https://doi.org/10.17576/apjitm-2021-1001-02
Asia-Pacific Journal of Information Technology and Multimedia
Jurnal Teknologi Maklumat dan Multimedia Asia-Pasifik
Vol. 10 No. 1, June 2021: 17 - 26
e-ISSN: 2289-2192

Received: 15 September 2020
Accepted: 19 February 2021
Published: 1 June 2021

BI-DIRECTIONAL MONTE CARLO TREE SEARCH

KRISTIAN SPOERER

ABSTRACT

This paper describes a new algorithm called Bi-Directional Monte Carlo Tree Search. The essential idea of Bi-directional Monte Carlo Tree Search is to run an MCTS forwards from the start state, and simultaneously run an MCTS backwards from the goal state, and stop when the two searches meet. Bi-Directional MCTS is tested on *8-Puzzle* and *Pancakes Problem*, two single-agent search problems, which allow control over the optimal solution length *d* and average branching factor *b* respectively. Preliminary results indicate that enhancing Monte Carlo Tree Search by making it Bi-Directional speeds up the search. The speedup of Bi-directional MCTS grows with increasing the problem size, in terms of both optimal solution length *d* and also branching factor *b*. Furthermore, Bi-Directional Search has been applied to a Reinforcement Learning algorithm. It is hoped that the speed enhancement of Bi-directional Monte Carlo Tree Search will also apply to other planning problems.

Keywords**:** Single Agent Search, Monte Carlo Tree Search, Bi-Directional Search, 8-Puzzle, Pancakes Problem, Reinforcement Learning.

INTRODUCTION

The shortest path problem consists of finding the path through a graph which minimizes the total edge cost. In 1959, Djikstra (Dijkstra, 1959) proposed an algorithm for finding the shortest path. The algorithm expands each node, starting from the start node, according to the cost from the start to node *n*, called *g(n)*. The next node with minimum g value of all found so far is expanded until the goal node is reached.

In 1966 Nicholson proposed Bi-directional Search (Nicholson, 1966) to enhance shortest path search. If Djikstra's algorithm performs a unidirectional search to find the shortest path of length *d*, where the average number of edges from a node is *b*, then the search will expand $O(b^d)$ nodes. Bi-directional Search runs forwards from the start and backwards from the goal. Each direction in Bi-Directional Search hopefully expands $O\left(b^{d/2}\right)$ nodes, and the sum of the two directions is less than the time required of a full Djikstra Search.

In 1968 a new algorithm, called A*, extended Djikstra's algorithm in a different way, by expanding the node with the smallest f value, such that $f(n) = g(n) + h(n)$ (Hart, 1968). The new term $h(n)$ is a domain heuristic underestimate of the distance to the goal node from node *n*. The heuristic directs the search towards a reasonable direction of the goal node, and therefore reduces the number of expanded nodes compared to Djikstra's algorithm. Let C^* be the g value of the goal node at the end of the shortest path, i.e. cost of the shortest path. Since A* expands the node with smallest f value, and because h is an underestimate of the actual

cost, the first node *n* which has $f = g = C^*$ will be the first goal node encountered which is also on the shortest path.
(Pohl, 1969) proposed a combination, Bi-directional A*, which led to investigations as to whether the forwards and backwards searches would actually meet in the middle. Eventually, (Holte et al, 2017) proposed a new Bi-Directional A* Search algorithm that is guaranteed to meet in the middle called MM. MM expands the next node with the lowest priority according to two *f* values,

$$f_F = max\big(g_F(n) + h_F(n), 2g_F(n)\big)$$

in the forwards direction and

$$f_B = max\big(g_B(n) + h_B(n), 2g_B(n)\big)$$

in the backwards direction. Here g_F is the forwards cost from start to node *n*, h_F is an underestimate of the cost from node *n* to the goal in the forwards direction, g_B is the backwards cost from goal to node *n*, and h_B is an underestimate of the cost from node *n* to the start backwards. Any node *n* which has $g_F > \frac{C^*}{2}$ will have a $f_F > C^*$ because h_F is an underestimate, which means the MM algorithm will encounter the backwards frontier before expanding any node *n* which has a $g_F > \frac{C^*}{2}$. The logic works analogously in the backwards direction, and therefore MM will meet in the middle.

