Intelligent Systems Coursework  
**Blockworld- Search Methods**

horizontal line

# 

# Problem Solving via implementing various search methods

### The Problem

The Blockworld puzzle, is a grid configuration of a number of blocks, labelled by distinct letters, and an agent. To solve this puzzle, the agent must move the block around in such a way that they end up in a configuration which matches a given goal state. The agent moves blocks via a mechanic in which, if an agent moves onto a tile occupied by a block, the block moves to where the agent was, and the agent moves to where the block was.

### My approach

To solve this programmatically, you can implement a search method to take a starting board configuration, and simulate the moves the agent could take from that start state, and find when it reaches a configuration that matches the goal state.

In this assignment, four search methods were implemented to solve the puzzle, three uninformed search methods (depth-first search, breadth-first search, and iterative deepening search), and one informed heuristic search (I used A\*).

As my start state, I used the board configuration defined in the assignment spec, which I have copied to the top of this report. This allowed testing of the implementation to be simpler, however it would be relatively trivial to change the implementation of the program to allow for either randomly generated start states, or ones defined by the user at run time.

Having a different start state would alter the number of moves the agent would have to take to get the board to a goal state, and subsequently change the number of nodes expanded as well.

For my goal state, I initially defined it as any board state that has the ‘c’ block on the bottom row, the ‘b’ block on the cell above it etc. However, as I will discuss in detail later, this makes it much harder to write a heuristic for in my informed search, so I simplified the implementation to having the goal state as the example state defined in the coursework spec.

For depth-first search, I used an iterative implementation of the algorithm. As depth-first search can get stuck in an infinite loop in trees with an infinite depth, I had to modify it to avoid this. To achieve this, I kept a HashSet of visited board states, and whenever I visit a node, if it is in the HashSet, then I traverse back up a level, since as I have visited that board state before I know there’s no solutions in the tree spawning for its children. This reduces the nodes expanded and agent moves considerably, without repetition checking, there are around 10 times more agent moves on average. There is also optimal randomisation of the order for adding nodes to the fringe, which can let the algorithm sometimes find a more optimal solution.

For breadth-first search, I used a standard implementation of the algorithm, but included the checking for repeated board states, which would filter down the branching number of some nodes if they contain repeated children, reducing the number of nodes expanded.

For iterative deepening, I simply took my depth first search, added a max traversal depth parameter, and use the function in another function that iterates the max depth till a solution is found.

For my heuristic search, I implemented the A\* search algorithm, using a heuristic that sums the manhattan distance of the current position all the board pieces, to their destination positions in the goal state.

Problem difficulty is controlled by increasing the size of the board, from 4,4 to 5,5 , 6,6 etc. To fix control variables, I moved the starting position of the blocks to be relative to their original position when increasing the board size, so they are always resting on the bottom row.

### Results Analysis

From the graph above we can see the results of the search methods, as we scale problem difficulty by increasing the number of tile spaces in the board. I included both the unrandomised and randomised results for depth-first search as they give noticeably different results. I could only test randomised dfs up to 36 tiles as I ran out of computer memory on program runs above that.

There seems to be a few anomalies scattered about, most noticeably in the unrandomised depth first search, where after 36 tile spaces, the number of nodes expanded starts decreasing. I have spent quite a few hours trying to find/work out the source of these anomalies, but due to a lack of time, I will have to derive an analysis with these anomalies included.

Even if there were no anomalies, it is quite clear that depth-first search is the least optimal algorithm for this problem, the only one to go above 100,000 nodes expanded for an 8x8 board. It is the only one where the most optimal path is not always found, as evidenced by the table below

### Result Conclusion

I am thinking that the reason that the dfs performs so poorly for this, is that in this problem, there is such a large number of nodes and possible board states in the search space, it means that because dfs traverses depth first, it ends up wasting a lot of time going deep into the tree when the solution is actually much shallower. On the other hands, bfs and iterative deepening both explore the width of the tree much sooner than dfs does, and so finds the solution much sooner, heuristic search will always find the most optimal path as it is an informed search. In fact dfs only becomes feasible via pruning of board states already visited, because without it the dfs can into a loop of the local board space, even with randomisation, it would not be completed within an acceptable timeframe compared to the other search methods.

Looking at the other results, it looks like most of them have a time complexity in linear time, if not log linear time, the repeated node pruning may be a large factor in this, taking out the repetition checking would not affect heuristic search that much, as the algorithm always takes the lowest cost move towards the goal state, so is unlikely it would go to a board state it had already seen. With the other two algorithms though, taking out the pruning would mean there wouldn’t be any cutting down of the branching number of a node, which is a large factors for the run time of the search, it means that *b* would be always a number 2-4, whereas pruning cuts the average *b* down by a significant amount after the set of visited board states becomes a significant size. Considering that in the worst case in a bfs, the number of nodes expanded is ~416 (4x109) for a 4x4 board, this significantly increases search time, which would also contribute to why unpruned dfs is so infeasible (although the board space loop is still the main factor).

This analysis is dependent on my method of increasing problem difficulty being valid, there has not been enough time to test other methods of scaling it though, but I would hope that the algorithms are designed in such a way that if a different method were to be used for scaling it, the search methods performance would also scale consistently when comparing different methods.

### Limitations

As is evident in my code, this implementation of the blockworld puzzle is heavily limited, it has one start state, and one goal state. In theory, the puzzle could support any board configuration as a start state, and the number of goal states would be at least the number of columns, if not more if you let the position of the agent within a goal state matter. Allowing for this variation would considerably increase the complexity of the program, most noticeably the heuristic calculation for the informed search, and because of randomisation, each search would need to be run multiple time to compute an average, massively increasing the time needed to retrieve a set a of results.

Obviously the anomalies in the results are a major weakness of my work, and indicate there is either something wrong with my method of scaling problem difficulty, or there is some part of my code that is making the algorithms perform in a different manner to how they are meant to. Given more time I would try and narrow this down and fix it, so that the results can be analysed properly. With the current data, all that I can really gather is that dfs is polynomial/exponential, and the other three are linear/log linear.

Generally dfs should be O(bm), where m is the max depth of the tree, although as the max depth is dependent on all the children of a node having already been visited, it is hard to estimate a value for m, bfs should be O(bd) where d is the most optimal depth of the solution. Iterative deepening is equivalent to bfs but uses less memory, the complexity of A\* is harder to define, as it depends on whether the error of the heuristic of the distance to goal grows faster than the log of the true distance to goal. As my manhattan distance heuristic would somewhat decrease in reliability the further away the current state is from the board state, the complexity is somewhere between exponential and polynomial time most likely, although still more efficient than the other three algorithms, as it is informed, it will scale to large problems much better. Given more time, I would also add metrics to measure the space complexity of the search methods, as this would allow them to be compared more strongly.

### Results

**Nodes Expanded**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tile Space** | **Depth-First Search** | **Breadth-First Search** | **Iterative Deepening** | **Heuristic** | **DFS Randomised** |
| **16** | 43533 | 3,829 | 312 | 3,629 | 27741 |
| **25** | 90815 | 6,718 | 947 | 3,007 | 162902 |
| **36** | 284735 | 8,142 | 1,235 | 3,585 | 605686 |
| **49** | 250667 | 9,073 | 1,412 | 4,354 |  |
| **64** | 168092 | 9,708 | 1,532 | 4,609 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **16** | 20,883 | 16 | 16 | 16 |
| **25** | 118,908 | 16 | 16 | 16 |
| **36** | 397,432 | 16 | 16 | 16 |
| **49** | 248,592 | 16 | 16 | 16 |
| **64** | 167,392 | 16 | 16 | 16 |

**Agent Moves**

**Appendix**

Output of program to prove that algorithms work

Solution State:

----

-a--

-b--

-c-x

Nodes expanded: 43533, Agent moves for solution: 5250

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Solution State:

----

-a--

-b--

-c-x

Nodes expanded: 3829, Agent moves for solution: 16

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Solution State:

----

-a--

-b—-

-c-x

Nodes expanded: 312, Agent moves for solution: 16

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Solution State:

