COMP2208 Assignment: Search Methods

Vlad Niculescu

ID: 29654556

November 29, 2018

Approach

As a programming language for this coursework, I chose Java, since it is the language I am most comfortable with. The game board is represented by a 'State' class, which contains a 'Tile' subclass. The board itself is represented by a square matrix of Tiles, while the tile class contains information such as coordinates and the name of each tile. The board could be interpreted as a matrix of chars. The 'Fringe' class represents a Tree which is used (to some extend) in all search methods. I choose to represent the tree as an Array List for Breadth-First Search, Depth-First Search and Iterative Deepening Search because it felt more versatile at the time. For A* I used a priority queue since it makes the search so much easier to implement.

For each search method, the fringe (or queue) was initialized with the root (initial state) as its only element. For BFS, a node would be checked and then expanded if it wasn't the goal state. Because the node's children were added to the end of the Array List, for BFS it was enough to just go through the list end expand each node as I went through it. My laptop was not able to reach the goal state (due to lack of memory) but on smaller inputs, BFS worked perfectly.

For DFS, a node's children would be expanded right after checking. Out of these children, I randomly picked one and deleted the others from the Array List (this was not necessary for time complexity but there was no need to keep in memory the paths I would not go through). Since the tree is infinite in length (if we are in a certain state, we can always make at least 2 moves), we would never reach a leaf and have to go back.

For IDS, I recursively called a Depth-Limited Search method with increasing limit. After searching a tree up to a limit, if there was no solution, a new Tree would be generated (with the root as its only element) and searched up until the new limit (old limit + 1).

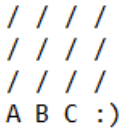
For A*, I chose as the evaluation function the sum of a node's depth (cost so far) and the Manhattan distance of that node (estimated cost left from node to goal). For this type of search, I used a Priority Queue that would sort the elements depending on their evaluation function values. Whenever a node is checked, it is popped from the queue. If the node is not the goal state, it is expanded and its children are added to the queue. The process continues until the goal state is reached. The heuristic is admissible since it never overestimates the cost to reach the goal. For example, if a tile's Manhattan distance is 5, that means it is tiles away from the position it is supposed to be in. A search method might require more than 5 moves to reach the specific state, but it will never require less than 5.

Evidence

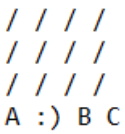
In the following paragraph I will show how the search methods work on some small inputs.

Depth-First Search

Let the initial and final states be

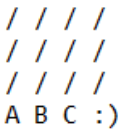


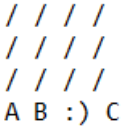
Initial state

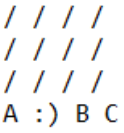


Final State

The algorithm sometimes guesses the optimal path, generating the following path (6 nodes generated)



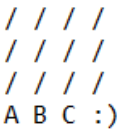




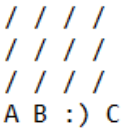
If we are not lucky, the algorithm can generate some random moves that eventually lead to the solution, but require many additional nodes (`311226 nodes generated in the following example`)

