# Solving Blocks World Problems

Ofir Marcus
Department of Computer Science
Hebrew University
Jerusalem, Israel
ofirule@gmail.com

Eliezer Steinbock
Department of Computer Science
Hebrew University
Jerusalem, Israel
eliesteinbock@gmail.com

***Abstract*—We investigate different methods of solving Blocks World. We compare the results when using a number of different algorithms and heuristics and different table sizes.**

## I. Introduction

The blocks world is one of the most famous planning domains in artificial intelligence. The program was created by Terry Winograd and is a limited-domain natural-language system that can understand typed commands and move blocks around on a surface.

Imagine a set of cubes (blocks) sitting on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved.

The simplicity of this toy world lends itself readily to symbolic or classical A.I. approaches, in which the world is modeled as a set of abstract symbols which may be reasoned about. It is shown that the blocks world is NP-hard [1].

The classical blocks world problem assumes an infinite amount of space to place blocks on the table. In our program, we added an option to limit the amount of space on the table.

## II. Implementation

### A. Problem

A problem is defined as a start state, a goal state and a table size. Our aim is to get from the start state to the goal state using the permitted actions.

### B. States

A state consists of the positions of all the blocks and which block the crane is holding (which may be no block at all).

Each state is fully defined. We know which block is above each block (which may be none) and which block is below it (which may be not be a block, but the table).

The following is an example of a state: block A is resting on the table and is under block B. Block B is resting on block A and is under no block. The crane is holding no block.

### C. Actions

There are two possible actions by which we go from one state to another state: 1) pick up block X, 2) put down on block Y (or TABLE)

There are a number of constraints that determine whether we can perform an action or not.

A block may only be picked up if there is no block currently on top of it and the crane is currently not holding any block.

A block X may only be put on another block Y if block Y currently has no block on top of it.

A block may only be put on the table if the number of blocks currently on the table is less than the table size.

### D. Other notes

In our implementation, we assumed the start and goal states were fully defined and that the goal state was unique. Other implementations allow for multiple goal states and partially defined states, but we chose not to implement our program in such a way.

## III. Algorithms used to solve the problem

We used a number of algorithms to solve the problem. These include DFS, BFS, UCS, A* and simulated annealing. We used six different heuristics to solve the problem using A*. Below is a description of each algorithm and heuristic we used.

### A. DFS - Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring.

In our problem, each node represents a state. The root node is the start state. The goal node is the goal state. There is an edge on the graph for every pair of states for which there is an action that brings us from one state to the other.

Worst case performance is $O(b^m)$ and worst case space complexity is O(b*m) where b is the maximum branching factor and m is the maximum depth of the state space.

DFS does not necessarily find an optimal solution.

### *B. BFS - Breadth First Search*

Breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

BFS is similar to DFS, but first searches neighboring nodes (breadth) and only later searches the deeper nodes (depth).

Worst case performance is $O(b^{d+1})$ and worst case space complexity is $O(b^{d+1})$ where b is the maximum branching factor and d is the depth of the cheapest solution.

BFS always finds an optimal solution.

### *C. UCS – Uniform Cost Search*

Uniform-cost search (UCS) is a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node. The uniform-cost search is complete and optimal if the cost of each step exceeds some positive bound ε [2]. The worst-case time and space complexity is $O(b^{1 + C^*/\varepsilon})$, where C* is the cost of the optimal solution. When all step costs are equal, this becomes $O(b^{d+1})$ [3].

UCS is the same as A* with a zero heuristic (i.e. the heuristic function always returns 0).

### *D. A**

A* is an algorithm that is widely used in path-finding and graph traversal, the process of plotting an efficiently traversable path between points. Noted for its performance and accuracy, it enjoys widespread use.

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first described the algorithm in 1968 [4]. It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance (with respect to time) by using heuristics.

A* is always complete and it is optimal on trees when the heuristic is admissible and optimal on graphs when the heuristic is both admissible and consistent.

All our heuristics are both admissible and consistent and therefore the solutions we find using A* are always optimal.

Below is a list of the heuristics we used to solve blocks world problems:

- Heuristic 1 – this heuristic calculates the number of blocks that are currently not in the correct 'position'. We do not count a block if it is currently in the arm of the crane.