----

-a--

-b--

-c-x

Nodes expanded: 3629, Agent moves for solution: 16

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Code

package game**.**board**;**

**import** java**.**awt**.**Rectangle**;**

**import** java**.**util**.**ArrayList**;**

**import** game**.**Game**;**

**import** game**.**piece**.**Agent**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**PiecePosition**;**

**import** search**.**Node**;**

**import** utilities**.**Pair**;**

public class BoardState

**{**

private Rectangle board**;**

private Agent agent**;**

private ArrayList**<**Block**>** blocks**;**

/\*\*

\* @param board

\* @param pieces

\*/

public BoardState**(**Rectangle board**,** Agent agent**,** ArrayList**<**Block**>** blocks**)**

**{**

**super();**

**this.**board **=** board**;**

**this.**agent **=** agent**;**

**this.**blocks **=** blocks**;**

**}**

/\*\*

\* Checks each element of the board state for equality with each element of another board state

\*

\* @param boardState

\* The other board state to be checked for equality with this one

\* @return True if board states are equal, otherwise false

\*/

@Override

public boolean equals**(**Object obj**)**

**{**

boolean equalBoardState **=** **true;**

**if** **(!(this.**getBoard**().**equals**(((**BoardState**)** obj**).**getBoard**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getAgent**().**equals**(((**BoardState**)** obj**).**getAgent**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getBlocks**().**equals**(((**BoardState**)** obj**).**getBlocks**())))**

**{**

equalBoardState **=** **false;**

**}**

**return** equalBoardState**;**

**}**

/\*\*

\* Generates a graphical representation of the BoardState, where empty tiles are '-', blocks are dentoed by their

\* letter, and the agent is represented by an 'x'

\*

\* @return A graphical representation of the BoardState

\*/

@Override

public String toString**()**

**{**

PiecePosition drawHead **=** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>());**

StringBuilder boardImage **=** **new** StringBuilder**();**

boardImage**.**append**(**""**);**

char tileImage**;**

**for** **(**int x **=** 0**;** x **<** **this.**getBoard**().**getWidth**();** x**++)**

**{**

**for** **(**int y **=** 0**;** y **<** **this.**getBoard**().**getHeight**();** y**++)**

**{**

tileImage **=** '-'**;**

drawHead**.**setPosition**(new** Pair**<**Integer**,** Integer**>(**x**,** y**));**

**for** **(**Block b **:** **this.**getBlocks**())**

**{**

**if** **((**drawHead**.**x**().**equals**(**b**.**getPosition**().**x**()))** **&&** **(**drawHead**.**y**().**equals**(**b**.**getPosition**().**y**())))**

**{**

tileImage **=** b**.**getName**().**charAt**(**0**);**

**}**

**}**

**if** **((**drawHead**.**x**().**equals**(this.**getAgent**().**getPosition**().**x**()))** **&&** **(**drawHead**.**y**().**equals**(this.**getAgent**().**getPosition**().**y**())))**

**{**

tileImage **=** 'x'**;**

**}**

boardImage**.**append**(**tileImage**);**

**}**

boardImage**.**append**(**"\n"**);**

**}**

**return** boardImage**.**toString**();**

**}**

public Rectangle getBoard**()**

**{**

**return** board**;**

**}**

public Agent getAgent**()**

**{**

**return** agent**;**

**}**

public ArrayList**<**Block**>** getBlocks**()**

**{**

**return** blocks**;**

**}**

/\*\*

\* Generate all possible moves of the agent, where a possible move is one that does not take it out of the bounds of

\* the board

\* @param currentLevel

\*

\* @return An ArrayList of all possible agent moves

\*/

public ArrayList**<**Node**>** generatePossibleMoves**(**Node currentNode**,** int currentDepth**)**

**{**

ArrayList**<**Node**>** validMoves **=** **new** ArrayList**<**Node**>();**

BoardState boardStateCopy **=** **this.**copy**();**

PiecePosition initAgent **=** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**boardStateCopy**.**getAgent**().**getPosition**().**x**(),** boardStateCopy**.**getAgent**().**getPosition**().**y**()));**

PiecePosition boardStateProbe **=** boardStateCopy**.**getAgent**().**getPosition**();**

ArrayList**<**Pair**<**Integer**,** Integer**>>** moveVectors **=** **new** ArrayList**<**Pair**<**Integer**,** Integer**>>();**

moveVectors**.**add**(new** Pair**<**Integer**,** Integer**>(-**1**,** 0**));**

moveVectors**.**add**(new** Pair**<**Integer**,** Integer**>(**0**,** 1**));**

moveVectors**.**add**(new** Pair**<**Integer**,** Integer**>(**1**,** 0**));**

moveVectors**.**add**(new** Pair**<**Integer**,** Integer**>(**0**,** **-**1**));**

**for** **(**Pair**<**Integer**,** Integer**>** moveVector **:** moveVectors**)**

**{**

ArrayList**<**Block**>** blockProbe **=** **new** ArrayList**<**Block**>();**

**for** **(**Block b **:** boardStateCopy**.**getBlocks**())**

**{**

blockProbe**.**add**(**b**.**copy**());**

**}**

//Adjust the agent by each cardinal direction and check for validity

boardStateProbe**.**setPosition**(new** Pair**<**Integer**,** Integer**>(**boardStateProbe**.**x**()** **+** moveVector**.**getLeft**(),** boardStateProbe**.**y**()** **+** moveVector**.**getRight**()));**

**if** **(**isPieceInBounds**(**boardStateProbe**))**

**{**

**for** **(**Block b **:** blockProbe**)**

**{**

**if** **((**boardStateProbe**.**x**()** **==** b**.**getPosition**().**x**())** **&&** **(**boardStateProbe**.**y**()** **==** b**.**getPosition**().**y**()))**

**{**

b**.**getPosition**().**setX**(**initAgent**.**x**());**

b**.**getPosition**().**setY**(**initAgent**.**y**());**

**}**

**}**

Agent newAgent **=** boardStateCopy**.**getAgent**().**copy**();**

newAgent**.**getPosition**().**setPosition**(new** Pair**<**Integer**,** Integer**>(**boardStateProbe**.**x**(),** boardStateProbe**.**y**()));**

BoardState validMove **=** **new** BoardState**(**boardStateCopy**.**getBoard**(),** newAgent**,** blockProbe**);**

validMoves**.**add**(new** Node**(**validMove**,** currentNode**,** **new** ArrayList**<**Node**>(),** currentDepth **+** 1**));**

**}**

boardStateProbe**.**setPosition**(new** Pair**<**Integer**,** Integer**>(**initAgent**.**x**(),** initAgent**.**y**()));**

**}**

**return** validMoves**;**

**}**

/\*\*

\* Checks whether the given board piece is within the bounds of the grid of the board

\*

\* @param boardStateProbe

\* The piece of which it's position will be checked

\* @return True if piece is within bounds, otherwise false

\*/

private boolean isPieceInBounds**(**PiecePosition boardStateProbe**)**

**{**

**if** **((**boardStateProbe**.**x**()** **>=** 0**)** **&&** **(**boardStateProbe**.**x**()** **<** **this.**getBoard**().**getHeight**()))**

**{**

**if** **((**boardStateProbe**.**y**()** **>=** 0**)** **&&** **(**boardStateProbe**.**y**()** **<** **this.**getBoard**().**getWidth**()))**

**{**

**return** **true;**

**}**

**}**

**return** **false;**

**}**

/\*\*

\* Checks that the heaviest block(biggest letter in the alphabet where B > A, C > B etc.) is on the bottom row on

\* the grid, and that the lighter letters in the array of blocks, going sequentially backwards along the alphabet

\* from the heaviest letter to A, are on top of the heavier letter, and that they are all in the same column

\*

\* @return true if this BoardState is a goal state, otherwise return false

\*/

public boolean isGoalState**()**

**{**

**if** **(this.**equals**(**Game**.**goalState**))**

**return** **true;**

**else**

**return** **false;**

**}**

/\*\*

\* @return A deep copy of this boardstate

\*/

public BoardState copy**()**

**{**

**return** **new** BoardState**(new** Rectangle**(this.**getBoard**()),** **this.**getAgent**().**copy**(),** **new** ArrayList**<**Block**>(this.**getBlocks**()));**

