain for a number of performance metrics, including speedup, number of generated nodes, number of computed heuristic distances, number of expanded nodes, number of duplicate checks, number of duplicates found and search effectiveness. The results proved the better performance of the parallel multithreaded A* heuristic search algorithm over the sequential A* heuristic search algorithm in terms of different performance metrics. For example, it has been shown analytically that the time complexity of the proposed parallel multithreaded A* algorithm is better than that of the sequential A* algorithm. In particular, the time complexity of the parallel multithreaded A* algorithm is $O(\beta^k/t)$, whereas the time complexity of the sequential A* algorithm is $O(\beta^k)$, where β is the branching factor, k is the depth of the shallowest solution and t is the number of threads. Another experimental example shows that, for a problem of size 40, the parallel multithreaded A* algorithm for PL-L5 run case achieved about 16 times better speedup of the sequential A* algorithm. Moreover, for a problem of size 37, the parallel multithreaded A* algorithm for PL-L6 run case generated about 1.3 million nodes and expanded to nearly 0.7 million nodes, whereas the sequential A* algorithm generated about 0.25 million nodes and expanded to nearly 0.1 million nodes, which means that the parallel multithreaded A* algorithm produces better searching throughput than the sequential A* algorithm.

In future work, the parallel multithreaded A* algorithm can be implemented using other multithreading libraries, such as Java threads. However, the algorithm's performance may differ with different thread libraries. Furthermore, it would be interesting to apply the proposed parallel multithreaded A* algorithm on different search problems and applications, such as the travelling salesman problem and web search engines.

Acknowledgement

The author would like to express his deep gratitude to the anonymous referees for their valuable comments and helpful suggestions, which improved the paper.

Funding

This research received no specific grant from any funding agency in the public, commercial or not-for-profit sectors.

References

[1] Russell S and Norvig P. *Artificial intelligence: A modern approach*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall, 2003.
[2] Chen K. Heuristic search and computer game playing IV. *Information Sciences* 2005; 175(4): 245–246.
[3] Pedrycz W and Vasilakos A. *Computational intelligence in telecommunications networks*, 1st edn. Boca Raton, FL: CRC Press, 2000.
[4] Tsai C-F, Tsai C-W and Tseng C-C. A new hybrid heuristic approach for solving large traveling salesman problem. *Information Sciences* 2004; 166(1–4): 67–81.
[5] Perry S and Willett P. A review of the use of inverted files for best match searching in information retrieval systems. *Journal of Information Science* 1983; 6(2–3): 59–66.
[6] Frants V, Shapiro J and Voiskunskii V. Optimal search available to an individual user. *Journal of Information Science* 1996; 22(3): 181–191.
[7] Singer G, Norbisrath U and Lewandowski D. Ordinary search engine users carrying out complex search tasks. *Journal of Information Science* 2013; 39(3): 346–358.
[8] Chioua S-W. An efficient search algorithm for road network optimization. *Applied Mathematics and Computation* 2008; 201(1–2): 128–137.
[9] Al-Ayyoub A. Performance evaluation of parallel iterative deepening A* on clusters of workstations. *Performance Evaluation* 2005; 60(1–4): 223–236.
[10] Grama A, Gupta A, Karypis G and Kumar V. *Introduction to parallel computing*, 2nd edn. Reading, MA: Addison Wesley, 2003.

[11] Valtorta M. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences* 1984; 34(1): 47–59.
[12] Jones T. *Artificial intelligence: A systems approach*, 1st edn. Sudbury, MA: Jones and Bartlett Publishers, 2009.
[13] Zhou R and Hansen E. Combining breadth-first and depth-first strategies in searching for tree width. In: Kitano H (ed.) *Proceedings of the 21st international joint conference on artificial intelligence*. San Francisco, CA: Morgan Kaufmann, 2009, pp. 640–645.
[14] Hamzaçebi C and Kutay F. A heuristic approach for finding the global minimum: Adaptive random search technique. *Applied Mathematics and Computation* 2006; 173(2): 1323–1333.
[15] Hart P, Nelson N and Raphael B. A formal basis for the heuristic determination of the minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 1968; 4(2): 100–107.
[16] Bauer B. The Manhattan pair distance heuristic for the 15-puzzle. Technical Report PC^2/TR 001 94, Paderborn Center for Parallel Computing, University of Paderborn, Germany, 1994.
[17] Cook D and Varnell R. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research* 1998; 9(1): 139–166.
[18] Powley C and Korf R. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1991; 13(5): 466–477.
[19] Duwairi R, Mahafzah B and Al-Ayyoub A. A Framework for performance assessment of parallel bi-directional heuristic search. In: *Proceedings of the international conference on artificial intelligence*. Las Vegas, NV: CSREA Press, 2002.
[20] Korf R and Felner A. Disjoint pattern database heuristics. *Artificial Intelligence*. 2002; 134(1–2): 9–22.
[21] Rao VN and Kumar V. Parallel depth-first search. Part I: Implementation. *International Journal of Parallel Programming* 1987; 16(6): 479–499.
[22] Mahafzah B. Parallel multithreaded IDA* heuristic search: Algorithm design and performance evaluation. *International Journal of Parallel, Emergent and Distributed Systems* 2011; 26(1): 61–82.
[23] Szer D, Charpillet F and Zilberstein S. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In: *Proceedings of the twenty-first conference on uncertainty in artificial intelligence*, Edinburgh, 2005, pp. 576–583.
[24] Zhang Y and Hansen E. Parallel breadth-first heuristic search on a shared-memory architecture. In: *Workshop on heuristic search, memory-based heuristics and their applications*. Boston, MA: The Association for the Advancement of Artificial Intelligence, 2006.
[25] Zabatta F. *Multithreaded constraint programming and applications*. PhD thesis, University of New York, New York, 1999.
[26] Blumofe R and Leiserson C. Scheduling multithreaded computations by work stealing. In: *Proceedings of the 35th annual IEEE conference on foundations of computer science*, Santa Fe, NM, 1994.
[27] Lu W and Gannon D. Parallel XML processing by work stealing. In: *Proceedings of the 2007 workshop on service-oriented computing performance: Aspects, issues, and approaches*, Monterey, California, 2007.
[28] Bongo L, Vinter B, Anshus O, Larsen T and Bjorndalen J. Using over decomposition to overlap communication latencies with computation and take advantage of SMT processors. In: *2006 international conference on parallel processing workshops*, 2006.
[29] Devine K, Boman E, Heaphy R, Hendrickson B, Teresco J, Faik J et al. New challenges in dynamic load balancing. *Applied Numerical Mathematics* 2005; 52(2–3): 133–152.
[30] Hayes R. The Sam Loyd 15-puzzle, http://www.cs.tcd.ie/publications/tech-reports/reports.01/TCD-CS-2001-24.pdf (2001, accessed August 2013).
[31] Hans J. Threads cannot be implemented as a library. In: *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, Chicago, IL, 2005.
[32] Mahafzah B. Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *Journal of Supercomputing* 2013; 66(1): 339–363.
[33] Hennesy J and Patterson D. *Computer architecture: A quantitative approach*, 3rd edn. San Francisco, CA: Morgan Kaufmann, 2003.
[34] Wilkenson B. *Computer architecture: Design and performance*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall, 1996.
[35] Ungerer T, Robic B and Silc J. Multithreaded processors. *The Computer Journal* 2002; 45(3): 320–348.
[36] Zang C, Imai S, Frank S and Kimura S. Issue mechanism for embedded simultaneous multithreading processor. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 2008; E91-A(4): 1092–1100.