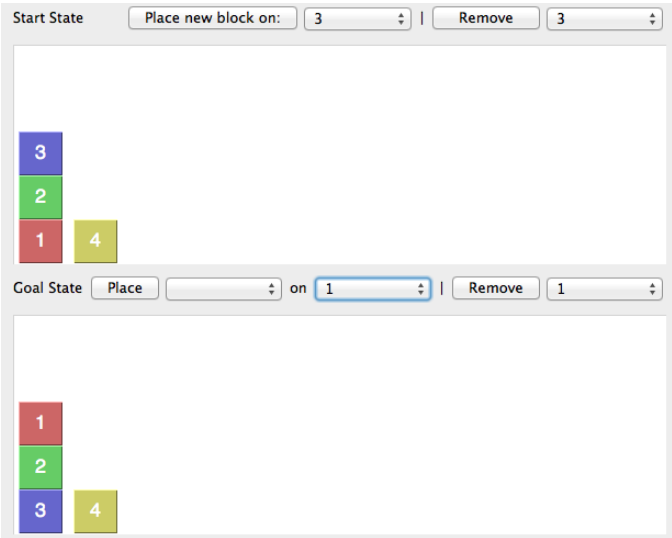



*Illustration 1: Heuristic 1 Example*

- Example of how Heuristic 1 (h1) works on Illustration 1: h1 gives a score of 3 to the start state. Blocks 1, 2 and 3 are all in incorrect positions giving us a score of 3. Block 4 is in the correct position, so we don't add anything for Block 4.

- Heuristic 2 – this heuristic is similar to Heuristic 1. It calculates the difference between the current state and the goal state, but looks at the details of each block. If Block A in the goal state is supposed to be on top of Block B and under Block C and in the current state it is neither on top of B or under C, then we add 2 to the heuristic. If it is over B, but not under C, we add only 1 to the heuristic. If it is both on top of B and under C in the current state, we add 0 to the heuristic. Below is an example of Heuristic 2 and how it differs from Heuristic 1.

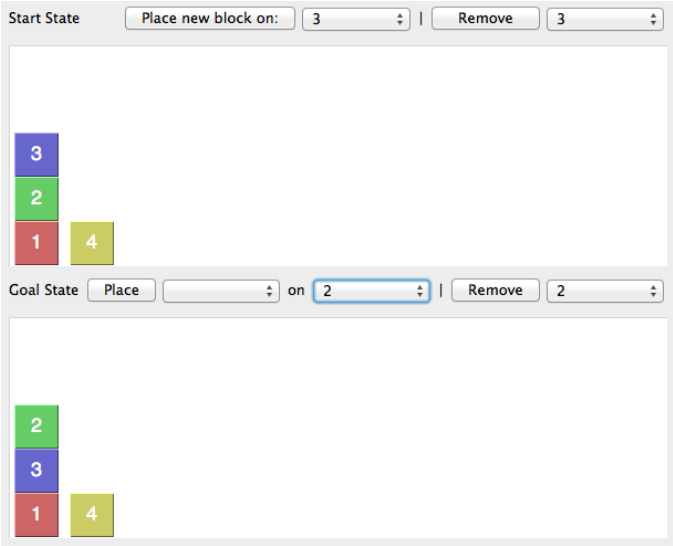



*Illustration 2: Heuristic 2 Example*

- Heuristic 2 gives a score of 5 to the above problem (Illustration 2). The calculation is as follows: Block 1 is on the table in both the goal and start states, but is not directly under Block 3 in the start state although it is directly under Block 3 in the goal state. This adds 1 to the score. Block 2 is neither on Block 3 nor clear in the start state. This adds 2 to the score. Block 3 is neither on Block 1 nor under Block 2 in the start state. This adds 2 to the score. Block 4 is on the table and clear in both the start and goal state. This adds 0 to the score. This gives us a total score of 1+2+2+0=5.
- Heuristic 1 gives a score of 3 to the start state in above problem (Illustration 2). The calculation is as follows: Blocks 1, 2 and 3 are not in the correction positions and block 4 is. This gives a total score of 1+1+1+0=3.
- Heuristic 3 – This heuristic adds 2 for every block that is not currently directly on top of the block on which it has to be in the goal state or if there is such a block somewhere below it (in the same pile). Below is an example using heuristic 3.