The current paper describes a new extension of Bi-directional heuristic search which is based on Monte Carlo Tree Search (Coulom, 2006) (Kocsis, 2006). MCTS was highly influential in the development of strong Go playing programs. For Go the main challenge was extending the αβ algorithm. There was no known way of formalizing heuristic information about the value of a given Go position. Monte Carlo Tree Search uses statistical information from many simulated games to evaluate a Go position instead. (Gelly, 2008) reports "the first program to achieve human master level" in 9x9 Computer Go. Eventually, a combination of Monte Carlo Tree Search with deep neural networks trained by supervised learning and Reinforcement Learning defeated the human grand-master of Go (Silver et al, 2016). Afterwards Monte Carlo Tree Search showed promising results in non-game applications (see (Browne et al, 2012), (Goh et al, 2019), (Matsumoto et al, 2010)).

The new algorithm proposed in this paper is called Bi-directional Monte Carlo Tree Search (Bi-directional MCTS). It can be viewed as a hybrid combining Bi-directional search with Monte Carlo Tree Search (MCTS), and can also be viewed as an enhancement of Monte Carlo Tree Search. The essential idea of Bi-directional MCTS is to run MCTS forwards from the start state, and simultaneously run MCTS backwards from the goal state, and stop when the two searches meet. Bi-directional MCTS is compared with MCTS for optimally solving *8-Puzzle* and *Pancakes Problem*, two single-agent search problems, and the speedup of the Bi-directional enhancement is analyzed. The motivation for the present analysis is to learn how the Bi-directional MCTS scales to larger problems.

METHODS

BI-DIRECTIONAL MONTE CARLO TREE SEARCH

This section of the paper describes the original MCTS algorithm, how it can be modified to solve single-agent search problems, and also how MCTS has been expanded to be Bi-directional MCTS.

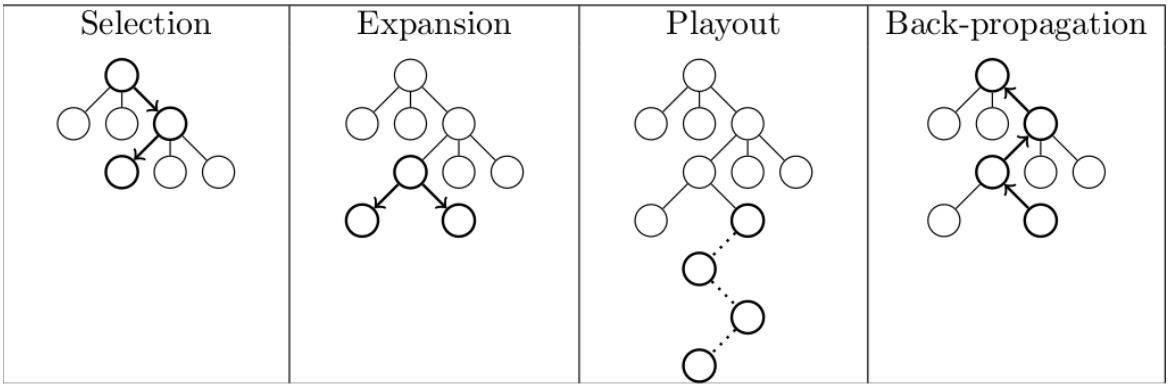

FIGURE 1. the four operators of the standard Monte Carlo Tree Search algorithm.

