University of Southampton

COMP2208: Intelligent Systems

Comparison of Search Methods

Huw Jones
27618153
hcbj1g15@soton.ac.uk

November 28, 2016

1 Approach

In order to analyse the differences in scalability, I decided to build a framework that would allow me to minimise the time spent on writing code. At the moment, I am most familiar with Java, therefore that is the language I chose to build my solution in. My code is nowhere near "good" or optimised (in terms of real time running, not nodes expanded), but it works.

1.1 The Setup

I tried to keep data structures simple and minimalistic. The state of the puzzle is stored in a **Node**. A **Node** has a **Grid** (which represents the state of the puzzle), a parent node reference, a depth (used for IDS), and a priority (used for A*). A **Grid** has a width/height, a 2D array of characters (representing blocks), and a HashMap that maps characters to their position.

1.2 The Framework

I created a framework whereby the program parameters can be manipulated via the command line.

```
-e, --exit [STATE]     Specifies exit state.
-h, --height           Sets the grid height.
    --help             Prints this help message.
-r, --ran [LONG]       Specifies the seed used for the pseudo-random number.
-s, --start [STATE]    Specifies the start state
-t, --type             Specifies the search type:
                                BFS - Breadth First Search
                                DFS - Depth First Search
                                IDS - Iterative Deepening Search
                                A* - A* Heuristic Search
-w, --width            Sets the grid width.
```

This framework allowed me to create scripts to automate my searching. It also allowed me to inject different start/finish grids, as well as injecting different grid sizes - all without having to rewrite any of my code.

In addition, I allowed the input of the random number seed. This helped during debugging my program. If a DFS search didn't work with one random number, I could provide the random number and debug that case.

2 Evidence

A **Grid** state is represented in a grid of dimensions $H \times W$. The agent is represented by a '*', blanks are represented by '-', and blocks are represented by a lowercase letter. In the evidence provided, the number above a **Grid** state is the node number.

2.1 Breadth First Search

Appendix A.1 shows the order that the program evaluated nodes. It is evident that BFS is working correctly as the tiles appear to jump around if the nodes are being read in number order. Here, the first layer of the tree are nodes 1 & 2. Nodes 3 to 5 are the second layer children of node 1. Nodes 6 to 8 are the second layer children of node 2. Nodes 9 through 11 are the third layer children of node 3. And so forth for the remainder of the nodes shown.

2.2 Depth First Search

The order of nodes evaluated is shown in Appendix A.2. With these set of nodes, the movement of the agent is fluid from state to state. Therefore, it can be concluded that the implementation of DFS is working correctly.

6 Code

6.1 Breadth First Search

```
/**
 * Breadth First Search
 *
 * @author Huw Jones
 * @since 21/10/2016
 */
public class BFS extends Search {

    private Queue<Pair<Node, DIRECTION>> nodeQueue;

    /**
     * Set up the initial environment before running the search
     */
    @Override
    protected void preRun() {
        this.nodeQueue = new ConcurrentLinkedQueue<>();
        this.rootNode = Node.createRootNode();
        this.rootNode.setGrid(this.startGrid);
    }

    /**
     * Where the actual search runs
     */
    @Override
    protected void runSearch() {
        this.currentNode = rootNode;

        while (true) {
            if (currentDirection != null) {
                try {
                    currentNode.setGrid(
                            GridController.move(
                                    currentNode.getParent().getGrid(),
                                    currentDirection
                                    )
                    );
                    numberOfNodes++;
                    if (this.checkExitCondition(currentNode.getGrid())) {
                        completed(currentNode);
                        break;
                    }
                } catch (InvalidDirectionException e) {
                    nextNode();
                    continue;
                }
            }

            for (DIRECTION direction : DIRECTION.values()) {
                nodeQueue.add(new Pair<>(new Node(currentNode), direction));
            }
            nextNode();
        }
    }

    @Override
    protected void nextNode() {
        currentPair = nodeQueue.poll();
        currentNode = currentPair.getKey();
        currentDirection = currentPair.getValue();
    }
}
```

6.2 Depth First Search

```
/**
 * Depth First Search
 *
 * @author Huw Jones
 * @since 27/10/2016
 */
public class DFS extends Search {

    private Stack<Pair<Node, DIRECTION>> nodeStack;

    @Override
    protected void preRun() {
        this.nodeStack = new Stack<>();
        this.rootNode = Node.createRootNode();
        this.rootNode.setGrid(this.startGrid);
    }

    @Override
    protected void runSearch() {
        // Init
        ArrayList<DIRECTION> directions = new ArrayList<>(4);
        Arrays.stream(DIRECTION.values()).forEach(directions::add);

        currentNode = rootNode;
        while (true) {



            if (currentDirection != null) {
                try {
                    currentNode.setGrid(
                            GridController.move(
                                    currentNode.getParent().getGrid(),
                                    currentDirection
                            )
                    );
                    numberOfNodes++;
                    if (this.checkExitCondition(currentNode.getGrid())) {
                        completed(currentNode);
                        break;
                    }
                } catch (InvalidDirectionException e) {
                    nextNode();
                    continue;
                }
            }

            Collections.shuffle(directions, this.random);

            for (DIRECTION direction : directions) {
                nodeStack.push(new Pair<>(new Node(currentNode), direction));
            }

            nextNode();
        }
    }

    @Override
    protected void nextNode() {
        currentPair = nodeStack.pop();
        currentNode = currentPair.getKey();
        currentDirection = currentPair.getValue();
    }
}
```