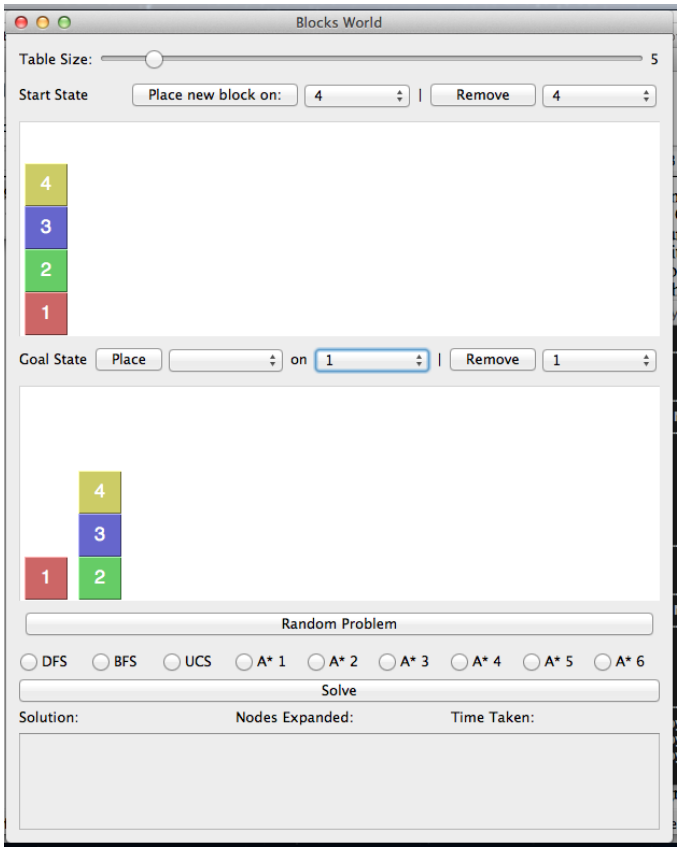



*Illustration 3: Heuristic 3 Example*

- The heuristic calculated by h3 in the start state shown in Illustration 3 is 6. This is because block 2 is in the wrong position and blocks 3 and 4 are somewhere above block 2, so for blocks 2, 3 and 4 we add 2 to the heuristic, which gives us 3*2 = 6.
- Heuristic 4 – this heuristic is twice the number of blocks that must be moved once plus four times the number of blocks that must be moved twice. A block that must be moved once is a block that is currently on a block different to the block upon which it rests in the goal state or a block that has such a block somewhere below it in the same pile. A block that must be moved twice is a block that is currently on the block upon which it must be placed in the goal state, but that block is a block that must be moved or if there exists a block that must be moved twice somewhere below it (in the same pile). A block that falls under both definitions is considered a block that is moved twice only. Below is an example.

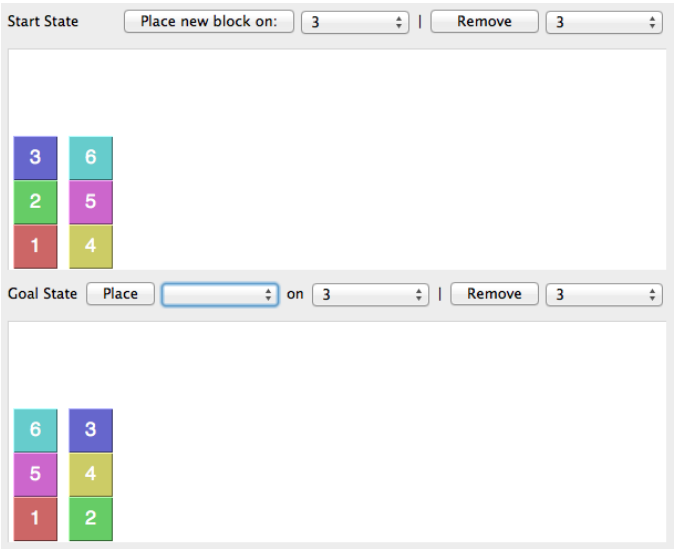



*Illustration 4: Heuristic 4 Example*

- In Illustration 4, B1 doesn't have to be moved (+=0). B2, B4 and B5 have to be moved once (+=(3*2)). B3 and B6 have to be moved twice because they are both above the block upon which they rest in the goal state (+=(2*4)). This gives a total score of: 0+6+8=12.
- Heuristic 5 – If blocks A and B are preventing each other from being moved to their goal positions, we call this a case of *mutual prevention*. This heuristic works the same way as heuristic 3, but adds 2 for each case of mutual prevention. Illustration 5 is a graphical example of a case of mutual prevention.