Breadth-First Search

Let the initial and final states be

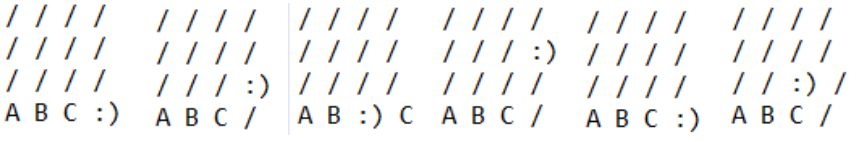


Initial state

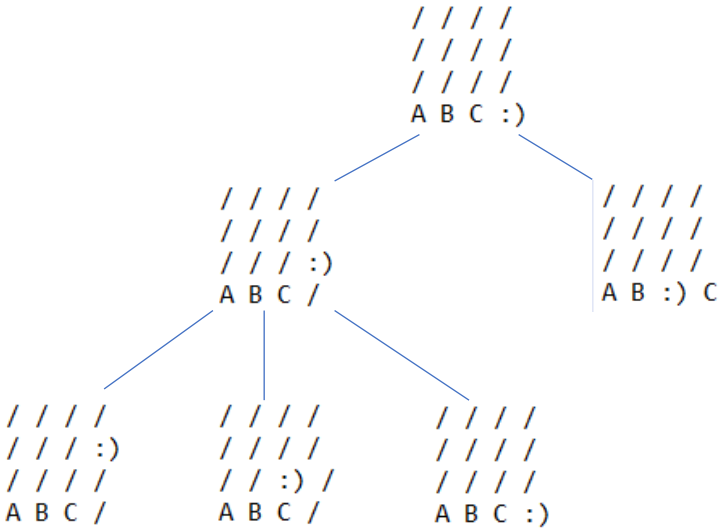


Final State

Even though there is 1 move necessary, the algorithm generates 6 nodes. The fringe generated is an array list (first image). I arranged the elements into a tree to make it easier to follow (second image).



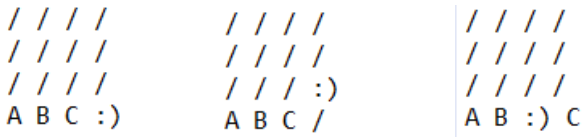
(Fig. 1)



(Fig. 2)

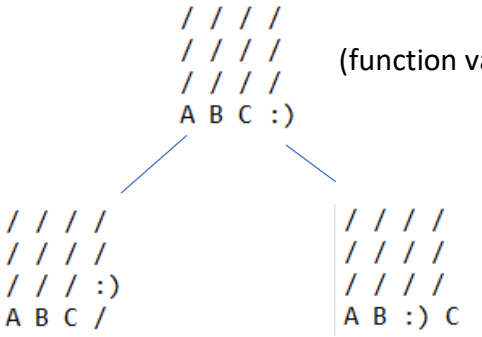
IterativeDeepeningSearch

For the initial and final state as the ones above, 4 nodes are generated. 1 (the initial state) for the first iteration and the following 3 more for the second.



*A**

For the initial and final state as the ones above, 3 nodes are generated.