Monte Carlo Tree Search (see Figure 1) constructs a tree, starting from a single root node which grows as the search iterations proceed. One iteration of the Monte Carlo Tree Search algorithm applies four operators

1. **Selection** - repeatedly select the best of the children nodes, typically using UCT (Kocsis, 2006), until a leaf node in the tree is reached.
2. **Expansion** - optionally add the children of the leaf node, using one of several heuristics (Yajima et al, 2010), into the tree.
3. **Playout** - play from the leaf node of the tree, typically using pure random moves or guided moves, until the end of the game.
4. **Backpropagation** - update the statistics for each visited node in the tree.

The work in (Schadd et al, 2008) details an application of MCTS to a single player game called SameGame. This present work proposes an alternative design of MCTS for Single Agent Search. The purpose of the present study is not a comparison with the Single Player MCTS in (Schadd et al, 2008), but rather as a way to make MCTS work as a Bi-Directional algorithm.

Standard Monte Carlo Tree Search can be adapted to single agent search problems like *8-Puzzle* and *Pancakes Problem* by modifying two of the MCTS operators as follows. The **selection** operator stops when the tree depth reaches *solution_length_limit* (The *solution_length_limit* parameter effectively limits the search, which is essential because solutions to *8-Puzzle* can form loops and might never reach a terminal state.). The **playout** operator first checks if the leaf in the tree is a solved game state and returns a special *SOLVED_IN_TREE* flag, otherwise, a playout plays pure random moves, if the resulting game state is solved returns 1 (success) and if the total playout length is *solution_length_limit* returns 0.

In the new proposed algorithm Bi-directional MCTS, two trees are kept in memory, an up-tree and a down-tree. The four MCTS operators are applied in sequence to the down-tree first and then applied in sequence to the up-tree. See Figure 2 for the pseudo-code of the proposed algorithm. Simple logic is added to the **playout** operator; If the leaf of the current tree is found in the opposing tree then a special *MET_IN_THE_MIDDLE* flag is returned, otherwise a playout is performed as described above.

Algorithm 1 Bi-Directional Monte Carlo Tree Search

```
1:  procedure BDMCTS
2:      while resource_remaining do
3:          down_leaf ← Selection(down_root)
4:          down_leaf ← Expansion(down_leaf)
5:          down_result = Playout(down_leaf, up_root)
6:          if MET_IN_MIDDLE or SOLVED_IN_TREE then
7:              break
8:          Backpropagation(down_leaf, down_result)
9:
10:         up_leaf ← Selection(up_root)
11:         up_leaf ← Expansion(up_leaf)
12:         up_result = Playout(up_leaf, down_root)
13:         if MET_IN_THE_MIDDLE then
14:             break
15:         Backpropagation(up_leaf, up_result)
16:
17:     solution ← RebuildSolution()
18:
```

FIGURE 2. Pseudo-code of the proposed Bi-directional MCTS algorithm.

TEST PROBLEMS: *8-PUZZLE* AND *PANCAKE PROBLEM*

This section of the paper describes *8-Puzzle* and *Pancakes Problem*, the two single-agent search problems which are used to test the proposed Bi-directional MCTS algorithm. The problems were chosen for testing because they are simple to implement, both have a well known algorithm (A*) and admissible heuristics for producing optimal solutions, and MCTS and Bi-directional MCTS can easily be modified for solving both problems. Additionally, the *Pancakes Problem* is useful for modifying the branching factor *b* easily by increasing the number of pancakes *N* on the stack.

1	2	3
4	5	6
7	8	

5		2
4	6	3
1	7	8

FIGURE 3. The *8-Puzzle* problem solved state and a solvable starting state.

8-Puzzle is a famous toy. The puzzle can start in any solvable non-solved state, for example a configuration similar to that shown right hand side of Figure 3, which can be optimally solved using the move sequence (LEFT, UP, RIGHT, RIGHT, DOWN, LEFT, UP, RIGHT, UP, LEFT, DOWN, DOWN, RIGHT, UP, LEFT, UP, LEFT). The *8-Puzzle* is played by sliding a numbered tile next to the empty space into the empty space, thereby also

moving the empty space. The puzzle is solved when it eventually appears like that shown in the left hand side of Figure 3.