**}**

@Override

public int hashCode**()**

**{**

int hash **=** 5**;**

hash **=** **(**int**)** **(**89 **\*** hash **+** **((this.**getBoard**().**getWidth**())** **+** **(this.**getBoard**().**getHeight**())));**

hash **=** 89 **\*** hash **+** **((this.**getAgent**().**getPosition**().**x**())** **+** **(this.**getAgent**().**getPosition**().**y**()));**

**for** **(**Block b **:** **this.**getBlocks**())**

**{**

hash **=** **(**89 **\*** hash **+** **(**b**.**getPosition**().**x**())** **+** **(**b**.**getPosition**().**y**()));**

**}**

**return** hash**;**

**}**

**}**

/\*\*

\*

\*/

package game**;**

**import** java**.**awt**.**Rectangle**;**

**import** java**.**io**.**BufferedReader**;**

**import** java**.**io**.**IOException**;**

**import** java**.**io**.**InputStreamReader**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Stack**;**

**import** game**.**board**.**BoardState**;**

**import** game**.**piece**.**Agent**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**PiecePosition**;**

**import** game**.**piece**.**PieceType**;**

**import** search**.**BreadthFirstSearch**;**

**import** search**.**DepthFirstSearch**;**

**import** search**.**Heuristic**;**

**import** search**.**IterativeDeepeningSearch**;**

**import** search**.**NoSolutionPossibleException**;**

**import** search**.**SearchType**;**

**import** utilities**.**Pair**;**

/\*\*

\*

\*/

public class Game

**{**

private BoardState startState**;**

public static BoardState goalState**;**

/\*\*

\* Creates a start state and a goal state for the game

\*/

public Game**()**

**{**

Rectangle startBoard **=** **new** Rectangle**(**4**,** 4**);**

ArrayList**<**Block**>** pieces **=** **new** ArrayList**<**Block**>();**

pieces**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 0**))));**

pieces**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**))));**

pieces**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 2**))));**

Agent agent **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 3**)));**

startState **=** **new** BoardState**(**startBoard**,** agent**,** pieces**);**

Rectangle goalBoard **=** **new** Rectangle**(**4**,** 4**);**

ArrayList**<**Block**>** piecesG **=** **new** ArrayList**<**Block**>();**

piecesG**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,** 1**))));**

piecesG**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,** 1**))));**

piecesG**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**))));**

Agent agentG **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 3**)));**

goalState **=** **new** BoardState**(**goalBoard**,** agentG**,** piecesG**);**

**}**

/\*\*

\* Menu system for running the game, after a search has been run, adds the results to it's relative accumulator, and

\* if the csv toggle has been selected, will print out the average result for each search time when all search runs

\* have been finished

\*

\* @param args

\*/

public static void main**(**String**[]** args**)**

**{**

Game game **=** **new** Game**();**

**try**

**{**

BufferedReader br **=** **new** BufferedReader**(new** InputStreamReader**(**System**.**in**));**

System**.**out**.**println**(**"Selection no. of iterations"**);**

int iterations **=** Integer**.**parseInt**(**br**.**readLine**());**

System**.**out**.**println**(**"Select search type: '1. DFS 2. BFS 3. Iterative Deepening 4. Heuristic"**);**

int searchTypeIndex **=** Integer**.**parseInt**(**br**.**readLine**());**

System**.**out**.**println**(**"CSV? (y/n)"**);**

boolean csvFile **=** br**.**readLine**().**equals**(**"y"**)** **?** **true** **:** **false;**

GameResult dfsGameResult **=** **null,** bfsGameResult **=** **null,** idGameResult **=** **null,** heurGameResult **=** **null;**

SearchType searchType **=** SearchType**.**values**()[**searchTypeIndex **-** 1**];**

int dfsNodesAccumulator **=** 0**,** bfsNodesAccumulator **=** 0**,** idNodesAccumulator **=** 0**,** heurNodesAccumulator **=** 0**,** dfsAgentAccumulator **=** 0**,** bfsAgentAccumulator **=** 0**,**

idAgentAccumulator **=** 0**,** heurAgentAccumulator **=** 0**,** i**;**

**switch** **(**searchType**)**

**{**

**case** DEPTH\_FIRST**:**

**for** **(**i **=** 0**;** i **<** iterations**;** i**++)**

**{**

dfsGameResult **=** DepthFirstSearch**.**depthFirst**(**game**.**getBoardstate**());**

dfsNodesAccumulator **+=** dfsGameResult**.**getNodesExpanded**();**

dfsAgentAccumulator **+=** dfsGameResult**.**getAgentMoves**();**

printResult**(**dfsGameResult**);**

**}**

**break;**

**case** BREADTH\_FIRST**:**

**for** **(**i **=** 0**;** i **<** iterations**;** i**++)**

**{**

bfsGameResult **=** BreadthFirstSearch**.**breadthFirst**(**game**.**getBoardstate**());**

bfsNodesAccumulator **+=** bfsGameResult**.**getNodesExpanded**();**

bfsAgentAccumulator **+=** bfsGameResult**.**getAgentMoves**();**

printResult**(**bfsGameResult**);**

**}**

**break;**

**case** ITERATIVE\_DEEPENING**:**

**for** **(**i **=** 0**;** i **<** iterations**;** i**++)**

**{**

idGameResult **=** IterativeDeepeningSearch**.**iterativeDeepening**(**game**.**getBoardstate**());**

idNodesAccumulator **+=** idGameResult**.**getNodesExpanded**();**

idAgentAccumulator **+=** idGameResult**.**getAgentMoves**();**

printResult**(**idGameResult**);**

**}**

**break;**

**case** A\_STAR**:**

**for** **(**i **=** 0**;** i **<** iterations**;** i**++)**

**{**

heurGameResult **=** Heuristic**.**heuristic**(**game**.**getBoardstate**());**

heurNodesAccumulator **+=** heurGameResult**.**getNodesExpanded**();**

heurAgentAccumulator **+=** heurGameResult**.**getAgentMoves**();**

printResult**(**heurGameResult**);**

**}**

**break;**

**default:**

**break;**

**}**

**if** **(**csvFile**)**

**{**

System**.**out**.**println**(((**int**)** game**.**getBoardstate**().**getBoard**().**getWidth**())** **+** "," **+** **((**int**)** game**.**getBoardstate**().**getBoard**().**getHeight**())** **+** ","

**+** dfsNodesAccumulator **/** iterations **+** "," **+** dfsAgentAccumulator **/** iterations**);**

System**.**out**.**println**(((**int**)** game**.**getBoardstate**().**getBoard**().**getWidth**())** **+** "," **+** **((**int**)** game**.**getBoardstate**().**getBoard**().**getHeight**())** **+** ","

**+** bfsNodesAccumulator **/** iterations **+** "," **+** bfsAgentAccumulator **/** iterations**);**

System**.**out**.**println**(((**int**)** game**.**getBoardstate**().**getBoard**().**getWidth**())** **+** "," **+** **((**int**)** game**.**getBoardstate**().**getBoard**().**getHeight**())** **+** ","

**+** idNodesAccumulator **/** iterations **+** "," **+** idAgentAccumulator **/** iterations**);**

System**.**out**.**println**(((**int**)** game**.**getBoardstate**().**getBoard**().**getWidth**())** **+** "," **+** **((**int**)** game**.**getBoardstate**().**getBoard**().**getHeight**())** **+** ","

**+** heurNodesAccumulator **/** iterations **+** "," **+** heurAgentAccumulator **/** iterations**);**

**}**

**}**

**catch** **(**IOException e**)**

**{**

e**.**printStackTrace**();**

**}**

**}**

/\*\*

\*

\*/

private static void printResult**(**GameResult gameResult**)**

**{**

BoardState finalState **=** gameResult**.**getSolutionPath**().**elementAt**(**0**);**

System**.**out**.**println**(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"**);**

System**.**out**.**println**(**"Solution State: "**);**

System**.**out**.**println**(**finalState**);**

System**.**out**.**println**(**" "**);**

System**.**out**.**println**(**"Nodes expanded: " **+** gameResult**.**getNodesExpanded**()** **+** ", Agent moves for solution: " **+** gameResult**.**getAgentMoves**());**

System**.**out**.**println**(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"**);**