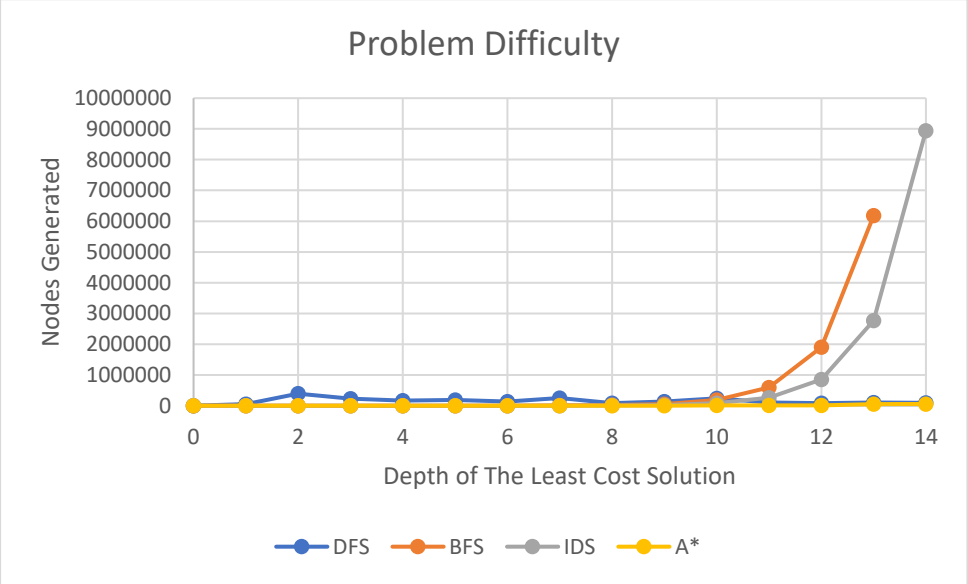


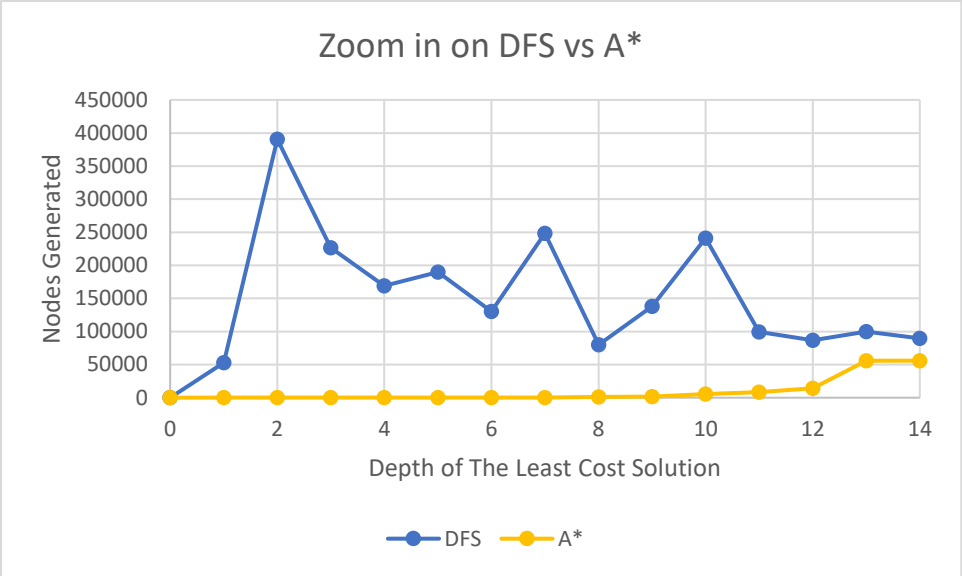
(function value = 1+1) (function value = 1+0)

The root is popped and checked. Since it is not the solution, the children are expanded and added into the queue. The child with the lowest function value is popped and checked after the root. That node is the solution, so we don't need to check other nodes after that.

Scalability Study

For this part, I plotted a graph representing the increase/decrease in the number of nodes generated in relation to the depth at which the optimal solution is found. I will also compare the search methods based on space complexity, which does not appear in the graphs. For DFS, the value plotted is the average of 10 searches.





The graphs above give us some useful information regarding the time complexity of the different search methods. While DFS appears to be doing better than most of the other search methods, we can see that the number of nodes generated is somewhat random, making the method very unreliable. Furthermore, DFS is not optimal. When it comes to space complexity, DFS stores only the nodes it goes through, but since it searches many unnecessary nodes, it can occupy more memory than expected.

BFS is another kind of blind search. This time, the search is optimal, but the space complexity is very high, since whenever it checks a node, it also generates the node's children. While in DFS we could discard nodes we didn't need from memory, BFS keeps ALL the nodes in the memory. For depth 14, I could not compute BFS on my laptop, since the space complexity grows exponentially, and I quickly ran out of memory.

IDS tries to fix the space issues of the previous search methods. This search methods expand nodes up to a certain point, keeping the nodes on the 'limit' level from expanding. Whenever a limit is reached, the memory is emptied and a new search begins on a tree with a bigger limit. From the graph we can observe that IDS actually has a better time complexity than BFS in the long run. This is because IDS does not generate the children on the nodes on the last level. This might not seem like much but generating those extra nodes more than DOUBLES the amount of memory required to store all the nodes.

Finally, we look at A*, the only heuristic algorithm implemented. We can observe that A* performs much better when it comes to time complexity than any other algorithm implemented. This is because the tree is searched with a vague idea of the right direction it is supposed to head in, rather than blindly generating all possible outcomes and trying them all. A* undoubtedly is the best search method of all 4.

Extras and limitations

No any extras attempts were made. While measuring space complexity or increasing the grid size extras that could've easily been implemented, a severe failure in time management on my part stopped me from attempting them.