0	*
1	***
2	*****
3	*******
4	*********
5	**********
6	**********
7	**********

0	*******
1	*********
2	**********
3	***
4	*
5	**********
6	*****
7	**********

FIGURE 4. The *Pancake Problem* solved state and possible starting state.

The *Pancake Problem* is a famous search problem consisting of a stack of *N* pancakes of unique sizes. The starting state can be any non-solved state like that shown on the right hand side of Figure 4. A flip reverses the order of all pancakes on the stack from that position upwards. The state shown in the right hand side of Figure 4 can be optimally solved using the flip sequence (5, 6, 3, 6, 7, 6). The problem is solved when the pancakes are stacked in order like that shown in the left hand side of Figure 4.

EXPERIMENTAL SETUP

Total of 5,000 random *8-Puzzle* problems were generated by applying 25 random moves backwards from the solved *8-Puzzle* state, and 1,000 random *Pancake problem* states were generated by applying *N* random moves (*N* being the number of pancakes on the stack) backwards from the solved *Pancake Problem* state.

For each random *8-Puzzle* and *Pancake Problem* starting state, A* was run to solve the problem optimally, and then MCTS and also Bi-directional MCTS were run to find a solution. The solutions produced by MCTS and Bi-directional MCTS are compared with the optimal A* solution. Puzzles with optimal solution length 1 were discarded from the analysis, and also puzzles which weren't optimally solved by both MCTS and Bi-directional MCTS algorithms. Essentially, the current analysis only considers the amount of iterations required to produce the optimal *8-Puzzle* and *Pancake Problem* solution.

The analysis concerns the speed enhancement of the Bi-directional enhancement of MCTS, so a measurement of $speedup$ was calculated using the formula

$$speedup = \frac{playouts_{MCTS}}{playouts_{BDMCTS}} \quad (1)$$

where $playouts_{MCTS}$ and $playouts_{BDMCTS}$ are the number of playouts required to optimally solve the *8-Puzzle* and *Pancakes Problem* problems.

The parameters used for the experiments are shown in Figure 5. *Solution_length_limit* stops **selection** and **playout** during MCTS and Bi-directional MCTS, and ensures theoretically the tree and playout are deep enough solve the puzzle.

TABLE 1. The parameter settings used for the experiments

Puzzle Generation	
num_moves (8-Puzzle)	25
num_moves (Pancake Problem)	Number of pancakes N
A*	
$solution\_length\_limit$	num_moves
Monte Carlo Tree Search	
$iteration\_limit$	2,000,000
Bi-directional Monte Carlo Tree Search	
$iteration\_limit$	200,000
MCTS and BDMCTS	
selection operator	UCT, $c = 1.0$
expansion operator	expand when $leafvisits = numsiblings * 10$
$solution\_length\_limit$	num_moves

In this analysis the Bi-directional MCTS algorithm was implemented in sequence, not in parallel. All puzzles and algorithms were implemented using C++. Experiments were run on Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz, 64 bits, with 3.6G RAM.

RESULTS

In this section is presented the experimental results, first for *8-Puzzle* and then afterwards for *Pancakes Problem*.

Table 2 shows the relevant statistics for the *8-Puzzle* analysis, including average speedup, number of samples include in the analysis, the standard deviation, and the 95% confidence interval which is calculated as:

$$conf = 2 * \frac{stdev}{\sqrt{Samples}} \qquad (2)$$

TABLE 2. Statistics for *8-Puzzle*.