**}**

/\*\*

\* @param solutionPath

\* The stack of board states that lead to the solution of the game

\*/

public void printSolutionPath**(**Stack**<**BoardState**>** solutionPath**)**

**{**

**while** **(!(**solutionPath**.**isEmpty**()))**

**{**

System**.**out**.**println**(**solutionPath**.**pop**());**

**}**

**}**

public BoardState getBoardstate**()**

**{**

**return** startState**;**

**}**

**}**

package game**;**

**import** java**.**util**.**Stack**;**

**import** game**.**board**.**BoardState**;**

public class GameResult

**{**

private boolean solutionFound**;**

private Stack**<**BoardState**>** solutionPath**;**

private Integer nodesExpanded**,** agentMoves**;**

/\*\*

\* An object containing the board states that lead to the goal state of a of a run of the board game, the number of

\* nodes expanded to calculate it, and the number of moves the agent needs to take to reach that goal state

\*

\* @param solutionPath

\* Stack of board states that the agent moves to reach goal state

\* @param nodesExpanded

\* The number of nodes expanded to calculate solution

\* @param agentMoves

\* The number of moves the agent has to take to reach the goal state from the start state

\*/

public GameResult**(**boolean solutionFound**,** Stack**<**BoardState**>** solutionPath**,** Integer nodesExpanded**,** Integer agentMoves**)**

**{**

**super();**

**this.**solutionFound **=** solutionFound**;**

**this.**solutionPath **=** solutionPath**;**

**this.**nodesExpanded **=** nodesExpanded**;**

**this.**agentMoves **=** agentMoves**;**

**}**

public boolean isSolutionFound**()**

**{**

**return** solutionFound**;**

**}**

public Stack**<**BoardState**>** getSolutionPath**()**

**{**

**return** solutionPath**;**

**}**

public Integer getNodesExpanded**()**

**{**

**return** nodesExpanded**;**

**}**

public Integer getAgentMoves**()**

**{**

**return** agentMoves**;**

**}**

**}**

/\*\*

\*

\*/

package game**.**piece**;**

**import** utilities**.**Pair**;**

public class Agent **extends** BoardPiece

**{**

/\*\*

\* @param name

\* @param type

\* @param position

\*/

public Agent**(**String name**,** PieceType type**,** PiecePosition position**)**

**{**

**super(**name**,** type**,** position**);**

// TODO Auto-generated constructor stub

**}**

public Agent copy**()**

**{**

**return** **new** Agent**(super.**getName**(),** **super.**getType**(),** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(super.**getPosition**().**x**(),** **super.**getPosition**().**y**())));**

**}**

public String toString**()**

**{**

**return** **new** String**(super.**getName**()** **+** "," **+** **super.**getType**().**toString**()** **+** "," **+** **super.**getPosition**().**toString**());**

**}**

@Override

public boolean equals**(**Object obj**)**

**{**

boolean equalBoardState **=** **true;**

**if** **(!(this.**getName**().**equals**(((**Agent**)** obj**).**getName**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getType**().**equals**(((**Agent**)** obj**).**getType**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getPosition**().**equals**(((**Agent**)** obj**).**getPosition**())))**

**{**

equalBoardState **=** **false;**

**}**

**return** equalBoardState**;**

**}**

@Override

public int hashCode**()**

**{**

int hash **=** 7**;**

hash **=** 97 **\*** hash **+** **(this.**getName**().**hashCode**());**

hash **=** 97 **\*** hash **+** **(this.**getType**().**hashCode**());**

hash **=** **(**97 **\*** hash **+** **(this.**getPosition**().**x**())** **+** **(this.**getPosition**().**y**()));**

**return** hash**;**

**}**

**}**

package game**.**piece**;**

public enum AgentMove

**{**

UP**,** RIGHT**,** DOWN**,** LEFT

**}**

package game**.**piece**;**

**import** utilities**.**Pair**;**

public class Block **extends** BoardPiece

**{**

public Block**(**String name**,** PieceType type**,** PiecePosition position**)**

**{**

**super(**name**,** type**,** position**);**

// TODO Auto-generated constructor stub

**}**

@Override

public String toString**()**

**{**

**return** **new** String**(super.**getName**()** **+** "," **+** **super.**getType**().**toString**()** **+** "," **+** **super.**getPosition**().**toString**());**

**}**

public Block copy**()**

**{**

**return** **new** Block**(super.**getName**(),** **super.**getType**(),** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(super.**getPosition**().**x**(),** **super.**getPosition**().**y**())));**

**}**

@Override

public boolean equals**(**Object obj**)**

**{**

boolean equalBoardState **=** **true;**

**if** **(!(this.**getName**().**equals**(((**Block**)** obj**).**getName**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getType**().**equals**(((**Block**)** obj**).**getType**())))**

**{**

equalBoardState **=** **false;**

**}**

**if** **(!(this.**getPosition**().**equals**(((**Block**)** obj**).**getPosition**())))**

**{**

equalBoardState **=** **false;**

**}**

**return** equalBoardState**;**

**}**

@Override

public int hashCode**()**

**{**

int hash **=** 11**;**

hash **=** 101 **\*** hash **+** **(this.**getName**().**hashCode**());**

hash **=** 101 **\*** hash **+** **(this.**getType**().**hashCode**());**

hash **=** **(**101 **\*** hash **+** **(this.**getPosition**().**x**())** **+** **(this.**getPosition**().**y**()));**

**return** hash**;**

**}**

**}**

package game**.**piece**;**

public class BoardPiece

**{**

private String name**;**

private PieceType type**;**

private PiecePosition position**;**

/\*\*

\* @param name

\* @param type

\* @param position

\*/

public BoardPiece**(**String name**,** PieceType type**,** PiecePosition position**)**

**{**

**super();**

**this.**name **=** name**;**

**this.**type **=** type**;**

**this.**position **=** position**;**

**}**

public String getName**()**

**{**

**return** name**;**

**}**

public PieceType getType**()**

**{**

**return** type**;**

**}**

public PiecePosition getPosition**()**

**{**

**return** position**;**

**}**

**}**

package game**.**piece**;**

**import** utilities**.**Pair**;**

public class PiecePosition

**{**

private Pair**<**Integer**,** Integer**>** position**;**

/\*\*

\* @param position

\*/

public PiecePosition**(**Pair**<**Integer**,** Integer**>** position**)**

**{**

**super();**

**this.**position **=** position**;**

**}**

public Integer x**()**

**{**

**return** position**.**getLeft**();**

**}**

public Integer y**()**

**{**

**return** position**.**getRight**();**

**}**

public void setX(Integer x)

{

position.setLeft(x);

}

public void setY(Integer y)

{

position.setRight(y);

}

public void setPosition(Pair<Integer, Integer> position)

{

this.position = position;

}

@Override

public boolean equals(Object obj)

{

boolean equalBoardState = true;

if ((!(this.x().equals(((PiecePosition) obj).x()))) || ((!(this.y().equals(((PiecePosition) obj).y())))))

{

equalBoardState = false;

}

return equalBoardState;

}

@Override

public String toString()

{

return new String(this.x() + "," + this.y());

}

@Override

public int hashCode()

{

int hash = 13;

hash = (103 \* hash + (this.x()) + (this.y()));

return hash;

}

}

package game**.**piece**;**

public enum PieceType

**{**

AGENT**,** BLOCK

**}**

/\*\*

\*

\*/

package search**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**HashSet**;**

**import** java**.**util**.**LinkedList**;**

**import** java**.**util**.**Stack**;**

**import** game**.**GameResult**;**

**import** game**.**board**.**BoardState**;**

/\*\*

\* @author Will

\*

\*/

public class BreadthFirstSearch **extends** Search

**{**

/\*\*

\* Performs a breadth first search to find a path from the given initial state, to the goal state of the board. It

\* explores each depth level fully before going down a level

\*

\* @param initBoardState

\* @return An object containing the solution path, number of nodes expanded, and number of moves to reach solution

\*/

public static GameResult breadthFirst**(**BoardState initBoardState**)**

**{**

int currentLevel **=** 0**;**

Node currentNode **=** **new** Node**(**initBoardState**,** **null,** **new** ArrayList**<**Node**>(),** currentLevel**);**

