COMP2208 Search Methods Report - Peter Johnstone

Approach

Language choice

I decided to use Java to solve this problem because of my familiarity with the language and because I wanted to take an object-oriented approach.

Classes

State

The State class represents the state of the Blocksworld puzzle as a 2D char array, stores the coordinates of the agent, represented on the grid by an asterisk (*), in an array of size 2. A State can be constructed in 2 ways: by creating a Start State where the grid is set up like the initial state in the assignment brief, or by copying the details of an existing State. The State class also contained methods regarding the movement of the Agent, whether a move in a given direction is legal and whether the State is a goal state - where every lettered block is in the same position on the grid of the current State and goal State.

Node

A Node was the data type used in the fringes of BFS, DFS and IDS. It contained a State, its parent Node in the tree, its depth in the tree, and the last move on the grid taken to get from the parent Node state to the current Node state.

Algorithms

All algorithms are generic tree search algorithms - they start at a root node and explore a state space by adding nodes to a fringe of unexpanded nodes and expanding them according to a search strategy until a node containing a solution has been expanded. Every algorithm takes a start State and a goal State as inputs, and outputs a path from the start State to the goal State
The algorithms only add Nodes to the fringe if the move will not take the agent out of bounds. For example, if the agent is on the top row of the grid it will not consider a Node that tries to move the agent up.

Breadth First Search (BFS)

In BFS the fringe is a FIFO queue which was implemented using a LinkedList since it has queue functionality. This ensures that the shallowest unexpanded nodes are expanded first as new successors are added to the end of the queue.

Depth-First Search (DFS)

In DFS, the fringe is a LIFO stack, which was implemented using Java's Stack class. The deepest unexpanded nodes are expanded first since successors are put at the front. The child nodes of an expanded node are added to the stack in a random order. This is done to avoid getting stuck in an endless loop where the agent keeps going up until the top is reached, then down one, then up one etc.

Iterative Deepening Search (IDS)

IDS performs Depth-Limited Searches -a DFS that backtracks after a depth limit is reached- with increasing depth limits until a solution is found. Like DFS, the fringe is a Stack, and children of an expanded node are added in a random order.

A* Search

A* Search is a type of Best-First Search which uses an admissible heuristic to determine which node is expanded next. I decided to use the sum of the Manhattan distances between the current positions of the blocks and their correct positions as the heuristic (why?). Nodes were added to a PriorityQueue based on their Manhattan distances, with smaller Manhattan distances having higher priority.

Evidence

I ran each algorithm on a 4x4 start state that was 2 moves away from the goal state, so a solution would be of depth 2. This was so that I could show a detailed example output from start to finish. Every time a Node is expanded, its depth and last move are displayed. Every time a child node is generated and added to the fringe, a print statement is made.

Breadth-First Search evidence

```
Running Breadth-First Search
Start state:
----
-*--
ba--
-c--

Goal state:
----
-a--
-b--
-c-*

Expanded node #1
Depth: 0
Last move: START
State:
----
-*--
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe

Expanded node #2
Depth: 1
Last move: UP
State:
-*--
----
ba--
-c--

Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe

Expanded node #3
Depth: 1
Last move: DOWN
State:
----
-a--
b*--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe

Expanded node #4
Depth: 1
Last move: LEFT
State:
----
*---
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #5
Depth: 1
Last move: RIGHT
State:
----
--*-
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #6
Depth: 2
Last move: DOWN
State:
----
-*--
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #7
Depth: 2
Last move: LEFT
State:
*---
----
ba--
-c--

Add node with last move DOWN to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #8
Depth: 2
Last move: RIGHT
State:
--*-
----
ba--
-c--

Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #9
Depth: 2
Last move: UP
State:
----
-*--
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #10
Depth: 2
Last move: DOWN
State:
----
-a--
bc--
-*--

Add node with last move UP to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #11
Depth: 2
Last move: LEFT
State:
----
-a--
*b--
-c--

Goal state found
Nodes expanded: 11
Depth: 2
Path to solution: START, DOWN, LEFT
```

