Gregory Walsh – 29785685 – gw2g17@soton.ac.uk

Part 1 – Basic Search Methods

Blocksworld Tile Puzzle

Abstract

In this exercise, the relationship between problem difficulty in the Blocksworld universe and running times of various search algorithms was investigated. A* search, breadth-first search (BFS), depth-first search (DFS) and iterative deepening search (IDS) were the algorithms analysed. Further study was carried out on the heuristics of A* search, the number of optimal solutions, and the effects of including impassable blocks in the world.

Approach

The algorithms to be studied were coded using the Python programming language, in an object-oriented paradigm. In all cases, the algorithms were coded as tree search variants. Tree search algorithms work by applying all possible actions to the state of the root node to generate a set of child nodes. Each child node is then added to the queue in a particular order, depending on the algorithm. The highest priority node is then popped off the queue, and the process of expansion and queueing is repeated, until a goal state is found.

DFS

Python's overhead for function calls is quite high, so DFS was implemented as an iterative approach so that it was possible to explore the search tree as quickly as possible. A first-in first-out (FIFO) queue was used to store the nodes awaiting expansion.

The action sequence was stored as a deque for fast popping. Since the function was coded in the iterative paradigm, some careful management of the action sequence was required. All child nodes were generated in the same iteration, in random order to prevent getting stuck in repetitive loops, and added sequentially to the queue before any other nodes were expanded.

Because the search tree has infinite depth for this problem, the worst-case scenario space complexity is infinite. However, since the solutions occur fairly frequently in the tree, the algorithm is likely to find one within a reasonable amount of time, so this is not of much concern.

IDS was built as a wrapper around the DFS function. The DFS function was designed with an optional variable, with a default value of infinity, that controlled the maximum depth to which the DFS algorithm would search. If a depth argument was supplied, and no goal node was generated within that depth, the function returned a None object.

To find a result, the DFS function was repeatedly called, with an increasing max depth argument beginning at one ply, until a result was returned.

BFS

BFS was implemented using the standard iterative approach. A last-in first-out (LIFO) queue was used to store the nodes awaiting expansion.

To facilitate creating action sequences which lead to goal states, each time a child node was generated, an edge was added to a Python dictionary object, with the child node set as the key and the parent node as the value. When a goal state was found, the action sequence was generated by navigating through the edges backwards from the goal node, until the initial node was reached. Without this object, although the search could determine if the goal state was reachable, it would be unable to reconstruct the path. Since this object contains all nodes generated, it is the reason why BFS has exponential space complexity $O(b^d)$ where b is the branching factor and d is the ply depth of the solution.

A* heuristic Search

Like the other algorithms, A* search works by iteratively generating a queue of nodes, and systematically exploring them until a goal state is found. However, unlike the other algorithms, the order in which the nodes are explored is dependent on the expected total cost of reaching the goal state. This is calculated in two parts. The first part is the cost so far, which for this experiment was the number of actions taken to reach the current state. The second component is the heuristic function which estimates the remaining distance to the goal. For the scalability study, the sum of Manhattan distances for each block to its target was used. This heuristic was selected because it is admissible, since it never over estimates the length of the path.

A* search also varies from the other algorithms since when a child node is created, it is not immediately checked against the goal state. Instead, all child nodes for the current state are generated and then added to the heap. Only when a node is popped is the goal state checked and this way we guarantee that the solution is optimal.

A heap was used to store the queued nodes according to their total expected cost, since heaps are fast to store and retrieve objects that need to be arranged by a priority metric.

Evidence

BFS

Appendix 1 shows the output for the BFS algorithm. It is categorised at two levels, and organised by oldest output first. At the highest level, information about the current node to be expanded is shown. At the second level, information about any child nodes which were created is shown, along with the current contents of the queue. Note - the positions of the blocks and agents are 0 indexed from the bottom left hand corner of the Blocksworld environment for all outputs.

We see the first node to be expanded is node 0, the initial node. Two children are generated, corresponding to a "Left" and an "Up" action respectively. Each time a child node is generated, it is added to the back of the queue. We can see this is the case because when the second node is expanded, it is node 1, which was the second node ("Left") to be added to the queue. Correspondingly, we see that the position of Block C has changed to the bottom right corner (x=3, y=0).

DFS

Appendix 2 shows the output for the DFS algorithm. The structure is the same as described earlier in the BFS section. DFS differs from BFS in that the last child node generated, when a node is expanded, is the next node to be popped from the queue. For example, when node 0 is expanded, the last child node to be generated is node 2. The second node expanded is this same node, node 2. This process repeats until a goal state is found (or the computer runs out of memory).

IDS

Appendix 3 shows the output for IDS. Like the output for DFS, the nodes are expanded in a FIFO order. However, if the wrapped DFS function reaches the maximum depth for that iteration of IDS, it terminates. For example, for a maximum depth of one, only a single node (the initial node) is expanded.

A* Search

Appendix 4 shows the output for A* search using the Manhattan Heuristic. The structure is very similar to that of DFS and BFS, with two important differences. There is no queue output, instead an unordered heap of nodes is displayed with the estimated cost to the goal state shown for each. The total cost shown here is the sum of the cost so far (number of actions taken to get to a state) plus the expected remaining cost to get to the goal function. This can be confirmed by noticing when the first child is generated for node 0, (agent moves left), the estimated cost is 7. This is two more than the combined Manhattan distances of the blocks when in their initial state since Block C has moved to the right, one space further away from its goal, and because a single action has been performed.

When a new node is explored, we see it is always the node with the lowest expected total cost from the last version of the heap. When two or more nodes on the heap have the same lowest expected cost, the oldest node is selected for expansion.

Solutions

The BFS, IDS and A* (with admissible heuristic) search algorithms all returned the same path to the goal state, with a total of 14 actions: *Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left.*

Although it was not guaranteed that these paths would all be the same, we can guarantee that they would all-have equal lengths, since these algorithms are all optimal. This is discussed further in the "Extras" section.

The paths returned by the DFS algorithm varied widely in length. For the Blocksworld problem, they often comprised hundreds of steps. The following is the solution for the simpler problem of getting Block A into the correct position: *Left, Up, Left, Left, Down, Right, Left, Right, Up, Left, Up, Right, Down.*

Scalability Study

To control the complexity of the problem, for the purposes of investigating the time complexity of the algorithms, the number of blocks to be positioned correctly was varied between one and six. When the tiles are few in number, the difficulty of the problem (the depth of the solution) is reduced.

The one block problem comprised getting only Block A to its target position. Additional blocks in alphabetical order were sequentially added to increase the difficulty of the problem. For the problems with four, five, and six blocks, more blocks were added in the row above blocks A, B and C. The targets for these extra blocks were the equivalent positions as for the original blocks, but shifted one column to the left.

To measure the time complexity of each algorithm, the number of nodes expanded to find a solution was recorded for each problem. For DFS, BFS and A* search, this was equal to the number of iterations of the while loops. For the IDS wrapper, this was the sum of nodes expanded for all calls to the DFS function. Because of the stochastic method in which child nodes were generated in the DFS algorithm, the median value for 25 runs, at each problem difficulty, was taken for DFS and IDS. A* search was run with two different heuristics, the sum of total Manhattan distance for misplaced blocks, and a boosted version of said heuristic (which is discussed in the Extras section).

Figure 1 shows how the time complexity varies with the number of blocks (note the logarithmic scale for time complexity). Inspecting it we see that the BFS, IDS and DFS are missing some data points for the more difficult problems. These series are incomplete for these difficulties either because no solution was found within a reasonable time (IDS, DFS), or because the computer ran out of memory (BFS). BFS and IDS were only able to

return solutions to problems with three or fewer blocks. DFS found solutions for five blocks, but only the A* search methods were able to find solutions for problems with more than five blocks. A* search is able to find solutions to more difficult problems within practical time and memory constraints because it searches far fewer nodes. This is because the order in which the nodes are explored is guided by the heuristic function in comparison to the other "blind search" methods.

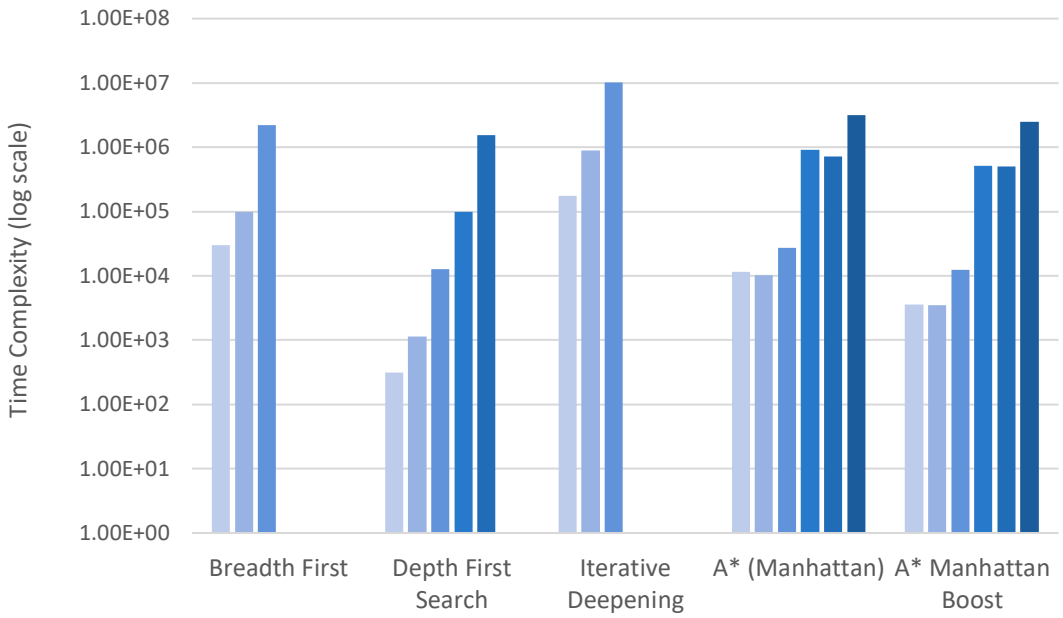

Figure 1 - Time Complexity for several search algorithms, shown with logarithmic scale. Darker coloured bars represent more difficult problems.