HashSet**<**BoardState**>** visitedBoardStates **=** **new** HashSet**<**BoardState**>();**

LinkedList**<**Node**>** fringe **=** **new** LinkedList**<**Node**>();**

int nodeCounter **=** 0**;**

boolean goalStateFound **=** **false;**

fringe**.**add**(**currentNode**);**

BoardState currentBoardState**;**

**while** **((!(**goalStateFound**)** **||** **!(**fringe**.**isEmpty**())))**

**{**

currentNode **=** fringe**.**remove**();**

currentBoardState **=** currentNode**.**getVal**();**

currentLevel **=** currentNode**.**getDepth**();**

// System.out.println(currentBoardState);

// System.out.println(nodeCounter);

**if** **(**currentBoardState**.**isGoalState**())**

**{**

goalStateFound **=** **true;**

Stack**<**BoardState**>** solutionPath **=** calculateSolutionPath**(**currentNode**);**

**return** **new** GameResult**(true,** solutionPath**,** visitedBoardStates**.**size**(),** solutionPath**.**size**());**

**}**

**else**

**{**

**if** **(**visitedBoardStates**.**add**(**currentBoardState**))**

**{**

nodeCounter**++;**

**}**

ArrayList**<**Node**>** possibleMoves **=** currentBoardState**.**generatePossibleMoves**(**currentNode**,** currentLevel**);**

//ArrayList<Tree<BoardState>> nextMoves = convertMovesToChildTrees(searchTree, possibleMoves, visitedBoardStates);

ArrayList**<**Node**>** nextMoves **=** filterOutSeenStates**(**currentNode**,** possibleMoves**,** visitedBoardStates**,** fringe**,** currentLevel**);**

fringe**.**addAll**(**nextMoves**);**

**}**

**}**

**return** **new** GameResult**(false,** **null,** nodeCounter**,** 0**);**

**}**

private static ArrayList**<**Node**>** filterOutSeenStates**(**Node currentNode**,** ArrayList**<**Node**>** possibleMoves**,** HashSet**<**BoardState**>** visitedBoardStates**,** LinkedList**<**Node**>** fringe**,**

int currentLevel**)**

**{**

ArrayList**<**Node**>** nextMoves **=** currentNode**.**getChildren**();**

**for** **(**Node bState **:** possibleMoves**)**

**{**

**if** **(!(**visitedBoardStates**.**contains**(**bState**.**getVal**())))** //If one of the possible moves has already been visited, then do not add it to nextMoves

**{**

boolean nodeInFringe **=** **false;**

**for** **(**Node fringeNode **:** fringe**)**

**{**

**if** **(**fringeNode**.**getVal**().**equals**(**bState**.**getVal**()))**

**{**

nodeInFringe **=** **true;**

**}**

**}**

**if** **(!(**nodeInFringe**))**

**{**

nextMoves**.**add**(new** Node**(**bState**.**getVal**(),** currentNode**,** **new** ArrayList**<**Node**>(),** currentLevel **+** 1**));**

**}**

**}**

**}**

**return** nextMoves**;**

**}**

// private static ArrayList<Node> convertMovesToChildTrees(Node currentNode, ArrayList<BoardState> possibleMoves, HashSet<BoardState> visitedBoardStates)

// {

// ArrayList<Node> nextMoves = currentNode.getChildren();

// for (BoardState bState : possibleMoves)

// {

// nextMoves.add(new Node(bState, currentNode, new ArrayList<Node>()));

// }

// return nextMoves;

// }

**}**

/\*\*

\*

\*/

package search**;**

**import** java**.**util**.**ArrayDeque**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Deque**;**

**import** java**.**util**.**HashSet**;**

**import** java**.**util**.**Random**;**

**import** java**.**util**.**Stack**;**

**import** game**.**GameResult**;**

**import** game**.**board**.**BoardState**;**

/\*\*

\* @author Will

\*

\*/

public class DepthFirstSearch **extends** Search

**{**

/\*\*

\* Performs a depth first search on the given initial board state to calculate the moves the agent can perform to

\* make the board reach a goal state

\*

\* @param initBoardState

\* The start state of the game

\* @return An object containing the solution path, number of nodes expanded, and number of moves to reach solution

\* @throws NoSolutionPossibleException

\* If a goal state can't be reached from the start state

\*/

public static GameResult depthFirst**(**BoardState initBoardState**)**

**{**

int currentLevel **=** 0**;**

Node currentNode **=** **new** Node**(**initBoardState**,** **null,** **new** ArrayList**<**Node**>(),** currentLevel**);**

HashSet**<**BoardState**>** visitedBoardStates **=** **new** HashSet**<**BoardState**>();**

Deque**<**Node**>** fringe **=** **new** ArrayDeque**<**Node**>();**

int nodeCounter **=** 0**;**

boolean goalStateFound **=** **false;**

fringe**.**addFirst**(**currentNode**);**

BoardState currentBoardState**;**

**while** **((!(**goalStateFound**)** **||** **!(**fringe**.**isEmpty**())))**

**{**

currentNode **=** fringe**.**removeFirst**();**

currentBoardState **=** currentNode**.**getVal**();**

currentLevel **=** currentNode**.**getDepth**();**

**if** **(**currentBoardState**.**isGoalState**())**

**{**

goalStateFound **=** **true;**

Stack**<**BoardState**>** solutionPath **=** calculateSolutionPath**(**currentNode**);**

**return** **new** GameResult**(true,** solutionPath**,** visitedBoardStates**.**size**(),** solutionPath**.**size**());**

**}**

**else**

**{**

**if** **(**visitedBoardStates**.**add**(**currentBoardState**))**

**{**

nodeCounter**++;**

**}**

ArrayList**<**Node**>** possibleMoves **=** currentBoardState**.**generatePossibleMoves**(**currentNode**,** currentLevel**);**

ArrayList**<**Node**>** nextMoves **=** filterOutSeenStates**(**currentNode**,** possibleMoves**,** visitedBoardStates**,** currentLevel**);**

**while** **(!(**nextMoves**.**isEmpty**()))**

**{**

fringe**.**addFirst**(**nextMoves**.**remove**(**0**));**

**}**

**}**

**}**

**return** **new** GameResult**(false,** **null,** nodeCounter**,** 0**);**

**}**

private static ArrayList**<**Node**>** filterOutSeenStates**(**Node currentNode**,** ArrayList**<**Node**>** possibleMoves**,** HashSet**<**BoardState**>** visitedBoardStates**,** int currentLevel**)**

**{**

ArrayList**<**Node**>** nextMoves **=** currentNode**.**getChildren**();**

nextMoves**.**clear**();**

**for** **(**Node possibleMove **:** possibleMoves**)**

**{**

BoardState succesorBState **=** possibleMove**.**getVal**();**

**if** **(!(**visitedBoardStates**.**contains**(**succesorBState**)))** //If one of the possible moves has already been visited, then do not add it to nextMoves

**{**

nextMoves**.**add**(new** Node**(**succesorBState**,** currentNode**,** **new** ArrayList**<**Node**>(),** currentLevel **+** 1**));**

**}**

**}**

**return** nextMoves**;**

**}**

**}**

package search**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Comparator**;**

**import** java**.**util**.**HashMap**;**

**import** java**.**util**.**HashSet**;**

**import** java**.**util**.**PriorityQueue**;**

**import** java**.**util**.**Stack**;**

**import** game**.**Game**;**

**import** game**.**GameResult**;**

**import** game**.**board**.**BoardState**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**BoardPiece**;**

public class Heuristic

**{**

private static HashMap**<**Node**,** Integer**>** gScore**;**

private static HashMap**<**Node**,** Integer**>** fScore**;**

private static Integer gScoreDefault **=** Integer**.**MAX\_VALUE**;**

private static Integer fScoreDefault **=** Integer**.**MAX\_VALUE**;**

/\*\*

\* Finds a solution to the initial board start by applying a heuristic to each neighbour of the node being explored

\* (h(n) = g(n) + f(n)) where g(n) is the cost from the start board state to the current board state, and f(n) is

\* the estimated cost of getting from the current board state to a goal state. Once this heuristic has been

\* calculated for each neighbour, they are all added the the fringe. The search decides the next node from the