Depth-first search evidence

It should be noted that it took several attempts to get the algorithm to find the optimal solution.

```
Running Depth-First Search
Start state:
----
-*--
ba--
-c--

Goal state:
----
-a--
-b--
-c-*
```

```
Expanded node #1
Depth: 0
Last move: START
State:
----
-*--
ba--
-c--

Add node with last direction RIGHT to fringe
Add node with last direction LEFT to fringe
Add node with last direction UP to fringe
Add node with last direction DOWN to fringe
```

```
Expanded node #2
Depth: 1
Last move: DOWN
State:
----
-a--
b*--
-c--

Add node with last direction DOWN to fringe
Add node with last direction UP to fringe
Add node with last direction RIGHT to fringe
Add node with last direction LEFT to fringe
```

```
Expanded node #3
Depth: 2
Last move: LEFT
State:
----
-a--
*b--
-c--

Goal state found
Nodes expanded: 3
Depth: 2
Path to solution: START, DOWN, LEFT
```

A* search evidence

With A* search, I also included the Manhattan distance when printing nodes.

```
Running A* Search
Start state:
----
-*--
ba--
-c--

Goal state:
----
-a--
-b--
-c-*
```

```
Expanded node #1
Depth: 0
Distance from goal state: 2
Last move: START
State:
----
-*--
ba--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #2
Depth: 1
Distance from goal state: 1
Last move: DOWN
State:
----
-a--
b*--
-c--

Add node with last move UP to fringe
Add node with last move DOWN to fringe
Add node with last move LEFT to fringe
Add node with last move RIGHT to fringe
```

```
Expanded node #3
Depth: 2
Distance from goal state: 0
Last move: LEFT
State:
----
-a--
*b--
-c--

Goal state found
Nodes expanded: 3
Depth: 2
Path to solution: START, DOWN, LEFT
```

Iterative Deepening Search evidence

```
Running Iterative Deepening Search
Start state:
----
-*--
ba--
-c--

Goal state:
----
-a--
-b--
-c-*
```

```
DLS with depth limit 0
Nodes expanded so far: 0
Expanded node #1
Depth: 0
Last move: START
State:
----
-*--
ba--
-c--

No solution found. Increasing depth limit
```

```
DLS with depth limit 1
Nodes expanded so far: 1
Expanded node #2
Depth: 0
Last move: START
State:
----
-*--
ba--
-c--

Add node with last direction UP to fringe
Add node with last direction DOWN to fringe
Add node with last direction LEFT to fringe
Add node with last direction RIGHT to fringe
```

```
Expanded node #3
Depth: 1
Last move: RIGHT
State:
----
--*-
ba--
-c--
```

```
Expanded node #4
Depth: 1
Last move: LEFT
State:
----
*---
ba--
-c--
```

```
Expanded node #5
Depth: 1
Last move: DOWN
State:
----
-a--
b*--
-c--
```

```
Expanded node #6
Depth: 1
Last move: UP
State:
-*--
----
ba--
-c--

No solution found. Increasing depth limit
```

```
DLS with depth limit 2
Nodes expanded so far: 6
Expanded node #7
Depth: 0
Last move: START
State:
----
-*--
ba--
-c--

Add node with last direction UP to fringe
Add node with last direction RIGHT to fringe
Add node with last direction DOWN to fringe
Add node with last direction LEFT to fringe
```

```
Expanded node #8
Depth: 1
Last move: LEFT
State:
----
*---
ba--
-c--

Add node with last direction UP to fringe
Add node with last direction RIGHT to fringe
Add node with last direction DOWN to fringe
```

```
Expanded node #9
Depth: 2
Last move: DOWN
State:
----
b---
*a--
-c--
```

```
Expanded node #10
Depth: 2
Last move: RIGHT
State:
----
-*--
ba--
-c--
```

```
Expanded node #11
Depth: 2
Last move: UP
State:
*---
----
ba--
-c--
```

```
Expanded node #12
Depth: 1
Last move: DOWN
State:
----
-a--
b*--
-c--

Add node with last direction UP to fringe
Add node with last direction RIGHT to fringe
Add node with last direction LEFT to fringe
Add node with last direction DOWN to fringe
```