6.3 Iterative Deepening Search

```
/**
 * Iterative Deepening Search
 *
 * @author Huw Jones
 * @since 12/11/2016
 */
public class IDS extends Search {

    private Stack<Pair<Node, DIRECTION>> nodeStack;
    private ArrayList<DIRECTION> directions;
    private int depth;

    /**
     * Set up the initial environment before running the search
     */
    @Override
    protected void preRun() {
        this.nodeStack = new Stack<>();
        this.rootNode = Node.createRootNode();
        this.rootNode.setGrid(this.startGrid);
        directions = new ArrayList<>(4);
        Arrays.stream(DIRECTION.values()).forEach(directions::add);

        currentNode = rootNode;
        currentPair = null;
        currentDirection = null;
    }

    /**
     * Where the actual search runs
     */
    @Override
    protected void runSearch() {
        while (true) {
            // Check if we've hit the depth limit
            if (currentNode.getDepth() > depth) {
                // If we have no more nodes in the stack, increase the depth
                if(this.nodeStack.size() == 0) {
                    increaseDepth();
                } else {
                    // Otherwise continue processing the stack
                    nextNode();
                }
                continue;
            }



            if (currentDirection != null) {
                try {
                    // Process the move and store the new state in the node
                    currentNode.setGrid(
                            GridController.move(
                                    currentNode.getParent().getGrid(),
                                    currentDirection
                            )
                    );
                    numberOfNodes++;
                    // Check if the grid meets the exit condition, if so, exit the search
                    if (this.checkExitCondition(currentNode.getGrid())) {
                        completed(currentNode);
                        break;
                    }
                } catch (InvalidDirectionException e) {
                    if(nodeStack.size() == 0){
                        increaseDepth();
                    } else {
                        nextNode();
                    }
                    continue;
                }
            }

            Collections.shuffle(directions, this.random);
```

```
            // Push new directions on the stack to be processed
            for (DIRECTION direction : directions) {
                nodeStack.push(new Pair<>(new Node(currentNode), direction));
            }

            nextNode();
        }
        System.out.println("Max Iterative Depth:");
        System.out.println(depth);
    }

    /**
     * Gets the next node off of the stack
     */
    @Override
    protected void nextNode() {
        currentPair = nodeStack.pop();
        currentNode = currentPair.getKey();
        currentDirection = currentPair.getValue();
    }

    /**
     * Increases the depth of the search
     */
    private void increaseDepth(){
        // Reset the search environment
        preRun();
        // Increment depth
        depth++;
        // Log new depth
        System.out.println("\r\nDepth increased: "+ depth);
    }
}
```