\* fringe to explore by picking the one with the lowest value for its's heuristic

\*

\* @param initBoardState

\* The initial state of the board, and the root node of the search

\* @return A GameResult object containing whether the solution is found, the solution path, the number of nodes

\* expanded, and the number of moves the agent took to find the solution

\*/

public static GameResult heuristic**(**BoardState initBoardState**)**

**{**

int currentDepth **=** 0**;**

EstimatedCostComparator comparator **=** **new** Heuristic**.**EstimatedCostComparator**();**

HashSet**<**Node**>** visitedBoardStates **=** **new** HashSet**<**Node**>();** //Already seen nodes

PriorityQueue**<**Node**>** fringe **=** **new** PriorityQueue**<**Node**>(**comparator**);** //The potential children to be explored

Node rootNode **=** **new** Node**(**initBoardState**,** **null,** **new** ArrayList**<**Node**>(),** 0**);**

fringe**.**add**(**rootNode**);**// The The set of children to be explored, starting with the root node

gScore **=** **new** HashMap**<**Node**,** Integer**>();**

gScore**.**put**(**rootNode**,** 0**);** // Cost from start along best path that is known

fScore **=** **new** HashMap**<**Node**,** Integer**>();**

fScore**.**put**(**rootNode**,** gScore**.**get**(**rootNode**)** **+** estimateCostToGoal**(**rootNode**));** //Estimated cost from the root node to a goal state

**while** **(!(**fringe**.**isEmpty**()))**

**{**

Node currentNode **=** fringe**.**peek**();**

currentDepth **=** currentNode**.**getDepth**();**

**if** **(**currentNode**.**getVal**().**isGoalState**())**

**{**

Stack**<**BoardState**>** solutionPath **=** Search**.**calculateSolutionPath**(**currentNode**);**

**return** **new** GameResult**(true,** solutionPath**,** visitedBoardStates**.**size**(),** solutionPath**.**size**());**

**}**

currentNode **=** fringe**.**remove**();**

visitedBoardStates**.**add**(**currentNode**);**

ArrayList**<**Node**>** possibleMoves **=** currentNode**.**getVal**().**generatePossibleMoves**(**currentNode**,** currentDepth**);**

currentNode**.**setChildren**(new** ArrayList**<**Node**>(**possibleMoves**));**

**for** **(**Node childNode **:** currentNode**.**getChildren**())**

**{**

**if** **(**visitedBoardStates**.**contains**(**childNode**))**

**{**

**break;** // Ignore neighbours which are already explored

**}**

Integer tentativeGScore **=** **(**gScore**.**get**(**currentNode**)** **==** **null** **?** gScoreDefault **:** gScore**.**get**(**currentNode**))** **+** 1**;**// Length of the path from the root node to the current node

**if** **(!(**fringe**.**contains**(**childNode**)))**

**{**

fringe**.**add**(**childNode**);** // Discover a new node

**}**

gScore**.**put**(**childNode**,** tentativeGScore**);**

fScore**.**put**(**childNode**,** gScore**.**get**(**childNode**)** **+** estimateCostToGoal**(**childNode**));**

**}**

**}**

**return** **new** GameResult**(false,** **null,** 0**,** 0**);**

**}**

/\*\*

\* Uses a heuristic function (sum of manhattan distances) to generate an estimated cost from the current state to

\* the goal state

\*

\* @param rootNode

\* @return

\*/

private static Integer estimateCostToGoal**(**Node rootNode**)**

**{**

int cost **=** 0**;**

ArrayList**<**Block**>** currentBlockConfig **=** rootNode**.**getVal**().**getBlocks**();**

ArrayList**<**Block**>** targetBlockConfig **=** Game**.**goalState**.**getBlocks**();**

**for** **(**int i **=** 0**;** i **<** currentBlockConfig**.**size**();** i**++)**

**{**

cost **+=** manhattanDistance**(**currentBlockConfig**.**get**(**i**),** targetBlockConfig**.**get**(**i**));**

**}**

**return** cost**;**

**}**

/\*\*

\* Calculates the difference between two positions into x and y components, then adds the x and y components to get

\* the manhattan distance between the two position

\*

\* @param currentPiece

\* @param targetPiece

\* @return The manhattan distance between the two parameters

\*/

private static int manhattanDistance**(**BoardPiece currentPiece**,** BoardPiece targetPiece**)**

**{**

int xDistance **=** Math**.**abs**(**currentPiece**.**getPosition**().**x**()** **-** targetPiece**.**getPosition**().**x**());**

int yDistance **=** Math**.**abs**(**currentPiece**.**getPosition**().**y**()** **-** targetPiece**.**getPosition**().**y**());**

**return** xDistance **+** yDistance**;**

**}**

/\*\*

\* A comparator for the fringe so that it orders them where the node with the lowest fScore is at the head of the queue

\*

\*/

static class EstimatedCostComparator **implements** Comparator**<**Node**>**

**{**

@Override

public int compare**(**Node o1**,** Node o2**)**

**{**

Integer o1Cost **=** fScore**.**get**(**o1**)** **==** **null** **?** fScoreDefault **:** fScore**.**get**(**o1**);**

Integer o2Cost **=** fScore**.**get**(**o2**)** **==** **null** **?** fScoreDefault **:** fScore**.**get**(**o2**);**

**return** o1Cost **-** o2Cost**;**

**}**

**}**

**}**

/\*\*

\*

\*/

package search**;**

**import** java**.**util**.**ArrayDeque**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Deque**;**

**import** java**.**util**.**HashSet**;**

**import** java**.**util**.**Iterator**;**

**import** java**.**util**.**Random**;**

**import** java**.**util**.**Stack**;**

**import** game**.**GameResult**;**

**import** game**.**board**.**BoardState**;**

public class IterativeDeepeningSearch **extends** Search

**{**

/\*\*

\* Performs a search that keeps calling a depth first search, iterating the max depth it can search to, until a

\* solution is found

\*

\* @param initBoardState

\* The start state of the game

\* @return An object containing the solution path, number of nodes expanded, and number of moves to reach solution

\*/

public static GameResult iterativeDeepening**(**BoardState initBoardState**)**

**{**

int maxDepth **=** 0**;**

boolean goalStateFound **=** **false;**

Node rootNode **=** **new** Node**(**initBoardState**,** **null,** **new** ArrayList**<**Node**>(),** 0**);**

**while** **(!(**goalStateFound**))**

**{**

GameResult searchResult **=** altDFS**(**rootNode**,** maxDepth**);**

**if** **(**searchResult**.**isSolutionFound**())**

**{**

**return** searchResult**;**

**}**

maxDepth**++;**

**}**

**return** **new** GameResult**(false,** **null,** **null,** **null);**

**}**

/\*\*

\* A depth first search with a max depth parameter, stopping the search when all the nodes at a given max depth have

\* been explored

\*

\* @param node

\* The start node

\* @param maxDepth

\* The maximum depth to search to

\* @return A GameResult object, which contains a parameter on whether the search has found a solution or not

\*/

private static GameResult altDFS**(**Node node**,** int maxDepth**)**

**{**

int currentDepth **=** **-**1**;**

Node currentNode **=** node**;**

HashSet**<**Node**>** visitedBoardStates **=** **new** HashSet**<**Node**>();**

Deque**<**Node**>** fringe **=** **new** ArrayDeque**<**Node**>();**

int nodeCounter **=** 0**;**

fringe**.**addFirst**(**currentNode**);**

BoardState currentBoardState**;**

**while** **(!(**fringe**.**isEmpty**()))**

**{**

currentNode **=** fringe**.**removeFirst**();**

currentBoardState **=** currentNode**.**getVal**();**

currentDepth **=** currentNode**.**getDepth**();**

**if** **(**currentBoardState**.**isGoalState**())**

**{**

Stack**<**BoardState**>** solutionPath **=** calculateSolutionPath**(**currentNode**);**

**return** **new** GameResult**(true,** solutionPath**,** visitedBoardStates**.**size**(),** solutionPath**.**size**());**