Comparing the methods, for all levels of difficulty, we see that DFS tends to find solutions using less computation than the other methods. However, the solutions it finds are almost never optimal, and so the comparison is not fair. Excluding DFS, we see A* search performs better (fewest nodes expanded) than both BFS and IDS. Between IDS and BFS, we see BFS expands fewer nodes. This is to be expected, since IDS must re-explore the tree each time the maximum depth for an iteration is reached.

Finally, we see that the results for A* search do not increase in a smooth exponential fashion (as they do with the other methods). Indeed, the number of iterations taken to solve the two misplaced blocks problem was lower than that to solve the single misplaced block problem. This seems to be a peculiarity of the particular design of Blocksworld, since when different arrangements of blocks were used, this behaviour disappeared.

Extras & Limitations

Other Heuristics

A heuristic function is an estimate of the cost to goal from the current state. If we could calculate this exactly, we would have what is known as a "perfect heuristic", but most of the time a perfect heuristic means we already know the path to the goal, so we have no need for the heuristic. In a more typical case, we have a heuristic which underestimates the remaining cost to the goal. To generate a heuristic, we can take the problem in hand, and relax some of the rules until the costs to the goal state is straightforward to calculate directly. In order to find a good heuristic for Blocksworld, some deeper analysis of the rules of the problem was performed. These rules are: blocks can only interact with the space, blocks switch places with the space when they interact, and blocks and spaces can only interact with objects in directly adjacent locations.

Describing these rules in general terms, and considering alternatives, we get the following:

1. Target
    a. Opposite - Blocks can only interact with spaces and vice versa
    b. Any - Blocks can interact with other blocks
2. Interaction
    a. Switch - A moved block or space switches with whatever is at the destination
    b. Stack - Blocks and the space stack up
3. Movement
    a. Adjacent - Blocks and spaces can only move to adjacent locations
    b. Column/Row - Blocks and spaces can only move to locations on the same column, or row
    c. Teleport - Blocks and spaces can teleport to any location

It is possible to create a game with any combination of these rules. To give a simple example, take rules 1b, 2b & 3c. In this game, we can win by simply teleporting each block to its goal location. This will take three actions, and is equivalent to the misplaced tiles heuristic. Table 1 shows the 12 possible combinations of these rules, and the heuristics found.

Target	Interaction	Movement	Heuristic
Opposite	Switch	Adjacent	A perfect heuristic for Blocksworld
Opposite	Switch	Column/Row	*None found*
Opposite	Switch	Teleport	Count of misplaced tiles
Opposite	Stack	Adjacent	Manhattan distance + boost*
Opposite	Stack	Column/Row	2 x misaligned axes (assuming have to move the space first)
Opposite	Stack	Teleport	2 x misplaced tiles (assuming have to move the space first)
Any	Switch	Adjacent	*None found*
Any	Switch	Column/Row	*None found*
Any	Switch	Teleport	Count of misplaced tiles
Any	Stack	Adjacent	Manhattan Distance
Any	Stack	Column/Row	Count of misaligned axes
Any	Stack	Teleport	Count of misplaced tiles

*boost – description given below

Table 1 – Analysis of possible heuristics for different games based on Blocksworld

The games in which switching is required (including Blocksworld) are more difficult to generate heuristics for than those in which stacking is allowed. Inspecting the rows in which stacking is allowed, we see the familiar count of misplaced tiles heuristic and the Manhattan distance metric. Upon further inspection, we see there is an interesting combination: Target = Opposite, Interaction = Stack, Movement = Adjacent. In this world, we need to navigate the space to a misplaced block (it would not be able to move otherwise) and then drag it to its goal position. The distance the blocks will need to move will be the Manhattan distance, but the space still needs to find the shortest route to collect all the blocks. Instead of trying to find this shortest path, we can make another simplification and say that this shortest path will be at least as long as the distance to the square next to the closest block which is nearest to the space. Adding in this boosting* factor helps keep the near the blocks when even when an action does not decrease the Manhattan distance. The improvement in performance can be seen in Figure 1, and varied between 50% and 5% depending on the difficulty of the puzzle.

Number of optimal solutions to Blocksworld

To determine how many optimal solutions Blocksworld has, the A* algorithm was modified to continue searching for solutions after the first solution was found. It was discovered that indeed there is a single solution, and explains why all three optimal methods returned the same result.

Impassable Squares

In a final extension, impassable squares were added to the environment in various positions, and searches using the A* search with the boosted Manhattan distance were performed. It was found that depending on the position of the impassable squares, that the time to find a solution might either increase or decrease. If the impassable squares were placed in positions adjacent to the blocks, the solution depth would always increase, since the unique optimal solution was no longer feasible. If the blocks were placed in the top row of the environment, either the time to solution decreased, or was unaffected. This is likely due to the boosted heuristic's preference to generate states in which the agent remains close to the blocks.

Part 2 – Performance Analysis of Negamax with Enhancements

Gregory Walsh
29785685
gw2g17@soton.ac.uk

ABSTRACT

In this report, a discussion is given of the properties of the negamax algorithm, a minimax algorithm variant, with and without alpha-beta pruning in the context of the children's game Connect 4. Furthermore, the performance benefits of alpha-beta pruning with and without node expansion ordering are analysed. In both cases, the enhancements were found to significantly reduce the number of nodes explored by the program, with benefits becoming more pronounced for greater maximum search depths.

1 INTRODUCTION

When dealing with two player adversarial turn-based games, which are zero-sum (meaning the gain of a player is exactly equal to their opponent's losses), such as chess, Connect 4 or backgammon, the simplest version of the minimax algorithm provides a method to find the optimal move for a particular player by exploring the entire game tree. However, since game trees for most games are large enough to make the calculation intractable, instead of attempting to evaluate the whole tree, one can instead evaluate an incomplete game tree with restricted depth, and a heuristic function can be used to estimate which player has the advantage for each leaf node and by how much at that depth [1].

Negamax is a variant on the minimax algorithm. Rather than using two functions which return the minimum and maximum choices at a particular level, negamax uses a single function, which selects moves which maximise the score at all levels from the perspective of the player at that level.

Alpha-beta pruning works by eliminating branches on the tree which are guaranteed to return values which will have no influence on the final result. Pruning branches can significantly reduce the search space, thereby enabling searches to deeper depths for the same number of nodes evaluated. It also possible to improve alpha-beta pruning by expanding nodes which are expected to result in the highest heuristic evaluations first, since more branches can be pruned.

In this report an analysis of the relative performance of the negamax function with and without alpha-beta pruning, and with and without node ordering is given.

2 METHODS

2.1 Choice of Game

Connect 4 was selected for the experiment since the game tree is relatively small in comparison to other games like chess, and therefore many games could be played and aggregated to get more reliable data for analysis. Furthermore, the algorithm could be run to deeper depths.

2.2 Negamax

Since negamax can be coded as a single function, the author expected that debugging a program implementing this algorithm would be more efficient than for the minimax function. Furthermore, the author felt that the negamax algorithm expresses more naturally the way in which competitors attempt to maximise their own score (after all, what chess player with black pieces looks at the board, sees that he is a pawn and a knight up and concludes his material score is minus four?). For these reasons, negamax was selected for evaluating the game space.

2.3 Base Case Checks

Base case checks are tests performed by a recursive function at the beginning of the execution which, if fulfilled, cause the immediate return of a value, rather than further recursion, and are required to avoid useless non-terminating behaviour. In Connect 4, there are two base cases to check.

First, we check if the maximum depth has been reached, in which case we estimate the value of a node using the heuristic function. Second, since Connect 4 has an equivalent goal state for both players (get four of your chips in a row), and a draw state (board is full and no winner), the program must check for these terminal states each time a new node is generated.

2.4 Heuristic Function

To evaluate the goodness of a particular move, for each chip on the board, the number of potential winning combinations to which that chip could contribute were counted and summed by player. The difference of these sums was then returned. This encouraged play towards the centre of the board. Blocked four-in-a-rows were also included in the counts as a simplification.

2.5 Alpha-Beta Pruning

To implement alpha-beta pruning in the negamax paradigm, the minimax version of alpha-beta pruning algorithm was analysed with the always-maximising approach of the negamax algorithm in mind to determine the necessary modifications, of which there are two.