Solution length	3	5	7	9	11	13
Speedup	2.43	1.85	2.19	3.75	9.93	25.68
Samples	666	808	1010	845	604	336
Stdev.	0.65	0.43	0.72	1.54	4.65	9.88
95% conf. int.	0.05	0.03	0.04	0.10	0.37	1.07

Figure 7 shows optimal solution length l plotted against the average speedup of Bi-directional MCTS vs MCTS over the runs that are included in the analysis. The error bars represent the 95% confidence interval. Also plotted is the curve

$$y = 0.02644031 * exp(0.52552977 * x) + 1.1205256 \quad (3)$$

which was fitted to the experimental data using scipy curve_fit, which performs a non-linear least squares fit to the data.

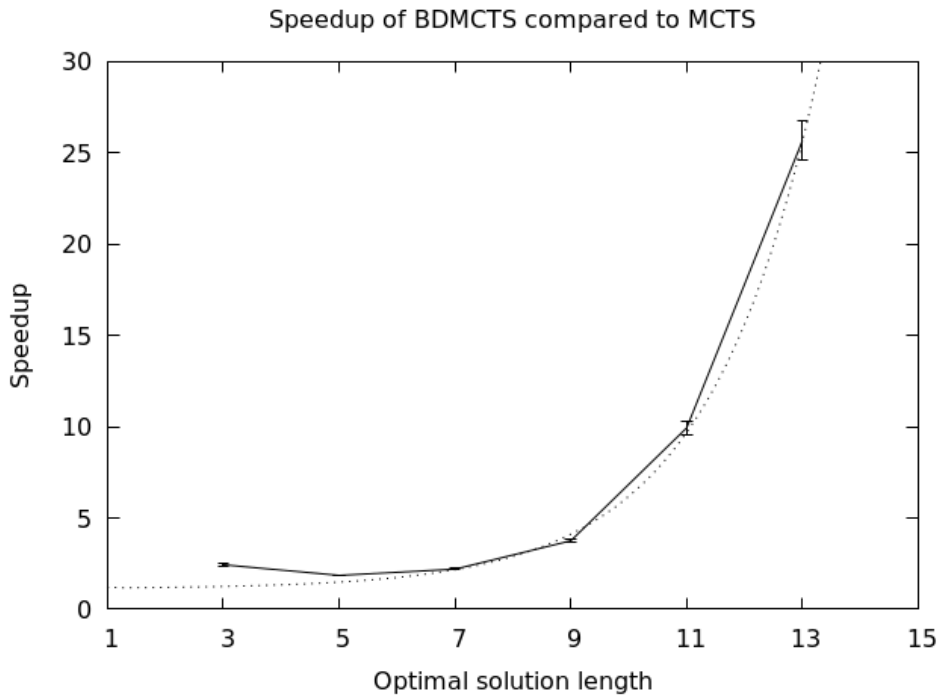

FIGURE 5. The speedup of BDMCTS compared to MCTS for *8-Puzzle* solutions of increasing optimal solution length.

In the case of *Pancake Problem* the analysis was performed on varying number of pancakes N where $6 \leq N \leq 9$, and for solutions of length 6 only. Table 3 shows the relevant statistics for the *Pancake Problem* analysis.

TABLE 3. Statistics for *Pancake Problem*.

Pancakes (N)	6	7	8	9
speedup	21.13	43.91	65.75	112.35
Samples	32	108	297	232
Stdev.	3.87	8.73	13.94	20.34
95% conf. int.	1.37	1.68	1.61	2.67

The speedup results are shown in Figure 9, which shows branching factor b plotted against average speedup of Bi-directional MCTS vs MCTS for the *Pancake Problem*. The error bars represent the 95% confidence interval. Also plotted is the curve:

$$y = 2.20277846 * exp(0.44500042 * x) - 9.19309151 \quad (4)$$

which was fitted to the experimental data using scipy curve_fit method, which performs a non-linear least squares fit to the data.