**}**

**else** **if** **(**currentDepth **<** maxDepth**)**

**{**

**if** **(**visitedBoardStates**.**add**(**currentNode**))**

**{**

nodeCounter**++;**

**}**

ArrayList**<**Node**>** possibleMoves **=** currentBoardState**.**generatePossibleMoves**(**currentNode**,** currentDepth**);**

currentNode**.**setChildren**(**possibleMoves**);**

ArrayList**<**Node**>** nextMoves **=** filterOutSeenStates**(**currentNode**,** possibleMoves**,** visitedBoardStates**,** currentDepth**);**

**for** **(**Node move **:** nextMoves**)**

**{**

fringe**.**addFirst**(**move**);**

**}**

**}**

**}**

**return** **new** GameResult**(false,** **null,** nodeCounter**,** 0**);**

**}**

/\*\*

\* Iterates through the list of possible moves from the current node, checking for each one if it is in the set of

\* visited board states, if it is, check if the depth of it is lower than the depth of the node when it was last

\* seen

\*

\* @param currentNode

\* @param possibleMoves

\* @param visitedNodes

\* @param currentDepth

\* @return

\*/

private static ArrayList**<**Node**>** filterOutSeenStates**(**Node currentNode**,** ArrayList**<**Node**>** possibleMoves**,** HashSet**<**Node**>** visitedNodes**,** int currentDepth**)**

**{**

ArrayList**<**Node**>** nextMoves **=** **new** ArrayList**<**Node**>(**currentNode**.**getChildren**());**

nextMoves**.**clear**();**

**for** **(**Node possibleMove **:** possibleMoves**)**

**{**

boolean shouldAddToFringe **=** **true;**

BoardState succesorBState **=** possibleMove**.**getVal**();**

Iterator**<**Node**>** visitedNodesIterator **=** visitedNodes**.**iterator**();**

**while** **(**visitedNodesIterator**.**hasNext**())**

**{**

Node nextnode **=** visitedNodesIterator**.**next**();**

**if** **((**succesorBState**.**equals**(**nextnode**.**getVal**()))** **&&** **(**currentDepth **>=** nextnode**.**getDepth**()))**

**{**

shouldAddToFringe **=** **false;**

**}**

**}**

**if** **(**shouldAddToFringe**)**

nextMoves**.**add**(new** Node**(**succesorBState**,** currentNode**,** **new** ArrayList**<**Node**>(),** currentDepth **+** 1**));**

**}**

**return** nextMoves**;**

**}**

**}**

package search**;**

**import** java**.**util**.**ArrayList**;**

**import** game**.**board**.**BoardState**;**

public class Node

**{**

private BoardState val**;**

private Node parent**;**

private ArrayList**<**Node**>** children**;**

private int depth**;**

/\*\*

\* @param val

\* @param parent

\* @param children

\*/

public Node**(**BoardState val**,** Node parent**,** ArrayList**<**Node**>** children**,** int depth**)**

**{**

**super();**

**this.**val **=** val**;**

**this.**parent **=** parent**;**

**this.**setChildren**(**children**);**

**this.**setDepth**(**depth**);**

**}**

public BoardState getVal**()**

**{**

**return** val**;**

**}**

public Node getParent**()**

**{**

**return** parent**;**

**}**

public ArrayList**<**Node**>** getChildren**()**

**{**

**return** children**;**

**}**

public void setChildren**(**ArrayList**<**Node**>** children**)**

**{**

**this.**children **=** children**;**

**}**

@Override

public boolean equals**(**Object obj**)**

**{**

boolean treesEqual **=** **true;**

**if** **(!(this.**getVal**().**equals**(((**Node**)** obj**).**getVal**())))**

**{**

treesEqual **=** **false;**

**}**

**if** **((this.**getParent**()** **==** **null)** **||** **(((**Node**)** obj**).**getParent**()** **==** **null))**

**{**

**if** **(!(((this.**getParent**()** **==** **null)** **&&** **(((**Node**)** obj**).**getParent**()** **==** **null))))**

**{**

treesEqual **=** **false;**

**}**

**}**

**else** **if** **(!(this.**getParent**().**getVal**().**equals**(((**Node**)** obj**).**getParent**().**getVal**())))**

**{**

treesEqual **=** **false;**

**}**

**for** **(**Node c1 **:** **this.**getChildren**())**

**{**

**for** **(**Node c2 **:** **this.**getChildren**())**

**{**

**if** **(!(**c1**.**getVal**().**equals**(**c2**.**getVal**())))**

**{**

treesEqual **=** **false;**

**}**

**}**

**}**

**if** **(!(this.**getDepth**()** **==** **((**Node**)** obj**).**getDepth**()))**

treesEqual **=** **false;**

**return** treesEqual**;**

**}**

@Override

public String toString**()**

**{**

ArrayList**<**BoardState**>** childVals **=** **new** ArrayList**<**BoardState**>();**

**for** **(**Node children **:** **this.**getChildren**())**

**{**

childVals**.**add**(**children**.**getVal**());**

**}**

**return** **new** String**(**"\nNode: \n" **+** **this.**getVal**()** **+** "Parent: \n" **+** **this.**getParent**().**getVal**()** **+** "Children: \n" **+** childVals **+** "\nDepth: \n" **+** **this.**getDepth**());**

**}**

@Override

public int hashCode**()**

**{**

int hash **=** 3**;**

hash **=** 17 **\*** hash **+** **this.**getVal**().**hashCode**();**

hash **=** 17 **\*** hash **+** **this.**getDepth**();**

**return** hash**;**

**}**

public int getDepth**()**

**{**

**return** depth**;**

**}**

public void setDepth**(**int depth**)**

**{**

**this.**depth **=** depth**;**

**}**

**}**

package search**;**

/\*\*

\* Thrown if for a given initial board state, there is no way to generate a goal state from it

\*/

public class NoSolutionPossibleException **extends** Exception

**{**

/\*\*

\*

\*/

private static final long serialVersionUID **=** **-**1061082434711759763L**;**

/\*\*

\*

\*/

public NoSolutionPossibleException**()**

**{**

**super();**

**}**

/\*\*

\* @param message

\* @param cause

\* @param enableSuppression

\* @param writableStackTrace

\*/

public NoSolutionPossibleException**(**String message**,** Throwable cause**,** boolean enableSuppression**,** boolean writableStackTrace**)**

**{**

**super(**message**,** cause**,** enableSuppression**,** writableStackTrace**);**

**}**

/\*\*

\* @param message

\* @param cause

\*/

public NoSolutionPossibleException**(**String message**,** Throwable cause**)**

**{**

**super(**message**,** cause**);**

**}**

/\*\*

\* @param message

\*/

public NoSolutionPossibleException**(**String message**)**

**{**

**super(**message**);**

**}**

/\*\*

\* @param cause

\*/

public NoSolutionPossibleException**(**Throwable cause**)**

**{**

**super(**cause**);**

**}**

**}**

package search**;**

**import** java**.**util**.**Stack**;**

**import** game**.**board**.**BoardState**;**

public class Search

**{**

/\*\*Works back from the goal state, up to the start state, to calculate the solution path taken

\* @param solutionState

\* The final state of a solution

\* @return A stack containing all the board states leading to the solution, minus the start state

\*/

protected static Stack**<**BoardState**>** calculateSolutionPath**(**Node solutionState**)**

**{**

Stack**<**BoardState**>** solutionPath **=** **new** Stack**<**BoardState**>();**

**while** **(**solutionState**.**getParent**()** **!=** **null)**

**{**

solutionPath**.**push**(**solutionState**.**getVal**());**

solutionState **=** solutionState**.**getParent**();**

**}**

**return** solutionPath**;**

**}**

**}**

package search**;**

public enum SearchType

**{**

DEPTH\_FIRST**,** BREADTH\_FIRST**,** ITERATIVE\_DEEPENING**,** A\_STAR**;**

**}**

package tests**;**

**import** static org**.**junit**.**Assert**.**assertFalse**;**

**import** static org**.**junit**.**Assert**.**assertTrue**;**

**import** java**.**awt**.**Rectangle**;**

**import** java**.**util**.**ArrayList**;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**Test**;**

**import** game**.**board**.**BoardState**;**

**import** game**.**piece**.**Agent**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**PiecePosition**;**

**import** game**.**piece**.**PieceType**;**

**import** utilities**.**Pair**;**

public class BoardTest

**{**

BoardState board1**,** board2**;**

@Before

public void setUp**()** **throws** Exception

**{**

ArrayList**<**Block**>** blocks1 **=** **new** ArrayList**<**Block**>();**

blocks1**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**0**,**0**))));**

blocks1**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,**0**))));**

blocks1**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,**0**))));**

Agent agent1 **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,**0**)));**

board1 **=** **new** BoardState**(new** Rectangle**(**8**,** 8**),** agent1**,** blocks1**);**