6.4 A* Heuristic Search

```
/**
 * A* Search
 *
 * @author Huw Jones
 * @since 27/11/2016
 */
public class AStar extends Search {

    PriorityQueue<Node> nodeQueue;

    /**
     * Set up the initial environment before running the search
     */
    @Override
    protected void preRun() {
        this.nodeQueue = new PriorityQueue<>(new PriorityComparator());
        this.rootNode = Node.createRootNode();
        this.rootNode.setGrid(this.startGrid);
    }

    /**
     * Where the actual search runs
     */
    @Override
    protected void runSearch() throws Exception {
        ArrayList<GridController.DIRECTION> directions = new ArrayList<>(4);
        Arrays.stream(GridController.DIRECTION.values()).forEach(directions::add);

        this.currentNode = rootNode;
        while_loop:
        while (true) {
            numberOfNodes++;

            // Check the if the node satisfies the exit condition
            if (this.checkExitCondition(currentNode.getGrid())) {
                completed(currentNode);
                break;
            }

            for (GridController.DIRECTION direction : directions) {
                try {
                    // Process the move and store the new state in the node
                    Node newNode = new Node(currentNode);
                    newNode.setGrid(
                            GridController.move(
                                    newNode.getParent().getGrid(),
                                    direction
                            )
                    );

                    // Calculate the node heuristic score and add it to the queue
                    newNode.setPriority(
                            calculatePriority(newNode)
                    );
                    nodeQueue.add(newNode);
                } catch (InvalidDirectionException e) {
                }
            }
            // Process the next node
            nextNode();
        }
    }

    @Override
    protected void nextNode() {
        currentNode = nodeQueue.poll();
    }

    /**
     * Calculates the priority (heuristic score) of the node
     *
     * @param node Node to calculate score for
     * @return Score for that node
     */
```

```
    private int calculatePriority(Node node) {
        int score = 0;
        score += getManhattanDistance(node.getGrid());
        score += getTilesInCorrectPlace(node.getGrid());
        score += node.getDepth();
        return score;
    }

    /**
     * Calculates the Manhattan Distance Heuristic
     *
     * @param grid Grid to calculate
     * @return score
     */
    private int getManhattanDistance(Grid grid) {
        int score = 0;
        ArrayList<Block> blocks = grid.getBlocks();
        for (Block block : blocks) {
            try {
                // Get the difference between the exit position and the current block position
                Position difference = this.exitGrid
                        .getBlock(block.getID())
                        .getPosition()
                        .subtract(block.getPosition());
                // Add the X/Y distance from target block position (distance not displacement,
                    hence Math.abs)
                score += Math.abs(difference.getX());
                score += Math.abs(difference.getY());
            } catch (NoSuchElementException ex) {
            }
        }
        return score;
    }

    /**
     * Calculates the number of tiles in the correct place
     *
     * @param grid Grid to calculate
     * @return score
     */
    private int getTilesInCorrectPlace(Grid grid) {
        ArrayList<Block> blocks = grid.getBlocks();
        int score = 0;
        for (Block block : blocks) {
            try {
                // Increment score for every incorrectly positioned block
                if (!this.exitGrid.getBlock(block.getID()).getPosition().equals(block.
                    getPosition())) score++;
            } catch (NoSuchElementException ex) {

            }
        }
        return score;
    }

    /**
     * Finds the highest priority node (node with lowest score)
     * Used to sort the PriorityQueue
     */
    private class PriorityComparator implements Comparator<Node> {
        @Override
        public int compare(Node o1, Node o2) {
            return o1.getPriority() - o2.getPriority();
        }
    }
}
```

Appendices

A Output Evidence

A.1 Breadth First Search

```
1      2      3      4
-----  -----  -----  -----
-----  -----  ---*  -----
---*   -----  -----  -----
abc-   ab*c   abc-   abc*

5      6      7      8
-----  -----  -----  -----
-----  -----  -----  -----
--*-   --*-   --*-   -----
abc-   ab-c   abc-   a*bc

9      10     11     12
---*   -----  -----  -----
-----  -----  --*-   -----
-----  ---*   -----  ---*
abc-   abc-   abc-   abc-

13     14     15     16
-----  -----  -----  -----
-----  --*-   -----  -----
-----  -----  ---*   --c-
ab*c   abc-   abc-   ab*-
```

A.2 Depth First Search

```
1      2      3      4
-----  -----  -----  -----
-----  -----  -----  -----
-----  -----  ---*   --*-
ab*c   abc*   abc-   abc-

5      6      7      8
-----  -----  -----  -----
-----  -----  -----  -----
--c-   --c-   --c-   --c-
ab*-   ab-*   ab*-   ab-*

9      10     11     12
-----  -----  -----  -----
-----  -----  -----  ---*
--c-   --c-   --c*   --c-
ab*-   ab-*   ab--   ab--

13     14     15     16
-----  -----  -----  -----
--*-   --c-   --c-   -*c-
--c-   --*-   -*--   -----
ab--   ab--   ab--   ab--
```

A.3 Iterative Deepening Search

```
Depth increased: 1
1      2
-----  -----
-----  -----
-----  ---*
ab*c   abc-

Depth increased: 2
3      4      5      6
-----  -----  -----  -----
-----  -----  -----  -----
-----  -----  -----  --*-
ab*c   abc*   a*bc   ab-c

7      8      9      10
```

```
————  ————  ————  ————
————  ————  ————  ———*
———*  ————  ——*—  ————
abc—  abc∗  abc—  abc—

Depth increased: 3
11    12    13    14
————  ————  ————  ————
————  ————  ————  ————
———*  ————  ———*  ————
abc—  abc∗  abc—  ab∗c

15    16    17
————  ————  ————
————  ————  ————
——*—  ——c—  —*——
abc—  ab∗—  abc—
```

A.4 A* Heuristic Search

```
1     2     3     4
————  ————  ————  ————
————  ————  ————  ———*
———*  ————  ——*—  ————
abc—  ab∗c  abc—  abc—

5     6     7     8
————  ————  ————  ————
————  ————  ————  ————
————  ————  —*——  —b——
abc∗  abc∗  abc—  a∗c—

9     10    11    12
————  ————  ————  ————
————  ————  ————  ————
—b——  —b——  —b∗—  —b——
ac∗—  ∗ac—  ac——  ac—∗

13    14    15    16
———*  ————  ————  ————
————  ————  ————  ————
————  —b——  ———*  ———*
abc—  ac∗—  abc—  abc—
```

B Example Output