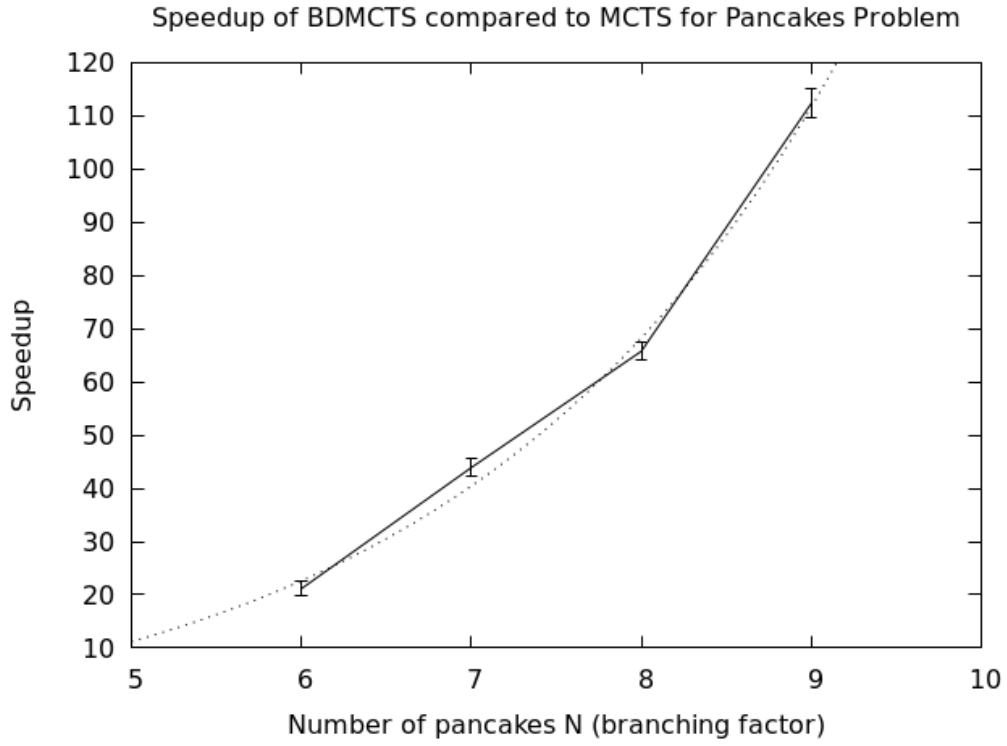

FIGURE 6. The speedup of Bi-directional MCTS compared to MCTS for *Pancake Problem* of increasing branching factor *b*.

DISCUSSIONS

Figure 5 shows that increasing the length of the optimal solution for *8-Puzzle* is marked by an increase in the speedup of Bi-directional MCTS compared to MCTS. The speedup of Bi-directional MCTS grows exponentially proportional to the optimal solution length. Figure 6 shows that increasing the number of pancakes (the branching factor *b*) for *Pancake Problem* is marked by an increase in the speedup of the Bi-Directional enhancement of MCTS.

The present work is a comparison between Monte Carlo Tree Search and a Bi-Directional enhanced Monte Carlo Tree Search. A* was used as a tool to construct optimal solutions to the *8-Puzzle* and *Pancakes Problem*, and has not been compared with either of the MCTS algorithms in terms of time complexity. This is because A* is guaranteed to find an optimal solution to the puzzles when there is an admissible heuristic, whereas MCTS and Bi-directional MCTS do not have the same guarantee. Future research will analyse optimality guarantees of Bi-directional MCTS.

Sturtevant and Felner (Sturtevant, 2018) compared four algorithms (A*, backwards A*, Bi-directional Brute Force Search, and Near-optimal Bi-directional Search) for solving four problems (*Pancakes Problem*, *4-peg Towers of Hanoi*, *Roads of Colorado*, and *Grid Mazes*). They showed that A* expands fewer nodes than Bi-directional Search, if A* has a "strong" heuristic. Otherwise Bi-directional search expands fewer nodes. This mirrors the results that are shown in Figures 5 and 6 in this present work, which suggests that the version of uni-directional MCTS in the present work is not a "strong" heuristic for *8-Puzzle* and *Pancakes Problem*. This would be because the majority of the search effort in uni-directional MCTS (the majority of playouts) would done in the second half the search, and therefore

running a Bi-directional MCTS forwards and backwards will probably remove the majority of the search effort by running the second half of a search in neither forward nor backward direction. The version of uni-directional MCTS in the present work not being a "strong" heuristic for *8-Puzzle* and *Pancakes Problem* is one explanation for the good results of Bi-directional MCTS.