Firstly, since the perspective of the values of scores switches each time the function is called from one layer to the next, necessarily the perspectives of the alpha and beta values must also

switch, and therefore we multiply the values by a factor of minus one.

Secondly, and more subtly, because the comparison between alpha and beta is always of the same form (that alpha < beta) and because we switch the signs of these values each time they are passed to a child node, the values of alpha and beta must be transposed. In other words, the lowest score a player at level 'n' can expect to get will become the highest score a player level n+1 can expect to get (alpha -> beta), and vice versa.

2.6 Node Ordering

Node ordering was accomplished by creating a simple heuristic which worked by looking up the number of four-in-a-rows to which each possible action could contribute, and then expanding the nodes with the greatest number of four-in-a-row contributions first. To optimise the process, the four-in-a-row counts were pre-calculated for all possible positions.

2.7 Debugging and Proof of Correct Operation

For the purposes of debugging the program and ensuring the results were consistent with the algorithm, functionality to capture and print the game tree was added to the program.

2.8 Design of the Performance Experiment

To investigate the difference in time complexity of the three variants, each version was run with maximum depths varying from one to seven plies. At each search depth, fifty games, initialised from the empty board state, were played between two computer controlled opponents. Within the negamax function, a counter was added to keep track of the number of nodes evaluated as a measure of time complexity.

To get an unbiased estimate of the time complexity which would better reflect the relative performance of the variants in general, moves were randomly selected from a set of candidates with maximum scores (often this would be a single move). If move selection were not randomised, the analysis would be biased to a left to right column ordering, i.e. the order in which possible nodes were generated and expanded by the first two variants.

3 RESULTS AND DISCUSSION

3.1 Proof of Correctness

Appendix 6 shows the output for a small search tree explored by the negamax function. The tree was generated with a search depth of three from a mid-game position (as illustrated in the output). In this particular game position, the current player (player two) can either win or lose within a couple of moves. The order in which the tree is printed, top to bottom, is the order in which the nodes of the tree were expanded.

Maximum Depth Reached Termination

To prove that the program halts at the correct depth, we must show that no more than three plies of actions are ever evaluated for this example. Inspecting the tree, we see this is the case.

Win Condition Termination

From the tree we see that should player two chose columns one or three as their first move, player one could win by subsequently choosing column five. As such the scores corresponding to either of these sequences of moves (nodes 10 and 17) are, at the player one level, given infinite value, and the tree terminates early, with no subsequent player two actions.

Negative Maximisation

Evidence of negative maximisation, equivalent to the cyclical minimisation and maximisation of the minimax algorithm is the final requirement for proof of correctness for the negamax algorithm. We see for each level that the negation of the maximum value is calculated and passed back to the calling function in the level before.

Alpha-Beta Pruning

To be sure alpha-beta pruning is working as expected, we must demonstrate that useless branches, which provably cannot generate better moves that have already been generated, are not expanded. Appendix 7 shows the game tree for the same initial state just described, but this time generated using alpha-beta negamax. Note – the values shown for alpha are the values calculated after the node has been evaluated.

Looking at node 10, we see player two has played column three which then makes a win in column three possible for player one. The value of alpha therefore becomes +infinity at the node 15 search level, and equal to beta. As a result, the final child of node 10 (which would correspond to player one selecting column 5) is not explored. A more involved example of pruning occurs immediately following node 8, in circumstances where beta is finite (due to the results of node 5) rather than infinite.

3.2 Performance Comparison - Negamax versus Negamax with Alpha-Beta Pruning

Figure 1 shows, for various maximum search depths, the mean number of iterations executed by the program for a single game, when running the negamax algorithm with and without the alpha-beta pruning. For all search depths greater than one, we see that the number of nodes evaluated by alpha-beta negamax is smaller than that of the standard negamax algorithm. For all three variants, we observe exponential growth (straight line on a logarithmic axis) as the maximum search depth increases.

In the case of a maximum search depth of one, no branches can be pruned from the graph when operating with fail-soft comparison, since beta takes the value of positive infinity for all nodes in the first layer, and so no value of alpha will ever be greater. If instead the fail-hard test is used, it is possible that nodes will be pruned even at the first level if an end-state is found, although any game having an end-state within a single move would be exceedingly dull to play.

Figure 2 shows the ratio of the average number of iterations in a single game for standard negamax and alpha-beta negamax. As the search depth increases we see the ratio between the two variants steadily falls. Because the number of iterations required at greater depths is very large (particularly for the plain negamax

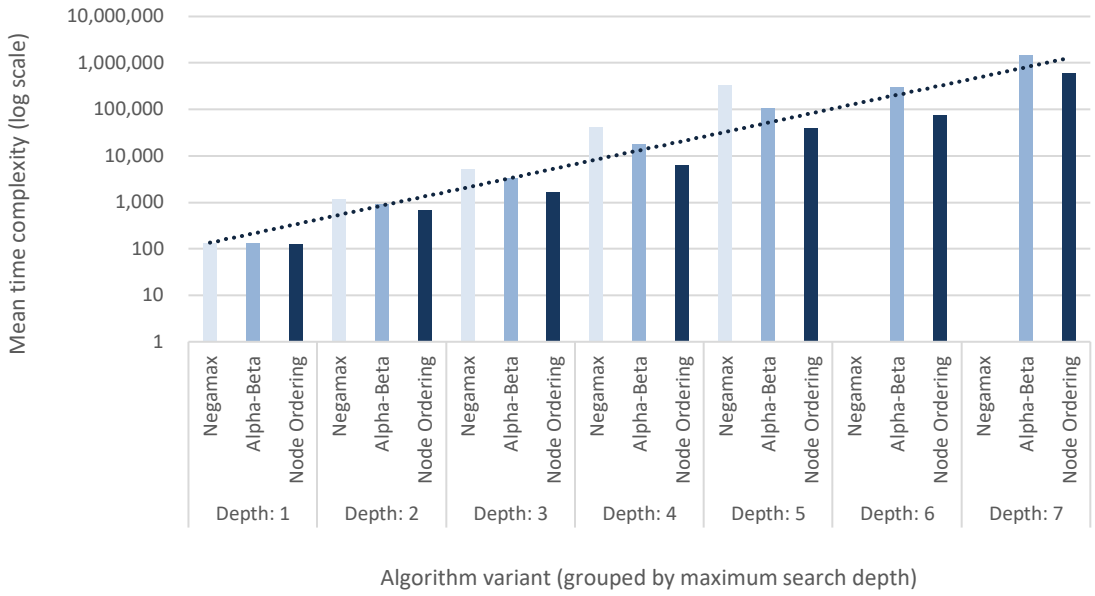

Figure 1. Time complexity for three variants of the negamax algorithm shown with logarithmic scale.

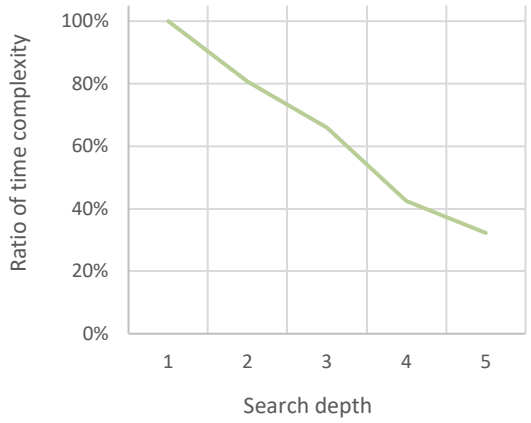

Figure 2. Ratio of time complexity Alpha-Beta / Negamax

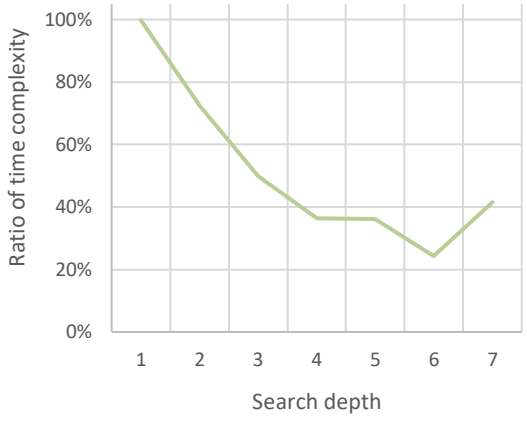

Figure 3. Ratio of time complexity Alpha-Beta with ordering / Alpha-Beta

variant), the analysis was limited to a search depth of six plies for negamax and seven plies for the alpha-beta pruning variants.