ArrayList**<**Block**>** blocks2 **=** **new** ArrayList**<**Block**>();**

blocks2**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**0**,**1**))));**

blocks2**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,**1**))));**

blocks2**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,**1**))));**

Agent agent2 **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,**1**)));**

board2 **=** **new** BoardState**(new** Rectangle**(**8**,** 8**),** agent2**,** blocks2**);**

**}**

@After

public void tearDown**()** **throws** Exception

**{**

**}**

@Test

public void boardEquality**()**

**{**

assertTrue**(**board1**.**equals**(**board1**));**

assertFalse**(**board1**.**equals**(**board2**));**

**}**

**}**

package tests**;**

**import** static org**.**junit**.**Assert**.**assertFalse**;**

**import** static org**.**junit**.**Assert**.**assertTrue**;**

**import** java**.**awt**.**Rectangle**;**

**import** java**.**util**.**ArrayList**;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**Test**;**

**import** game**.**Game**;**

**import** game**.**board**.**BoardState**;**

**import** game**.**piece**.**Agent**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**BoardPiece**;**

**import** game**.**piece**.**PiecePosition**;**

**import** game**.**piece**.**PieceType**;**

**import** utilities**.**Pair**;**

public class HeuristicTest

**{**

BoardState board1**,** board2**,** board3**;**

@Before

public void setUp**()** **throws** Exception

**{**

ArrayList**<**Block**>** blocks1 **=** **new** ArrayList**<**Block**>();**

blocks1**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 0**))));**

blocks1**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**))));**

blocks1**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 2**))));**

Agent agent1 **=** **new** Agent**(**"agent1"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 3**)));**

board1 **=** **new** BoardState**(new** Rectangle**(**4**,** 4**),** agent1**,** blocks1**);**

ArrayList**<**Block**>** blocks2 **=** **new** ArrayList**<**Block**>();**

blocks2**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 0**))));**

blocks2**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**))));**

blocks2**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 2**))));**

Agent agent2 **=** **new** Agent**(**"agent2"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 3**)));**

board2 **=** **new** BoardState**(new** Rectangle**(**4**,** 4**),** agent2**,** blocks2**);**

ArrayList**<**Block**>** blocks3 **=** **new** ArrayList**<**Block**>();**

blocks3**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,** 1**))));**

blocks3**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,** 1**))));**

blocks3**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**))));**

Agent agent3 **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 3**)));**

board3 **=** **new** BoardState**(new** Rectangle**(**4**,** 4**)** **,** agent3**,** blocks3**);**

**}**

@After

public void tearDown**()** **throws** Exception

**{**

**}**

@Test

public void manhattandDistanceSameTest**()**

**{**

int cost **=** 0**;**

ArrayList**<**Block**>** currentBlockConfig **=** board1**.**getBlocks**();**

ArrayList**<**Block**>** targetBlockConfig **=** board2**.**getBlocks**();**

**for** **(**int i **=** 0**;** i **<** currentBlockConfig**.**size**();** i**++)**

**{**

cost **+=** manhattanDistance**(**currentBlockConfig**.**get**(**i**),** targetBlockConfig**.**get**(**i**));**

**}**

assertTrue**(**cost **==** 0**);**

**}**

@Test

public void manhattandDistanceDifferenceTest**()**

**{**

int cost **=** 0**;**

ArrayList**<**Block**>** currentBlockConfig **=** board1**.**getBlocks**();**

ArrayList**<**Block**>** targetBlockConfig **=** board3**.**getBlocks**();**

**for** **(**int i **=** 0**;** i **<** currentBlockConfig**.**size**();** i**++)**

**{**

int subCost **=** manhattanDistance**(**currentBlockConfig**.**get**(**i**),** targetBlockConfig**.**get**(**i**));**

cost **+=** subCost**;**

**}**

System**.**out**.**println**(**"total cost: " **+** cost**);**

assertTrue**(**cost **!=** 0**);**

**}**

private static int manhattanDistance**(**BoardPiece currentPiece**,** BoardPiece targetPiece**)**

**{**

System**.**out**.**println**(**"coords: " **+** currentPiece**.**getPosition**()** **+** " -- " **+** targetPiece**.**getPosition**());**

int xDistance **=** Math**.**abs**(**currentPiece**.**getPosition**().**x**()** **-** targetPiece**.**getPosition**().**x**());**

System**.**out**.**println**(**"xDistance: " **+** xDistance**);**

int yDistance **=** Math**.**abs**(**currentPiece**.**getPosition**().**y**()** **-** targetPiece**.**getPosition**().**y**());**

System**.**out**.**println**(**"yDistance: " **+** yDistance**);**

**return** xDistance **+** yDistance**;**

**}**

**}**

package tests**;**

**import** static org**.**junit**.**Assert**.**assertFalse**;**

**import** static org**.**junit**.**Assert**.**assertTrue**;**

**import** java**.**awt**.**Rectangle**;**

**import** java**.**util**.**ArrayList**;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**Test**;**

**import** game**.**board**.**BoardState**;**

**import** game**.**piece**.**Agent**;**

**import** game**.**piece**.**Block**;**

**import** game**.**piece**.**PiecePosition**;**

**import** game**.**piece**.**PieceType**;**

**import** search**.**Node**;**

**import** utilities**.**Pair**;**

public class TreeTest

**{**

Node tree1**,** tree2**;**

BoardState board1**,** board2**;**

@Before

public void setUp**()** **throws** Exception

**{**

ArrayList**<**Block**>** blocks1 **=** **new** ArrayList**<**Block**>();**

blocks1**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**0**,** 0**))));**

blocks1**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,** 0**))));**

blocks1**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,** 0**))));**

Agent agent1 **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 0**)));**

board1 **=** **new** BoardState**(new** Rectangle**(**8**,** 8**),** agent1**,** blocks1**);**

ArrayList**<**Block**>** blocks2 **=** **new** ArrayList**<**Block**>();**

blocks2**.**add**(new** Block**(**"a"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**0**,** 1**))));**

blocks2**.**add**(new** Block**(**"b"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**1**,** 1**))));**

blocks2**.**add**(new** Block**(**"c"**,** PieceType**.**BLOCK**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**2**,** 1**))));**

Agent agent2 **=** **new** Agent**(**"agent"**,** PieceType**.**AGENT**,** **new** PiecePosition**(new** Pair**<**Integer**,** Integer**>(**3**,** 1**)));**

board2 **=** **new** BoardState**(new** Rectangle**(**8**,** 8**),** agent2**,** blocks2**);**

tree1 **=** **new** Node**(**board1**,** **null,** **new** ArrayList**<**Node**>(),** 0**);**

tree2 **=** **new** Node**(**board2**,** **null,** **new** ArrayList**<**Node**>(),** 0**);**

**}**

@After

public void tearDown**()** **throws** Exception

**{**

**}**

@Test

public void treeEquality**()**

**{**

assertTrue**(**board1**.**equals**(**board1**));**

assertFalse**(**board1**.**equals**(**board2**));**

**}**

**}**

package utilities**;**

public class Pair**<**L**,** R**>**

**{**

private L left**;**

private R right**;**

public Pair**(**L left**,** R right**)**

**{**

**this.**left **=** left**;**

**this.**right **=** right**;**

**}**

public Pair**()**

**{**

**}**

public L getLeft**()**

**{**

**return** left**;**

**}**

public R getRight**()**

**{**

**return** right**;**

**}**

public void setLeft**(**L left**)**

**{**

**this.**left **=** left**;**

**}**

public void setRight**(**R right**)**

**{**

**this.**right **=** right**;**

**}**

@Override

public String toString**()**

**{**

**return** **new** String**(**left **+** "," **+** right**);**

**}**

**}**