Since the new proposed algorithm Bi-directional MCTS is a form of Bi-directional search, it can only be applied to problems where the goal state is known, e.g. roads or computer networks. The results described in the present article suggest that the Bi-directional MCTS scales well as problems grow larger, since the speed-up of Bi-directional MCTS compared to MCTS increases with problem size. This is encouraging for the use of Bi-directional MCTS for larger problems because it will likely be faster than MCTS.

This paper proposes a new algorithm called Bi-Directional Monte Carlo Tree Search, and presents preliminary results indicating that enhancing Monte Carlo Tree Search by making it Bi-Directional speeds up the search. Bi-directional MCTS speeds up MCTS exponentially with solution length *d* as shown in the *8-Puzzle* results, and also with average branching factor *b* as shown in the *Pancakes Problem* results. This makes Bi-Directional Monte Carlo Tree Search potentially an effective search enhancement that can be applied to other planning problems where there is no known heuristic. Additionally, Bi-Directional Search has been applied to a Reinforcement Learning algorithm, which has previously been reported in (Baldassarre, 2003).

REFERENCES

Baldassarre, G. 2003. Forward and Bidirectional Planning Based on Reinforcement Learning and Neural Networks in a Simulated Robot. *In Anticipatory Behavior in Adaptive Learning Systems*, 179-200.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. *In International Conference on Computers and Games (CG 2006)*, 72–83.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *In Numerische mathematik*, 1(1):269–271.

Gelly, S. and Silver, D. 2008. Achieving master level play in 9 x 9 computer go. *In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 1537–1540.

Goh, S. L., Kendall, G., Sabar, N. R. 2019. Monte Carlo Tree Search in Finding Feasible Solutions for Course Timetabling Problem. *In International Journal on Advanced Science Engineering Information Technology*, 9(6):1936-1943.

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

Holte, R. C., Felner, A., Sharon, G., Sturtevant, N. R., Chen, J. 2017. MM: A bidirectional search algorithm that is guaranteed to meet in the middle. *In Artificial Intelligence*,252: 232-266.

Kocsis, L. and Szepesvàri, C. 2006. Bandit based monte-carlo planning. *In European Conference on Machine Learning*, 282–293.

Matsumoto, S., Hirosue, N., Itonaga, K., Ueno, N., and Ishii, H. 2010. Monte-Carlo Tree Search for a reentrant scheduling problem. *In International Conference on Computers and Industrial Engineering*, 1–6.

Nicholson, T. A. J. 1966. Finding the shortest route between two points in a network. *In The Computer Journal*, 9(3):275–280.

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

Schadd, M. P. D., Winands, M. H. M., Jaap van den Herik, H., Chaslot, G. M. J. B., Uiterwijk, Jos W. H. M. 2008. Single-Player Monte-Carlo Tree Search. *In International Conference on Computers and Games*, 1-12.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L. van den Driessche, G., Schrittwieser, J., Antonoglou, I., vam Veda Lanctot, M. P., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

Sturtevant, N. R. and Felner, A. 2018. A Brief History and Recent Achievements in Bidirectional Search. *AAAI*.

Yajima, T., Hashimoto, T., Matsui, T., Hashimoto, J., and Spoerer, K. 2010. Node-expansion operators for the uct algorithm. *In International Conference on Computers and Games (CG 2010)*, 116–123.

Kristian Spoerer
School of Computer Science,
University of Nottingham, Nottingham, United Kingdom
kristian.spoerer@nottingham.ac.uk