If it were practical to analyse the methods to deeper depths, we would expect the ratio to eventually stabilise as the as the measured time complexities approach the predicted time complexity in the limit that the depth d approaches infinity. However, as the search depth approaches the length of the game (at most 42 moves, frequently less) we would expect the trend to reverse, and the ratio begin to increase. Towards the end of the game, fewer moves become available as columns fill up, so the branching factor decreases and alpha-beta pruning becomes less effective. Consider for example the extreme case in which the branching factor b approaches 1. Under these circumstances the ratio of the time complexity for negamax $O(b^d)$ and alpha-beta pruning $O\left(b^{\frac{d}{2}}\right)$, we have:

$$\forall d > 0: \lim_{b \to 1} b^{\left(\frac{d}{2} - d\right)} = 1$$

In other words, the expected time complexities would converge. [2]

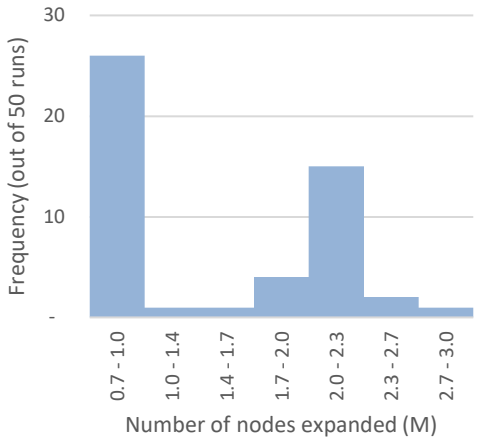

Figure 4. Histogram showing frequency of number of nodes expanded for alpha-beta negamax with a maximum search depth of seven

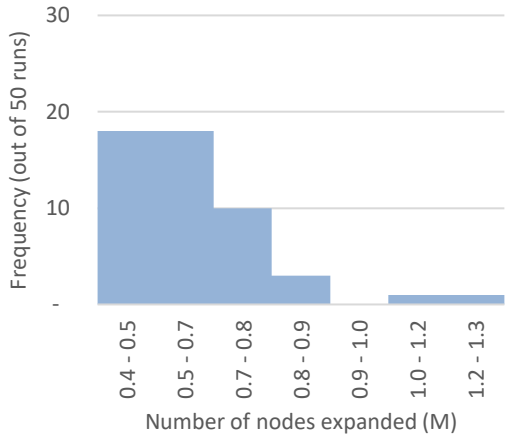

Figure 5. Histogram showing frequency of number of nodes expanded for alpha-beta negamax with node ordering and maximum search depth of seven

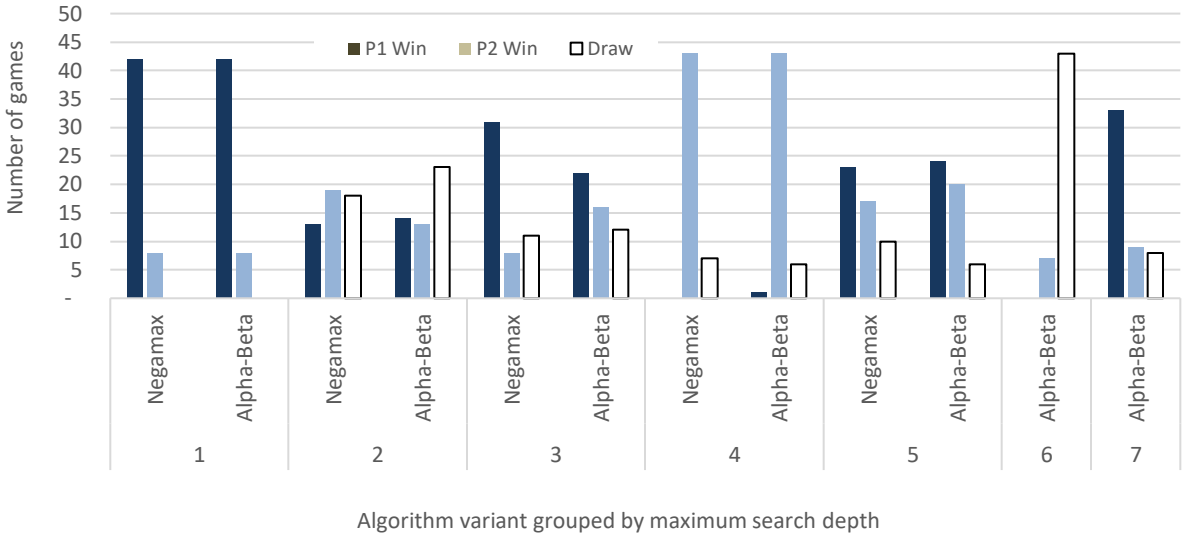

Figure 6. Win, loss and draw counts by player. All games are computer versus computer with equal max search depth.

3.2 Performance Comparison - Alpha-Beta versus Alpha-Beta with Node Ordering

Figure 1 also shows the time complexity of alpha-beta pruning with node ordering. For all search depths greater than one, we see that the number of nodes evaluated by alpha-beta pruning with node ordering is smaller than those of the other two variants.

Figure 3 shows the ratio of the average number of iterations in a single game for negamax with alpha beta pruning with and without node ordering. From depths one to six, we observe the ratio between the two variants steadily falls. However, something unexpected happens at depth seven, and the ratio actually increases.

Inspecting the distributions of the number of iterations per run (Figure 4 and Figure 5) for the games played with a maximum search depth of seven for the two alpha-beta pruning variants, we see they are quite different. The distribution for variant without node ordering appears to be bi-modal, with a large proportion of runs having iteration counts between 0.7 and 1M. In contrast, the variant with node-ordering does not show a bi-modal distribution.

It is possible therefore, that for some trees built without node ordering, a particular kind of critical situation may arise when evaluating early levels which has significant knock-on effects on the total number of nodes evaluated. Some supporting evidence for this theory can be found by inspecting Figure 6 (which shows win/loss ratios for player one and two at different depths over fifty

games), and noting that at a depth of seven, player one suddenly dominates, suggesting some strategy for an easy win emerges at this depth, perhaps in the early game.

The consequence of the differences between these two distributions means not only can we expect shorter running times for the node ordering variant in comparison to basic alpha-beta pruning, but also that the running times will be more consistent. This consistency property would be attractive when building applications which have strictly limited execution times, especially one using iterative deepening, since one would have more confidence in the depth to which the program could search within a particular time frame

4 CONCLUSIONS

It has been shown in this report that alpha-beta pruning combined with node ordering outperforms the other negamax variants in terms of time complexity. However, the theorisations provided on the possible reasons for the unusual relationship between maximum search depth and the ratio of nodes explored between alpha-beta negamax with and without node ordering are very tentative.

In an attempt to better understand this phenomenon, one could quantitatively analyse the relationships amongst the maximum search depth, the turn number of winning moves, and the total number of nodes evaluated. In doing so, one could get a better picture of how the iteration distributions are composed. Furthermore, since the results of games between two computer opponents appear to be highly dependent on the maximum search depth (as illustrated by the volatility in the counts of wins, draws and losses at different depths - see Figure 6), developing a qualitative understanding of the specific game play patterns which emerge at different depths, by studying individual games, in combination with the quantitative analysis would likely help to resolve a more complete picture of the system's dynamic behaviour.

REFERENCES

[1] Shannon, C. E. (1988). Programming a computer for playing chess. In Computer chess compendium (pp. 2-13). Springer New York.

[2] Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd Edition).

Appendix 1 – BFS Proof

```
BREADTH FIRST SEARCH

============================================== First node expanded
==============================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)


============================================== Second node expanded
==============================================

Expanded node's state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)

==================== First child generated ====================
Agent move to: Position(x=1, y=0)
New state: ID 3   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 3   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=0)
New state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2,
y=0)  ||  C at Position(x=3, y=0)
Priority 3 state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Third child generated ====================
Agent move to: Position(x=2, y=1)
New state: ID 5   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)
OUTPUT TRUNCATED
```

Appendix 2 – DFS Proof

```
DEPTH FIRST SEARCH
========================================== First node expanded ==========================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)


========================================== Second node expanded ==========================================

Expanded node's state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=3, y=2)
New state: ID 3   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=0)
New state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 3 state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Third child generated ====================
Agent move to: Position(x=2, y=1)
New state: ID 5   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 3 state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 4 state: ID 5   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
OUTPUT TRUNCATED
```

Appendix 3 – IDS Proof

```
ITERATIVE DEEPENING SEARCH

#############################################################################################################
Searching to a depth of 1
#############################################################################################################

========================================== First node expanded ==========================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 2 state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

#############################################################################################################
Searching to a depth of 2
#############################################################################################################

========================================== First node expanded ==========================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Second child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 2   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 2   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)


========================================== Second node expanded ==========================================

Expanded node's state: ID 2   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=1)
New state: ID 3   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
```

```
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)

==================== Second child generated ====================
Agent move to: Position(x=1, y=0)
New state: ID 4   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 3 state: ID 4   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2,
y=0)  ||  C at Position(x=3, y=0)

==================== Third child generated ====================
Agent move to: Position(x=3, y=0)
New state: ID 5   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 3   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)
Priority 3 state: ID 4   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2,
y=0)  ||  C at Position(x=3, y=0)
Priority 4 state: ID 5   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)


##################################################################################################################
Searching to a depth of 3
##################################################################################################################


========================================== First node expanded ==========================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)

==================== Second child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 2   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Queue
Priority 1 state: ID 1   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=2, y=0)
Priority 2 state: ID 2   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1,
y=0)  ||  C at Position(x=3, y=0)


OUTPUT TRUNCATED
```