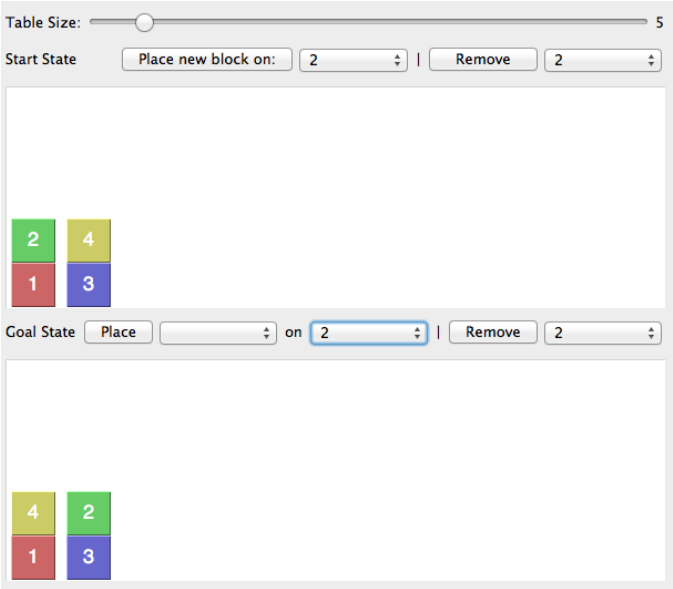



*Illustration 5: Mutual Prevention Example*

- The start state in Illustration 5 has one mutual prevention, since blocks 2 and 4 are preventing one another from reaching their goal positions. The heuristic calculated for the start state will be 6. 2 for the mutual prevention, 2 to lift up block 2 and 2 to lift up block 4.
- Heuristic 6 – This heuristic is a combination of heuristics 4 and 5, except when there is a block that must be moved twice (heuristic 4) and is also one of the blocks in a mutual prevention (heuristic 5), we don't add anything for this mutual prevention. Below is an example.

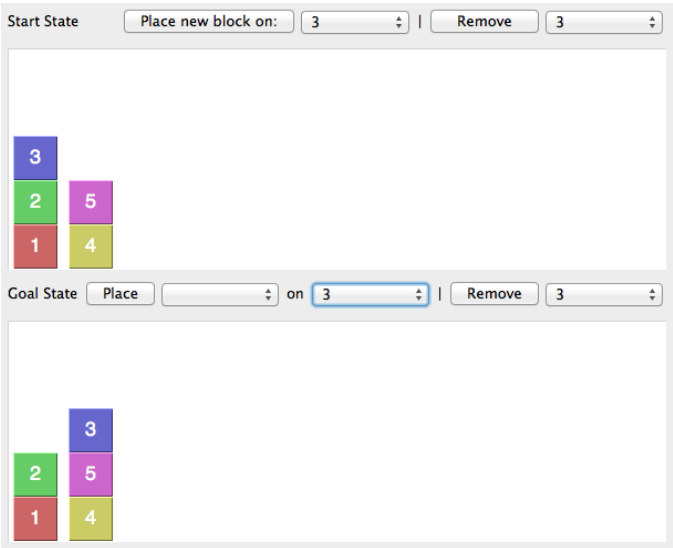



*Illustration 6: Heuristic 6 Example*

- Heuristic 6 gives a score of 14 to the start state shown in Illustration 6. Blocks 1, 3 and 5 need to be moved once (+=(3*2)). Blocks 2 and 4 need to be moved twice (+=(2*4)). There exists a mutual prevention between blocks 2 and 5, but Block 2 must be moved twice, so we don't add anything for this mutual prevention (+=0). In total we have a score of 6+8+0 = 14.

### E. *Simulated Annealing*

Simulated annealing (SA) is a generic probabilistic meta-heuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects, both are attributes of the material that depend on its thermodynamic free energy.

This notion of slow cooling is implemented in the Simulated Annealing algorithm as a slow decrease in the probability of accepting worse solutions as it explores the solution space. Accepting worse solutions is a fundamental property of meta-heuristics because it allows for a more extensive search for the optimal solution.