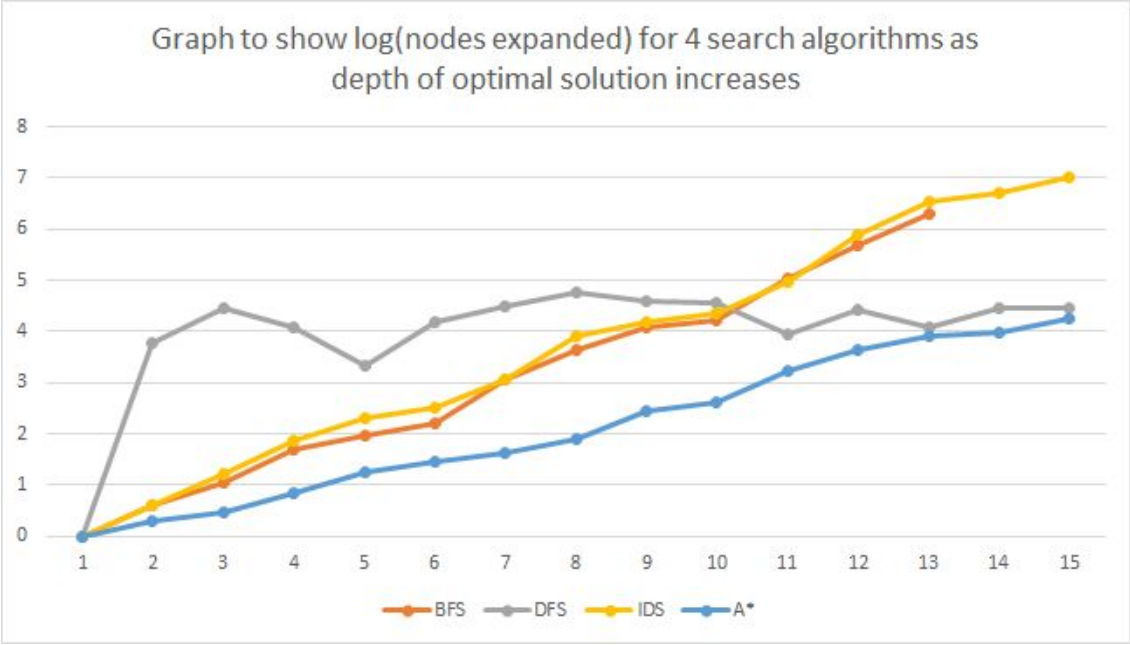
```
Expanded node #13
Depth: 2
Last move: DOWN
State:
----
-a--
bc--
-*--
```

```
Expanded node #14
Depth: 2
Last move: LEFT
State:
----
-a--
*b--
-c--

Goal state found
Nodes expanded: 14
Depth: 2
Path to solution: START, DOWN, LEFT
```

Scalability Study

Problem difficulty was controlled by moving the start state closer to the goal state so that the optimal solutions were on different depths of the tree. I ran each algorithm 5 times for each depth of least cost solution, and took their median value for nodes expanded. If an algorithm failed for a given depth, due to running out of memory or any other reason, there is no data point for that algorithm at that depth.
I originally plotted a graph with nodes expanded on the y axis and depth of optimal solution on the x axis. However, because the IDS generated 10323601 nodes for a solution at depth 14, that graph would not show anything valuable since that value is significantly bigger than any other. Instead, I plotted a graph of the logarithm of nodes expanded versus depth of optimal solution since it shows how the number of nodes generated increases in a much better way.



As the graph shows, A* search is the most scalable search algorithm of the four in terms of nodes expanded, as it consistently expands the fewest number of nodes. This is because the heuristic means that it will not expand nodes that take it further away from a goal state.

IDS generates the most number of nodes because it has to generate the previous tree after every iteration. Despite that, it is the second most scalable algorithm since it still finds an optimal solution at depths 13 and 14. Also, as shown in the extras section, it is the most efficient algorithm in terms of space complexity.

BFS failed to work for optimal solution depths of 13 and 14, since it would run out of memory after generating approximately 3,000,000 nodes. It is incredibly inefficient since it is a blind search that considers every node of a certain depth before going to a lower depth.

While DFS could still find solutions at higher depths, none of the solutions it found were optimal - it would simply pick random legal moves until it happened to find a solution.