Appendix 4 – A* Search Proof

```
A* SEARCH

========================================== First node expanded ==========================================

Expanded node's state: ID 0   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=0)
New state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Heap
State: ID 1   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=1)
New state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Heap
State: ID 1   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)
State: ID 2   ||  Estimated Cost 6   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)


========================================== Second node expanded ==========================================

Expanded node's state: ID 2   ||  Agent at Position(x=3, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== First child generated ====================
Agent move to: Position(x=2, y=1)
New state: ID 3   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Heap
State: ID 1   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== Second child generated ====================
Agent move to: Position(x=3, y=0)
New state: ID 4   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Heap
State: ID 1   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 4   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

==================== Third child generated ====================
Agent move to: Position(x=3, y=2)
New state: ID 5   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Heap
State: ID 1   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 4   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 5   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
```

```
========================================== Third node expanded ==========================================

Expanded node's state: ID 1   ||  Agent at Position(x=2, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)

=================== First child generated ====================
Agent move to: Position(x=1, y=0)
New state: ID 6   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=2, y=0)
||  C at Position(x=3, y=0)

Updated Heap
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 4   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 5   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 6   ||  Estimated Cost 9   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=2, y=0)  ||  C at Position(x=3, y=0)

=================== Second child generated ====================
Agent move to: Position(x=3, y=0)
New state: ID 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=2, y=0)

Updated Heap
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 4   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 5   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 6   ||  Estimated Cost 9   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=2, y=0)  ||  C at Position(x=3, y=0)
State: ID 7   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)

=================== Third child generated ====================
Agent move to: Position(x=2, y=1)
New state: ID 8   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at Position(x=1, y=0)
||  C at Position(x=3, y=0)

Updated Heap
State: ID 3   ||  Estimated Cost 7   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 4   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 5   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=2)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 6   ||  Estimated Cost 9   ||  Agent at Position(x=1, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=2, y=0)  ||  C at Position(x=3, y=0)
State: ID 7   ||  Estimated Cost 7   ||  Agent at Position(x=3, y=0)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=2, y=0)
State: ID 8   ||  Estimated Cost 8   ||  Agent at Position(x=2, y=1)  ||  A at Position(x=0, y=0)  ||  B at
Position(x=1, y=0)  ||  C at Position(x=3, y=0)
OUTPUT TRUNCATED
```

Appendix 5.1 – Main Script (Author's Code)

```
from collections import namedtuple
from globalsearch import a_star_tree_search, depth_first_tree_search, breadth_first_tree_search,
iterative_deepening_tree_search
from heuristics import blockworld_misplaced_tile_count, blockworld_manhattan, blockworld_manhattan_boost
from problemconcepts import Problem, State, Agent, SwitchTilesAction, Grid

def line_from_file_generator(file_name):
    with open(file_name) as text_file:
        for line in text_file:
            yield line

def build_grid_from_delimited_strings(delimited_strings, delimiter):
    grid_data = []
    for delimited_string in delimited_strings:
        grid_data.append(map(int, delimited_string.strip().split(delimiter)))
    return Grid(grid_data)

##################################### SET UP THE PROBLEM ######################################

# CREATE AGENT

def search_algorithm_wrapper(problem):
    #return iterative_deepening_tree_search(problem)
    return depth_first_tree_search(problem)
    #return breadth_first_tree_search(problem)
    #return a_star_tree_search(problem, blockworld_manhattan)

agent = Agent(name='Tile Switcher',
              skill=SwitchTilesAction,
              search_algorithm=search_algorithm_wrapper)

# CREATE INITIAL ENVIRONMENT AND STATE
lines_from_file = line_from_file_generator('blocksworld.txt')
environment = build_grid_from_delimited_strings(delimited_strings=lines_from_file, delimiter=' ')

# Define a pseudo-class for position
Position = namedtuple('Position', ['x', 'y'])

# Set agents initial position
agent_initial_position = Position(x=3, y=0)

# Create the tiles and goals
tiles = {'A': Position(0, 0),
         'B': Position(1, 0),
         'C': Position(2, 0),
         'D': Position(0, 1),
         'E': Position(1, 1),
         'F': Position(2, 1)
         }

goals = {'A': Position(1, 2),
         'B': Position(1, 1),
         'C': Position(1, 0),
         'D': Position(0, 2),
         'E': Position(0, 1),
         'F': Position(0, 0)
         }

initial_state = State(environment_data={'agent_location': agent_initial_position, 'tiles': tiles, 'goals':
goals})

# CREATE A PROBLEM OBJECT

# Define the goal test
def goal_test_function(state):
    tiles = state.environment_data['tiles']
    goals = state.environment_data['goals']
    all_goals_met = True
    for tile_key, position in tiles.items():
        if position != goals[tile_key]:
            all_goals_met = False
            break
```

```
    return all_goals_met

problem = Problem(goal_test_function=goal_test_function,
                  initial_state=initial_state,
                  environment=environment,
                  agent=agent,
                  all_skills={SwitchTilesAction})

##################################### EXECUTE THE PROBLEM ########################################

# Execute search
num_iters_sample = []

for i in range(25):
    solution = problem.search()
    if solution[0] is not None:
        num_iters_sample.append(solution[1])
        print(solution[1])

print(num_iters_sample)

solution = problem.search()
# Print any successful action sequence with cost
if solution[0] is not None:
    goal_state, num_iterations, final_ply_depth, action_sequence = solution
    total_cost = 0
    state = initial_state
    moves = []
    i = 0
    for action in action_sequence:
        i += 1
        result = action.execute(state)
        x_change = result.state.environment_data['agent_location'].x -
state.environment_data['agent_location'].x
        y_change = result.state.environment_data['agent_location'].y -
state.environment_data['agent_location'].y
        if x_change == -1:
            moves.append("Left")
        elif x_change == 1:
            moves.append("Right")
        elif y_change == -1:
            moves.append("Down")
        elif y_change == 1:
            moves.append("Up")
        state = result.state
        tiles = state.environment_data['tiles']
        print("Action" , str(i), "\nMove to", action.agent_destination)
        for tile_key, position in tiles.items():
            print("Result:", tile_key, "at", tiles[tile_key])
        total_cost += result.cost
    print('Total cost', total_cost)
    print('Final ply depth: ', final_ply_depth)
    print('Action sequence len: ', len(action_sequence))
    print('Total nodes expanded: ', num_iterations)
    print('Action sequence works:', goal_test_function(state))
    print('Moves:', moves)
else:
    print('No solution found')
```

Appendix 5.2 – Search Algorithms (Author's Code)