After making a lot of changes to the algorithm (we changed the starting temperature, cooling factor, the ending temperature and the state evaluation heuristic) we found that Simulated Annealing does not produce good results and takes a long time to run.

After a little bit of testing using Simulated Annealing, we realized that it was not an effective algorithm for solving blocks world problems and we decided not to include it in the analysis below.

## IV. Analysis

We analyzed the different algorithms on a range of problems. We changed the table size and the number of blocks to see how the algorithms dealt with the different conditions. We also analyzed whether decreasing the table size made the problems easier or harder to solve and how it affected the running time of the different heuristics. Below are the results of our analysis.

### A. *Simple Test*

We created a random problem of 5 blocks and tested each algorithm against it.

Below is a picture of a random problem and the solution returned by A* using DFS:

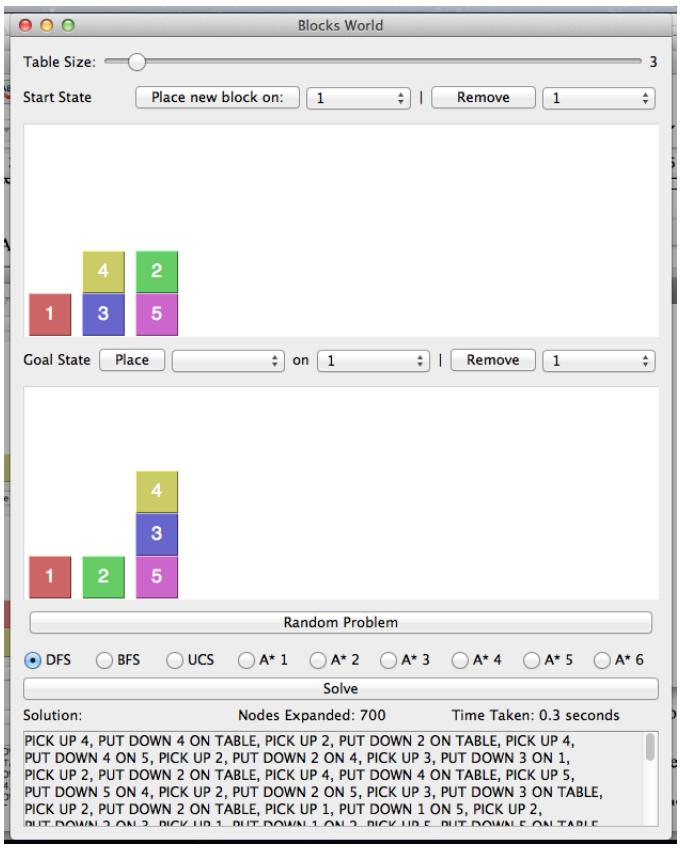



*Illustration 7: Simple Test*

We can see that this relatively simple problem was solved in 0.3 seconds using DFS and 700 nodes were expanded in finding this solution. The table size in the above example is 3, which can be seen at the top of the window above.

Note the above solution is not the optimal solution. The optimal solution consists of only 8 actions and DFS found a solution to the problem of length greater than 50. (All the other algorithms returned an optimal solution as expected).

Here are the results of running the other algorithms on this problem:

| Algorithm | Nodes Expanded | Time Taken (s) |
|---|---|---|
| **DFS** | 700 | 0.3 |
| **BFS** | 425 | 0.18 |
| **UCS** | 460 | 0.22 |
| **A* 1** | 63 | 0.03 |
| **A* 2** | 50 | 0.02 |
| **A* 3** | 19 | 0.01 |
| **A* 4** | 10 | 0.01 |
| **A* 5** | 10 | 0.01 |
| **A* 6** | 19 | 0.01 |

*Table 1: Results for the simple test*

This is just one example, but from it we can already see that the A* algorithms are much faster at solving blocks world problems (as would be expected) than the other algorithms and that the more complex heuristics find solutions in a shorter amount of time. (A* 1 refers to running A* using Heuristic 1).

### *B. More complex tests*

To truly test how good each heuristic is, we did multiple tests on many different scenarios. Below are the results of our tests. The average is taken over a set of 10 random problems. Testing on a larger sample set would have been better, but due to time constraints, we tested on only 10 problems. Each heuristic was tested on the same problems. The table size in the tests below is equal to the number of blocks we have (which is equivalent to a table of unlimited size).