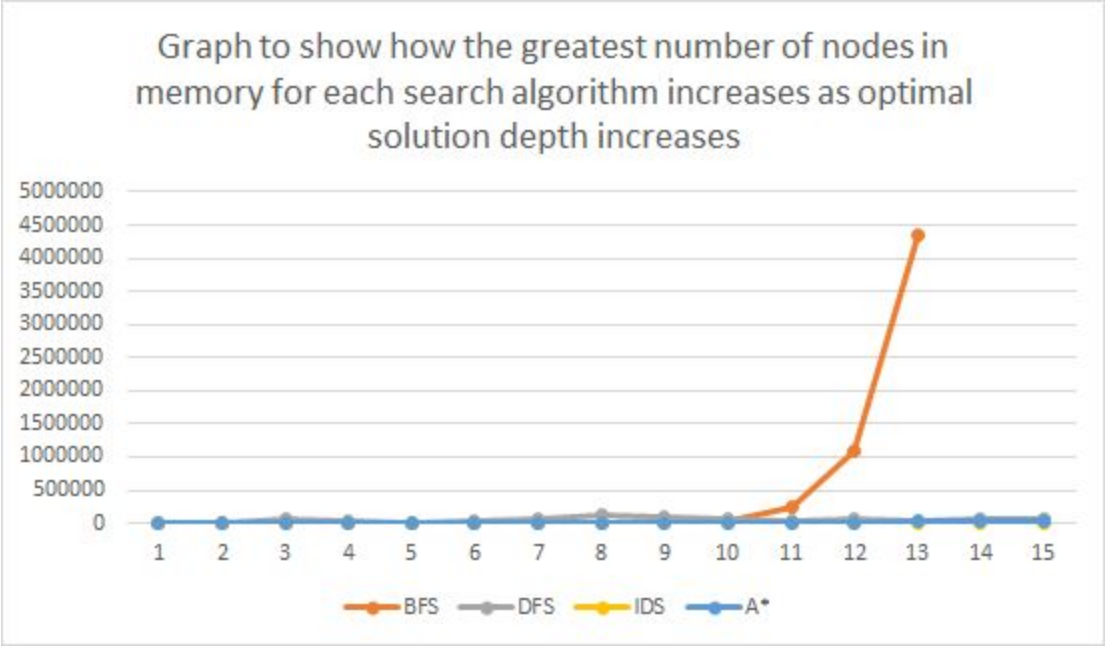
From this scalability study, we can conclude that blind search is incredibly inefficient and not very scalable in comparison to heuristic search.

Extras and limitations

Extras

The first extra challenge I implemented was allowing the Blocksworld to be a NxN grid with N-1 blocks in the puzzle. This could be used if you want to make the solution depth even further down a tree. However, IDS and A* search start to run out of memory past solution depth 15.

The second extra I made was a graph plotting space complexity- measured by the maximum number of nodes in memory- versus optimal solution depth.



The graph shows that BFS stores significantly more nodes in memory than any other search algorithm. This is because it essentially stores the entire search tree, and is why BFS runs out of memory before any other node does. DFS has the second biggest space complexity, mainly due to how far it goes down a branch of the search tree before finding a solution. A* search has the 3rd biggest space complexity, as although it does not expand many nodes, it still has to store all possible moves from a state, even if they are stored towards the back of the queue. IDS is the best algorithm in terms of space complexity; it only stores 38 nodes at most for a solution of depth 14.

Limitations

There were a number of weaknesses in the way I approached this problem. By storing the grid as a 2D array, each state took up an unnecessary amount of memory, which meant BFS ran out of memory earlier than it needed to. If I instead just stored the coordinates of the blocks and agent and created a grid from that, it would have scaled better. It would have also meant that my method to calculate Manhattan distance would have had a better time complexity, as currently it uses several nested loops to find the coordinates of points on two different states.

My DFS was also very inefficient since it never backtracked, meaning that it just took random moves until a solution was found. It also sometimes went in one direction then went back in the opposite direction .I could have implemented checks to ensure it never visited previously visited states, but then it would be a graph search algorithm and not a tree search algorithm

Code

DepthFirstSearch.java