Note - Search Algorithms with debugging output can be found in the zip file

```
from collections import deque
from random import shuffle
from priority_queue import PriorityQueue

def extract_action_sequence(action_edges, terminating_action):

    action_sequence = [terminating_action]
    parent = action_edges.get(terminating_action, None)

    while parent:
        action_sequence.append(parent)
        parent = action_edges.get(parent, None)

    action_sequence.reverse()

    return action_sequence


def depth_first_tree_search(problem, max_depth=float('inf')):

    # Initiate return variables
    action_sequence = deque()
    num_nodes_expanded = 0
    current_ply_depth = 0
    state_incrementer = 0

    # First check if start state is the goal state
    if problem.goal_test_function(problem.initial_state):
        return problem.initial_state, num_nodes_expanded, current_ply_depth, action_sequence
    else:

        # Initiate queue
        parent_action = None
        queue = deque([(problem.initial_state, state_incrementer, current_ply_depth, parent_action)])

        while queue:

            # Track the number of nodes expanded
            if num_nodes_expanded % 10000 == 0:
                print('Number of nodes expanded: ', num_nodes_expanded)
            num_nodes_expanded += 1

            # Get the next node to expand
            state, state_id, current_ply_depth, parent_action = queue.pop()
            child_ply_depth = current_ply_depth + 1

            # Backtrack/update the action sequence
            num_prior_actions = current_ply_depth
            for i in range(len(action_sequence) - num_prior_actions):
                action_sequence.pop()
            action_sequence.append(parent_action)

            # Determine if max_depth will be exceeded when running in iterative deepening mode
            if child_ply_depth <= max_depth:

                # Determine applicable actions
                applicable_actions = problem.get_applicable_actions(state)
                shuffle(applicable_actions)

                # Check all child nodes for goal state, and add failures to the queue
                for child_action in applicable_actions:
                    state_incrementer +=1
                    child_state_id = state_incrementer
                    child_result = child_action.execute(state)
                    child_state = child_result.state

                    # Test for the goal state
                    if problem.goal_test_function(child_state):
                        # If goal state then return sequence
                        actions = [action for action in action_sequence] + [child_action]
                        return child_state, num_nodes_expanded, current_ply_depth, actions[1:]

                    else:
```

```
                        queue.append((child_state, child_state_id, child_ply_depth, child_action))

        # If no solution is found return None
        return None, num_nodes_expanded


def iterative_deepening_tree_search(problem, max_depth=float('inf')):

    # Initiate loop variables
    iteration_max_depth = 1
    total_num_nodes_expanded = 0
    nodes_per_level = []
    while iteration_max_depth <= max_depth:

        # Get depth limited result
        result = depth_first_tree_search(problem, max_depth=iteration_max_depth)
        total_num_nodes_expanded += result[1]
        nodes_per_level.append(result[1])

        # Return the result if not None, otherwise continue searching deeper
        if result[0]:
            print(nodes_per_level)
            goal_state, num_nodes_expanded, current_ply_depth, action_sequence = result
            return goal_state, total_num_nodes_expanded, current_ply_depth, action_sequence
        else:
            iteration_max_depth += 1

    # If no solution is found at max_depth return None
    return None, total_num_nodes_expanded


def breadth_first_tree_search(problem):

    # Initiate return variables
    action_sequence = []
    num_nodes_expanded = 0
    current_ply_depth = 0
    parent_action = None
    state_incrementer = 0

    # First check if start state is the goal state
    if problem.goal_test_function(problem.initial_state):
        return problem.initial_state, num_nodes_expanded, current_ply_depth, action_sequence
    else:

        # Initiate queue and tree data structure
        queue = deque([(problem.initial_state, state_incrementer, current_ply_depth, parent_action)])
        action_tree_edges = {}

        while queue:

            # Track the number of nodes expanded
            if num_nodes_expanded % 10000 == 0:
                print('Number of nodes expanded: ', num_nodes_expanded)
            num_nodes_expanded += 1

            # Get the next node to expand
            state, state_id, current_ply_depth, parent_action = queue.pop()
            child_ply_depth = current_ply_depth + 1

            # Expand the current nodes and add children to the queue
            for child_action in problem.get_applicable_actions(state):

                state_incrementer +=1
                child_state_id = state_incrementer

                # Store the action edge
                action_tree_edges[child_action] = parent_action

                # Generate the child state
                child_result = child_action.execute(state)
                child_state = child_result.state

                # Test for the goal state
                if problem.goal_test_function(child_state):
                    # Calculate the action sequence
```

```
                    action_sequence = extract_action_sequence(action_edges=action_tree_edges,
terminating_action=child_action)
                    return child_state, num_nodes_expanded, current_ply_depth, action_sequence
                else:
                    queue.appendleft((child_state, child_state_id, child_ply_depth, child_action))

        # If no solution is found return None
        return None, num_nodes_expanded


def a_star_tree_search(problem, heuristic_function):

    # Initiate return variables
    num_nodes_expanded = 0
    current_ply_depth = 0
    action_tree_edges = {}
    state_incrementer = 0

    # Initiate loop variables
    estimated_total_cost = 0  # Value is irrelevant for root node since always popped first
    cost_to_current_ply = 0
    parent_action = None
    queue = PriorityQueue()
    queue.add_task((problem.initial_state, state_incrementer, current_ply_depth, cost_to_current_ply,
parent_action), estimated_total_cost)

    while not queue.is_empty():

        # Track the number of nodes expanded
        if num_nodes_expanded % 10000 == 0:
            print('Number of nodes expanded: ', num_nodes_expanded)
        num_nodes_expanded += 1

        # Get the next node to explore
        state, state_id, current_ply_depth, cost_to_current_ply, parent_action = queue.pop_task()

        # Test for the goal state, with admissible heuristic this is guaranteed to be the optimal solution
        if problem.goal_test_function(state):
            action_sequence = extract_action_sequence(action_edges=action_tree_edges,
terminating_action=parent_action)

            return state, num_nodes_expanded, current_ply_depth, action_sequence

        else:

            # Expand the current nodes and add children to the queue
            for child_action in problem.get_applicable_actions(state):

                state_incrementer +=1
                child_state_id = state_incrementer

                # Store the action edge
                action_tree_edges[child_action] = parent_action

                # Generate the child state
                result = child_action.execute(state)
                child_state = result.state

                # Calculate the cost to this point
                cost_to_child_ply = cost_to_current_ply + result.cost
                estimated_total_cost = cost_to_child_ply + heuristic_function(child_state)

                # Insert next node into the queue
                child_ply_depth = current_ply_depth + 1
                queue.add_task((child_state, child_state_id, child_ply_depth, cost_to_child_ply,
child_action), estimated_total_cost)

    # If no solution is found return None
    return None, num_nodes_expanded
```

Appendix 5.3 – Heuristics (Author's Code)

```
def blockworld_manhattan_boost(state):

    def manhattan_distance_to_closest_misplaced_tile(agent_location, tiles, goals):
        # check for misplaced tile and find the distance to the closest one
        min_distance_yet = None
        for tile_key, tile_position in tiles.items():
            if tile_position != goals[tile_key]:
                if min_distance_yet is None:
                    min_distance_yet = manhattan_dist(agent_location, tile_position) - 1
                else:
                    min_distance_yet = min(min_distance_yet, manhattan_dist(agent_location, tile_position) -
1)

        return min_distance_yet

    # Manhattan distance for all tiles
    tiles = state.environment_data['tiles']
    goals = state.environment_data['goals']
    agent_location = state.environment_data['agent_location']

    manhattan_cost = blockworld_manhattan(state)

    # Compute agent cost to nearest misplaced tile
    agent_cost = manhattan_distance_to_closest_misplaced_tile(agent_location=agent_location, tiles=tiles,
goals=goals)
    # Check if is None since there may be no misplaced tiles
    if agent_cost:
        return manhattan_cost + agent_cost

    return manhattan_cost


def blockworld_manhattan(state):
    total_distance = 0
    for tile_key, position in state.environment_data['tiles'].items():
        total_distance += manhattan_dist(position, state.environment_data['goals'][tile_key])
    return total_distance


def blockworld_misplaced_tile_count(state):
    count_misplaced = 0
    for tile_key, position in state.environment_data['tiles'].items():
        if position != state.environment_data['goals'][tile_key]:
            count_misplaced += 1
    return count_misplaced


def manhattan_dist(old_position, new_position):
    x_dist = abs(old_position.x - new_position.x)
    y_dist = abs(old_position.y - new_position.y)
    return x_dist + y_dist
```

Appendix 5.4 – Problem, Agent, State & Environment (Author's Code)

```
from collections import namedtuple
from heuristics import manhattan_dist


Position = namedtuple('Position', ['x', 'y'])


class Problem(object):

    def __init__(self, goal_test_function, initial_state, environment, agent, all_skills):
        self.goal_test_function = goal_test_function
        self.initial_state = initial_state
        self.environment = environment
        self.agent = agent
        self.all_skills = all_skills

    def permissible_skills(self, state=None):
        return self.all_skills

    def get_applicable_actions(self, state):
        actions = []

        for applicable_target, cost in self.agent.skill.get_targets(state=state, grid=self.environment):
            actions.append(self.agent.skill(applicable_target, cost))

        return actions

    def search(self):
        return self.agent.search_algorithm(self)


class Result(object):
    def __init__(self, state, cost):
        self.state = state
        self.cost = cost


class SwitchTilesAction(object):

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    @staticmethod
    def get_targets(state, grid):

        current_position = state.environment_data['agent_location']

        for x_move, y_move in SwitchTilesAction.moves:
            new_position = Position(current_position.x + x_move, current_position.y + y_move)
            if SwitchTilesAction.is_acceptable_position(new_position, grid):
                cost = manhattan_dist(current_position, new_position)
                yield new_position, cost


    @staticmethod
    def is_acceptable_position(position, grid):

        x_within_bounds = 0 <= position.x <= grid.width - 1
        y_within_bounds = 0 <= position.y <= grid.height - 1

        if x_within_bounds and y_within_bounds:
            return grid.is_passable(position)
        return False

    def __init__(self, agent_destination, cost=None):
        self.cost = cost
        self.agent_destination = agent_destination


    def execute(self, state):
        new_state = self.generate_new_state(state)
        return Result(state=new_state, cost=self.cost)
```

```
    def generate_new_state(self, state):


        # Record the location of the agent before the action
        new_state = State(environment_data={'agent_location': self.agent_destination, 'tiles': {}, 'goals':
state.environment_data['goals']})

        # Update the positions of the tiles if they moved
        agent_location_before_action = state.environment_data['agent_location']
        for tile_key, position in state.environment_data['tiles'].items():
            if position == self.agent_destination:
                new_state.environment_data['tiles'][tile_key] = agent_location_before_action
            else:
                new_state.environment_data['tiles'][tile_key] = position

        return new_state


class State(object):

    def __init__(self, environment_data):
        self.environment_data = environment_data

    def __eq__(self, other):
        return type(self) is type(other) and self.environment_data == other.environment_data

    def __ne__(self, other):
        return not self == other

    def __hash__(self):
        return hash(frozenset(self.environment_data))


class Agent(object):

    def __init__(self, name, skill, search_algorithm):
        self.name = name
        self.skill = skill
        self.search_algorithm = search_algorithm


class Grid(object):

    def __init__(self, grid_data):
        self.grid_data = tuple(tuple(row) for row in grid_data)
        self.width = len(self.grid_data[0])
        self.height = len(self.grid_data)

    def is_passable(self, position):
        return self.grid_data[position.y][position.x] != 0
```