Due to time constraints, we stopped a heuristic from continuing on a problem if it had already expanded 3000 nodes.

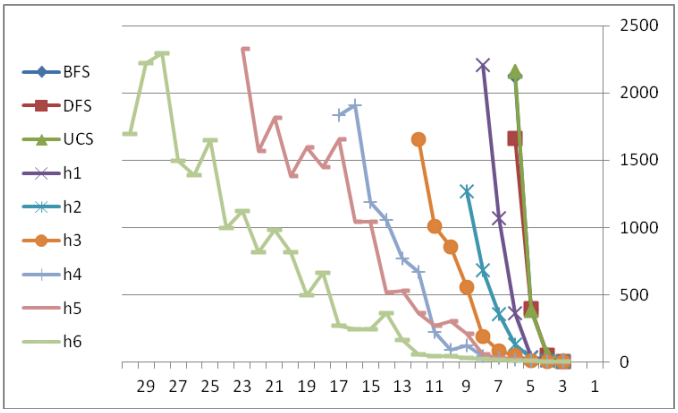
| Algorithm | No. of Blocks | Average Nodes Expanded | Average Time Taken (s) |
|---|---|---|---|
| **A* 1** | 5 | 59 | 0.020 |
| **A* 2** | 5 | 29 | 0.009 |
| **A* 3** | 5 | 13 | 0.004 |
| **A* 4** | 5 | 10 | 0.004 |
| **A* 5** | 5 | 13 | 0.004 |
| **A* 6** | 5 | 10 | 0.004 |
| **A* 1** | 10 | 3000 | 12.5 |
| **A* 2** | 10 | 2137 | 8.68 |
| **A* 3** | 10 | 864 | 2.25 |
| **A* 4** | 10 | 285 | 0.489 |
| **A* 5** | 10 | 123 | 0.122 |
| **A* 6** | 10 | 59 | 0.065 |
| **A* 1** | 15 | - | - |
| **A* 2** | 15 | - | - |
| **A* 3** | 15 | 2837 | 17.2 |
| **A* 4** | 15 | 1066 | 5.3 |
| **A* 5** | 15 | 1148 | 5.9 |
| **A* 6** | 15 | 122 | 0.264 |

*Table 2: Multiple Tests on the different heuristics*

A* 1 and A* 2 already run very slowly for problems with only 10 blocks, so we didn't do tests on them for larger problems. A* 1 was unable to solve any of the problems with 10 blocks in them using less than 3000 nodes.

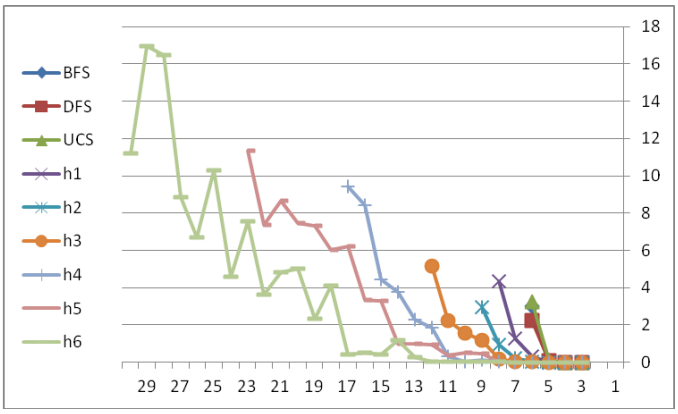
We can see that as the problems get harder, there is a clear distinction between how good each heuristic is. As was expected, the more complex heuristics provide much better results.

The following graph shows the number of nodes expanded in running each algorithm over twenty random problems of different a varying number of blocks. Each heuristic was tested on the same problems. Due to time constraints, we stopped a heuristic from continuing on a problem if it had already expanded 3000 nodes. The x-axis of the graph below is the number of blocks in the problem. The y-axis is the number of nodes expanded.




*Graph 1: Nodes expanded by each algorithm on different sized problems (number of blocks on the x-axis and number of nodes on the y-axis).*

The following graph is the same as the above except that it is a plot of the number of blocks against time taken to solve the problem, in seconds (as opposed to number of nodes).