Appendix 6 – Proof of Negamax Operation

```
Algorithm: negamax
Alpha: N/A
Beta: N/A
Num iterations: 31
Search depth: 3
Player 1 is O, Player 2 is X

Board State:

.  X  .  X  .  X  X
.  X  .  X  .  X  O
.  O  .  O  .  O  X
.  X  .  X  .  X  O
O  O  .  X  O  O  O
O  O  X  O  X  O  O

1  2  3  4  5  6  7


Turn: Player 2
Chosen action: column 5
Score: 15

==== Action Tree ====

Node 1      Player 2  |  Action: column 1  |  Score: -inf
Node 2                  Player 1  |  Action: column 1  |  Score: -18
Node 3                              Player 2  |  Action: column 1  |  Score: 11
Node 4                              Player 2  |  Action: column 3  |  Score: 15
Node 5                              Player 2  |  Action: column 5  |  Score: 18
Node 6                  Player 1  |  Action: column 3  |  Score: -inf
Node 7                              Player 2  |  Action: column 1  |  Score: 9
Node 8                              Player 2  |  Action: column 3  |  Score: inf
Node 9                              Player 2  |  Action: column 5  |  Score: 15
Node 10                 Player 1  |  Action: column 5  |  Score: inf

Node 11     Player 2  |  Action: column 3  |  Score: -inf
Node 12                Player 1  |  Action: column 1  |  Score: -21
Node 13                            Player 2  |  Action: column 1  |  Score: 15
Node 14                            Player 2  |  Action: column 3  |  Score: 21
Node 15                            Player 2  |  Action: column 5  |  Score: 21
Node 16                Player 1  |  Action: column 3  |  Score: inf
Node 17                Player 1  |  Action: column 5  |  Score: inf

Node 18     Player 2  |  Action: column 5  |  Score: 15
Node 19                Player 1  |  Action: column 1  |  Score: -inf
Node 20                            Player 2  |  Action: column 1  |  Score: 18
Node 21                            Player 2  |  Action: column 3  |  Score: 21
Node 22                            Player 2  |  Action: column 5  |  Score: inf
Node 23                Player 1  |  Action: column 3  |  Score: -inf
Node 24                            Player 2  |  Action: column 1  |  Score: 15
Node 25                            Player 2  |  Action: column 3  |  Score: inf
Node 26                           Player 2  |  Action: column 5  |  Score: inf
Node 27                Player 1  |  Action: column 5  |  Score: -15
Node 28                            Player 2  |  Action: column 1  |  Score: 12
Node 29                            Player 2  |  Action: column 3  |  Score: 15
Node 30                            Player 2  |  Action: column 5  |  Score: 15
```

Appendix 7 – Proof of Alpha-Beta Negamax Operation

```
Algorithm: negamax_ab
Alpha: -inf
Beta: inf
Num iterations: 28
Search depth: 3
Player 1 is O, Player 2 is X


.  X  .  X  .  X  X
.  X  .  X  .  X  O
.  O  .  O  .  O  X
.  X  .  X  .  X  O
O  O  .  X  O  O  O
O  O  X  O  X  O  O

1  2  3  4  5  6  7


Turn: Player 2
Chosen action: column 5
Score: 15

==== Action Tree ====

Node 1      Player 2  |  Action: column 1  |  Score: -inf  |  Alpha: -inf  |  Beta: inf
Node 2                Player 1  |  Action: column 1  |  Score: -18  |  Alpha: -18  |  Beta: inf
Node 3                          Player 2  |  Action: column 1  |  Score: 11  |  Alpha: 11  |  Beta: inf
Node 4                          Player 2  |  Action: column 3  |  Score: 15  |  Alpha: 15  |  Beta: inf
Node 5                          Player 2  |  Action: column 5  |  Score: 18  |  Alpha: 18  |  Beta: inf
Node 6                Player 1  |  Action: column 3  |  Score: -inf  |  Alpha: -18  |  Beta: inf
Node 7                          Player 2  |  Action: column 1  |  Score: 9  |  Alpha: 9  |  Beta: 18
Node 8                          Player 2  |  Action: column 3  |  Score: inf  |  Alpha: inf  |  Beta: 18
Node 9                Player 1  |  Action: column 5  |  Score: inf  |  Alpha: inf  |  Beta: inf

Node 10     Player 2  |  Action: column 3  |  Score: -inf  |  Alpha: -inf  |  Beta: inf
Node 11               Player 1  |  Action: column 1  |  Score: -21  |  Alpha: -21  |  Beta: inf
Node 12                         Player 2  |  Action: column 1  |  Score: 15  |  Alpha: 15  |  Beta: inf
Node 13                         Player 2  |  Action: column 3  |  Score: 21  |  Alpha: 21  |  Beta: inf
Node 14                         Player 2  |  Action: column 5  |  Score: 21  |  Alpha: 21  |  Beta: inf
Node 15               Player 1  |  Action: column 3  |  Score: inf  |  Alpha: inf  |  Beta: inf

Node 16     Player 2  |  Action: column 5  |  Score: 15  |  Alpha: 15  |  Beta: inf
Node 17               Player 1  |  Action: column 1  |  Score: -inf  |  Alpha: -inf  |  Beta: inf
Node 18                         Player 2  |  Action: column 1  |  Score: 18  |  Alpha: 18  |  Beta: inf
Node 19                         Player 2  |  Action: column 3  |  Score: 21  |  Alpha: 21  |  Beta: inf
Node 20                         Player 2  |  Action: column 5  |  Score: inf  |  Alpha: inf  |  Beta: inf
Node 21               Player 1  |  Action: column 3  |  Score: -inf  |  Alpha: -inf  |  Beta: inf
Node 22                         Player 2  |  Action: column 1  |  Score: 15  |  Alpha: 15  |  Beta: inf
Node 23                         Player 2  |  Action: column 3  |  Score: inf  |  Alpha: inf  |  Beta: inf
Node 24               Player 1  |  Action: column 5  |  Score: -15  |  Alpha: -15  |  Beta: inf
Node 25                         Player 2  |  Action: column 1  |  Score: 12  |  Alpha: 12  |  Beta: inf
Node 26                         Player 2  |  Action: column 3  |  Score: 15  |  Alpha: 15  |  Beta: inf
Node 27                         Player 2  |  Action: column 5  |  Score: 15  |  Alpha: 15  |  Beta: inf
```

Appendix 8.1 – Main Script
(Author's Code)

```
import numpy as np
from connectfour import BoardState
from search import *

def clear_screen():
    print('\n' * 20)

def play_game(board=BoardState(), search_depth=4, players=('cpu', 'cpu'), player=1, search_algo=negamax,
ui=True):

    def get_winner():
        if board.end_state() == 'lose':
            winner = 1 if board.last_player == 1 else 2
        else:
            winner = None
        return winner

    def print_winner():

        winner = get_winner()
        if winner:
            print('=' * 8, 'Player', winner, 'Wins!', '=' * 8, )
        else:
            print('=' * 10, 'It''s a draw', '=' * 10)

    def get_action():

        def get_input():
            action = None

            player_num = '1' if player == 1 else '2'

            while action is None:

                try:
                    user_input = int(input('P%s - Which column?\n' % player_num))
                    if 1 <= user_input <= BoardState.WIDTH:
                        action = user_input - 1
                    else:
                        print('Oops - that''s not a valid number. Enter 1-%d' % BoardState.WIDTH)
                except ValueError:
                    print('Oops - that''s not a valid number. Enter 1-%d' % BoardState.WIDTH)
            return action

        if player == 1:
            if players[0] == 'human':
                action = get_input()
            else:
                action = search_algo(state=board, player=player, remaining_plys=search_depth)[1]
        else:
            if players[1] == 'human':
                action = get_input()
            else:
                action = search_algo(state=board, player=player, remaining_plys=search_depth)[1]

        return action

    if ui:
        clear_screen()
        print(board, '\n')

    while board.end_state() is None:

        action = get_action()

        if ui: clear_screen()
        board = board.execute_action(action)

        if ui: print(board, '\n')

        player = -player
```

```
    if ui: print_winner()

    return get_winner(), search_algo.counter

search_depth = 3
players = ('cpu', 'cpu')
search_algos = [negamax, negamax_ab, negamax_ab_with_ordering]

for search_depth in range(1, 8):
    game_results = []
    iterations = []
    print('Depth:', search_depth)
    for algo in search_algos:
        for i in range(50):
            algo.counter = 0
            winner, counter = play_game(search_depth=search_depth, players=players, search_algo=algo,
ui=False)
            game_results.append(winner)
            iterations.append(algo.counter)

        print('Algo:', algo.__name__)
        print('Negamax iter:', iterations)
        print('Negamax results:', game_results)
        print()


"""
.  X  .  X  .  X  X
.  X  .  X  .  X  O
.  O  .  O  .  O  X
.  X  .  X  .  X  O
O  O  .  X  O  O  O
O  O  X  O  X  O  O
"""

board = [
    [1, 1, 0, 0, 0, 0],
    [1, 1, -1, 1, -1, -1],
    [-1, 0, 0, 0, 0, 0],
    [1, -1, -1, 1, -1, -1],
    [-1, 1, 0, 0, 0, 0],
    [1, 1, -1, 1, -1, -1],
    [1, 1, 1, -1, 1, -1]
]

board = np.array(board, dtype=np.int8)
col_counts = [2, 6, 1, 6, 2, 6, 6]
player = -1
board = BoardState(board=board, column_counts=col_counts,player=-player, last_action_coords=None)
search_algo = negamax

score, action = search_algo(state=board, player=player, remaining_plys=search_depth)
print('Algorithm:', search_algo.__name__)
print('Alpha: -inf')
print('Beta: inf')
print('Num iterations:', search_algo.counter)
print('Player 1 is ''O'', Player 2 is ''X''', '\n')
print(board)
print_graph(search_algo.edges, score, action, player)
```

Appendix 8.2 – Connect Four
(Author's Code)

```
import numpy as np
from operator import itemgetter

class BoardState():

    WIDTH = 7
    HEIGHT = 6
    DIRECTIONS = ((1,0), (0,1), (1, 1), (1, -1))
    WEIGHTS = np.array(
        [
            [3, 4, 5, 5, 4, 3],
            [4, 6, 8, 8, 6, 4],
            [5, 8, 11, 11, 8, 5],
            [7, 10, 13, 13, 10, 7],
            [5, 8, 11, 11, 8, 5],
            [4, 6, 8, 8, 6, 4],
            [3, 4, 5, 5, 4, 3]
        ],
        dtype=np.int8
    )

    counter = -1

    def __init__(self, board=None, column_counts=None, player=None, last_action_coords=None):
        """
        :param board: A byte array
        """
        if board is not None:
            self.board = board
            self.column_counts = column_counts
            self.last_player = player
            self.next_player = -self.last_player
            self.last_action_coords = last_action_coords
        else:
            self.board = np.zeros(shape=(BoardState.WIDTH, BoardState.HEIGHT), dtype=np.int8)
            self.column_counts = tuple(0 for i in range(BoardState.WIDTH))
            self.last_player = -1
            self.next_player = -self.last_player
            self.last_action_coords = None

        self.id = BoardState.counter
        BoardState.counter += 1

    def evaluate(self):
        """
        Evaluates the current position
        :return: A signed integer. Higher is better for Player-1
        """

        return np.sum(np.multiply(BoardState.WEIGHTS, self.board))

    def possible_actions(self):
        """
        Gets a list of all legal moves
        :return: A list of actions
        """

        actions = (i for i, count in enumerate(self.column_counts) if count < BoardState.HEIGHT)
        return actions

    def sorted_possible_actions(self):
        """
        Gets a list of all legal moves
        :return: A list of actions
        """

        actions = [i for i, count in enumerate(self.column_counts) if count < BoardState.HEIGHT]
        actions.sort(key=lambda action: BoardState.WEIGHTS[action, self.column_counts[action]], reverse=True)
        return actions

    def execute_action(self, action):
        """
```

```
        Executes an action and returns a new board state
        :return: A new BoardState object
        """
        assert self.column_counts[action] < BoardState.HEIGHT, "Cannot make a move in this column"

        new_col_counts = tuple(count + 1 if i == action else count for i, count in
enumerate(self.column_counts))

        new_board = np.copy(self.board)
        new_board[action, self.column_counts[action]] = self.next_player

        return BoardState(new_board, new_col_counts, self.next_player, (action, new_col_counts[action] - 1))

    def end_state(self):
        """
        Determines if the game has come to an end
        :return:
        """

        if self.last_action_coords:
            # CHECK WIN
            for direction in BoardState.DIRECTIONS:
                num_in_a_row = 0
                for i in range(-3, 4):
                    offset = (direction[0] * i, direction[1] * i)
                    nearby_chip_coords = (offset[0] + self.last_action_coords[0], offset[1] +
self.last_action_coords[1])
                    if 0 <= nearby_chip_coords[0] < BoardState.WIDTH and 0 <= nearby_chip_coords[1] <
BoardState.HEIGHT:
                        if self.board[nearby_chip_coords[0], nearby_chip_coords[1]] == self.last_player:
                            num_in_a_row += 1
                        else:
                            num_in_a_row = 0

                        if num_in_a_row == 4:
                            return 'lose'

            # CHECK DRAW AFTER CHECKING WIN
            if all(count == BoardState.HEIGHT for count in self.column_counts):
                return 'draw'

        # OTHERWISE
        return None

    def __str__(self):
        chars = ('.', 'O', 'X')
        board_str = ''
        for row in reversed(tuple(zip(*self.board))):
            board_str += '  '.join((chars[slot] for slot in row)) + '\n'
        return board_str
```

Appendix 8.3 – Search
(Author's Code)

```
from random import choice as rand_choice
from collections import deque

def negamax(state, remaining_plys, player):

    negamax.counter += 1

    # BASE CASES
    # Is a leaf node
    end_state = state.end_state()
    if end_state:
        if end_state == 'lose':
            return -float('inf'), None
        else:
            return 0, None

    # Is max depth
    if remaining_plys == 0:
        return player * state.evaluate(), None

    # RECURSIVE CASE
    best_score_yet = -float('inf')
    best_actions_yet = []

    for possible_action in state.possible_actions():

        child_state = state.execute_action(possible_action)

        child_state_score = -negamax(child_state, remaining_plys - 1, -player)[0]

        negamax.edges.setdefault(state.id, []).append( (child_state.id, possible_action, child_state_score,
player, None, None) )

        if child_state_score > best_score_yet:
                best_score_yet = child_state_score
                best_actions_yet = [possible_action]
        elif child_state_score == best_score_yet:
                best_actions_yet.append(possible_action)

    return best_score_yet, rand_choice(best_actions_yet)


def negamax_ab(state, remaining_plys, player, alpha=-float('inf'), beta=float('inf')):

    negamax_ab.counter += 1

    # BASE CASES
    # Is a leaf node
    end_state = state.end_state()
    if end_state:
        if end_state == 'lose':
            return -float('inf'), None
        else:
            return 0, None

    # Is max depth
    if remaining_plys == 0:
        return player * state.evaluate(), None

    # RECURSIVE CASE
    best_score_yet = -float('inf')
    best_actions_yet = []

    for possible_action in state.possible_actions():

        child_state = state.execute_action(possible_action)

        child_state_score = -negamax_ab(child_state, remaining_plys - 1, -player, -beta, -alpha)[0]

        if child_state_score > best_score_yet:
                best_score_yet = child_state_score
                best_actions_yet = [possible_action]
```

```
        elif child_state_score == best_score_yet:
                best_actions_yet.append(possible_action)

        alpha = max(alpha, best_score_yet)

        negamax_ab.edges.setdefault(state.id, []).append( (child_state.id, possible_action, child_state_score,
player, alpha, beta) )

        if alpha >= beta:
            break

    return best_score_yet, rand_choice(best_actions_yet)


def negamax_ab_with_ordering(state, remaining_plys, player, alpha=-float('inf'), beta=float('inf')):

    negamax_ab_with_ordering.counter += 1

    # BASE CASES
    # Is a leaf node
    end_state = state.end_state()
    if end_state:
        if end_state == 'lose':
            return -float('inf'), None
        else:
            return 0, None

    # Is max depth
    if remaining_plys == 0:
        return player * state.evaluate(), None

    # RECURSIVE CASE
    best_score_yet = -float('inf')
    best_actions_yet = []

    for possible_action in state.sorted_possible_actions():

        child_state = state.execute_action(possible_action)

        child_state_score = -negamax_ab_with_ordering(child_state, remaining_plys - 1, -player, -beta, -
alpha)[0]

        if child_state_score > best_score_yet:
                best_score_yet = child_state_score
                best_actions_yet = [possible_action]
        elif child_state_score == best_score_yet:
                best_actions_yet.append(possible_action)

        alpha = max(alpha, best_score_yet)

        negamax_ab_with_ordering.edges.setdefault(state.id, []).append( (child_state.id, possible_action,
child_state_score, player, alpha, beta) )

        if alpha >= beta:
            break

    return best_score_yet, rand_choice(best_actions_yet)


def print_graph(edge_dict, score, action, player):

    def get_player_name():
        return 1 if player == 1 else 2

    depth = 0
    first_node_id = 0
    queue = deque([(first_node_id, depth, score, action, player, None, None)])
    rows_printed = 1
    print('Turn: Player %d' % (get_player_name(),))
    print('Chosen action: column', action + 1)
    print('Score:', score, '\n')
    print('==== Action Tree ====', '\n')

    while queue:

        node_id, depth, score, action, player, alpha, beta = queue.pop()
        player_name = get_player_name()
```

```
        if depth > 0:
            if alpha:
                print("Node %d%sPlayer %d  |  Action: column %d  |  Score: %f  |  Alpha: %f  |  Beta: %f" %
                      (rows_printed, '\t\t' * (depth - 1), player_name, action + 1, score, alpha, beta))
            else:
                print("Node %d%sPlayer %d  |  Action: column %d  |  Score: %f" %
                      (rows_printed, '\t\t' * (depth - 1), player_name, action + 1, score))
            rows_printed += 1

        child_depth = depth + 1
        children = edge_dict.get(node_id, [])
        children.reverse()
        for child_id, action, child_score, player, alpha, beta in children:
            queue.append((child_id, child_depth, child_score, action, player, alpha, beta))



negamax.counter = 0
negamax.edges = {}

negamax_ab.counter = 0
negamax_ab.edges = {}

negamax_ab_with_ordering.counter = 0
negamax_ab_with_ordering.edges = {}
```