*Graph 2: Nodes expanded by each algorithm on different sized problems (number of blocks on the x-axis and time on the y-axis).*

We can see once again that the more complex heuristics have better results.

Surprisingly, h4 (A* using Heuristic 4) performs consistently better than h5 on smaller problems (problems with 12 blocks or less in them), but h5 solves larger problems much faster than h4.

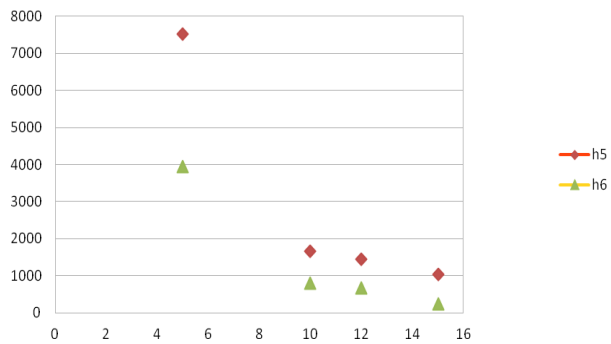
Overall, h6 is by far the best heuristic and can solve problems of up to 30 blocks fairly quickly. H6 was able to solve a random problem with 40 blocks in 109 seconds and it expanded 6427 nodes in the process. When we ran h6 on a random problem with 50 blocks, it was unable to find a solution using less than 10,000 nodes.

### C. *Varying Table Sizes*

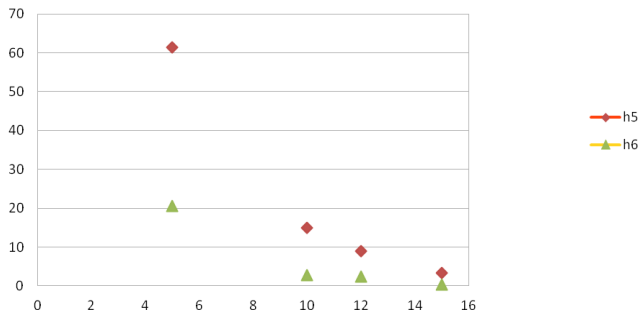
We tested how long it took A* to solve problems with 15 blocks using heuristics 5 and 6. The other algorithms and heuristics for A* were unable to solve problems of 15 blocks and a table size of 5 or 10 without expanding over 10,000 nodes. Surprisingly, despite h4 seemingly being quite a good heuristic, it too was unable to solve problems with 15 blocks and a table size of 10 while expanding less than 10,000 nodes.

As can be seen in the graphs below, h6 performs better than h5, which is to be expected. We also see a correlation between the table size and the number of nodes expanded (and the amount of time). The greater the table size, the quicker the time in which the problem can be solved. On the one hand this is intuitive, because the problem becomes easier when we increase the table space. On the other hand though, the increase in speed is not immediately obvious due to the fact that an increased table size also means an increased state space over which each algorithm must traverse.

The graphs below show the average number of nodes expanded and time taken to solve problems of 15 blocks and varying table size. We ran 10 tests for each table size.




*Graph 3: Effects of varying the table size. The x-axis is the table size and the y-axis is the number of nodes expanded.*




*Graph 4: Effects of varying the table size. The x-axis is the table size and the y-axis is the time taken to solve the problem in seconds.*

## V. Conclusions and Future Work

In our analysis, we found which algorithms are good at solving blocks world problems and which are not. We also saw the effects of writing good heuristics and how a good heuristic can speed up the solving of a blocks world problem by a factor of at least 20 and in some cases a lot more.

The ideas in this paper could be expanded upon in future work. A limit could be placed on the maximum height of a pile of blocks. The effects of multiple cranes and different size blocks or even different shaped objects could be investigated. The blocks world problem could also be modified to consider the cost of raising each block, with heavier blocks costing more to raise.

## Acknowledgment

We'd like to thank our teachers Jeff Rosenschein and Yoad Lewenberg for teaching the Artificial Intelligence course this semester and enriching our own intelligence on the topic.

## References

[1] Stephen V. Chenowet, “ON THE NP-HARDNESS OF BLOCKS WORLD”.

[2] Stuart Russell; Peter Norvig (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2

[3] Stuart Russell; Peter Norvig (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall. ISBN 978-0-13-604259-4.

[4] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107.