When it comes to limitations, there are certainly some aspects which I could've approached in a better way. First of all, the use of Array Lists instead of Stacks or Queues was probably not the best idea. While it doesn't affect the time or space complexity, I believe that it affects the running time of my programme. Relying so heavily on the Array List left me in the end with some redundant data, since for A* I had to copy the contents of the Array List into the Priority Queue and delete them from the list afterwards. When it comes to the graph, the data provided might not be the most accurate. I struggled to find solutions that would require a certain number of moves, so some of the data was collected by using different paths. Given the nature of the board and the fact that the number of children varies for each state, different paths mean different numbers of nodes generated. While I believe that my data was not heavily corrupted by this, I still consider it something to take into consideration.

References

I only used the slides from the COMP2208 lectures as guidance for this coursework.

Code

The State class

```
public class State
{
	//the board of the game
	Tile[][] board;
	int size;

	//the state the board is in
	public State(int n)
	{
		size = n;
		//initialised with the starting position
		board = new Tile[n][n];
		for(int i = 0; i < n; i++)
			for(int j = 0; j < n; j++)
				board[i][j] = new Tile("/",i,j);

		board[n-1][0].name = "A";
		board[n-1][1].name = "B";
		board[n-1][2].name = "C";
		board[n-1][3].name = ":)";
	}
```

```
public boolean isFinal()
{

	if(board[1][1].name != "A")
		return false;
	if(board[2][1].name != "B")
		return false;
	if(board[3][1].name != "C")
		return false;
	return true;
}

public class Tile
{
	//Coordinates
	int row,col;
	public String name;

	public Tile(String name,int i, int j)
	{
		this.name = name;
		row = i;
		col = j;
	}
}

//find the coordinates of a tile
public Tile find(String name)
{
	for(int i = 0; i < board.length; i++)
		for(int j = 0; j < board.length; j++)
			if(board[i][j].name.equals(name))
			{
				return board[i][j];
			}
	return null;
}

//moving the agent
public int move(String direction)
{
	int n = board.length;
	Tile agent,aux;
	agent = this.find(":)");

	//move right
	if(direction.equals("right"))
	{
		if(agent.col == n-1)
		{
			return -1;
		}
		else
		{
			aux = agent;
			agent = board[agent.row][agent.col + 1];
			board[agent.row][agent.col - 1] = aux;
			board[agent.row][agent.col - 1].name = agent.name;
```

```
                agent.name = ":)";
                aux = null;
                return 0;
        }
}
//move left
else if(direction.equals("left"))
{
        if(agent.col == 0)
        {
                return -1;
        }
        else
        {
                aux = agent;
                agent = board[agent.row][agent.col - 1];
                board[agent.row][agent.col + 1] = aux;
                board[agent.row][agent.col + 1].name = agent.name;
                agent.name = ":)";
                aux = null;
                return 0;
        }
}
//move up
else if(direction.equals("up"))
{
        if(agent.row == 0)
        {
                return -1;
        }
        else
        {
                aux = agent;
                agent = board[agent.row - 1][agent.col];
                board[agent.row + 1][agent.col] = aux;
                board[agent.row + 1][agent.col].name = agent.name;
                agent.name = ":)";
                aux = null;
                return 0;
        }
}
//move down
else if(direction.equals("down"))
{
        if(agent.row == n-1)
        {
                return -1;
        }
        else
        {
                aux = agent;
                agent = board[agent.row + 1][agent.col];
                board[agent.row - 1][agent.col] = aux;
                board[agent.row - 1][agent.col].name = agent.name;
                agent.name = ":)";
                aux = null;
                return 0;
        }
}
```

```
			else
			{
				return -1;
			}

		}

		//display the current state
		public void display()
		{
			for(int i = 0; i < this.board.length; i++)
			{
				for(int j = 0; j< this.board.length; j++)
					System.out.print(this.board[i][j].name + " ");
				System.out.println();
			}
			System.out.println();
		}
}
```

The Fringe class

```
import java.util.ArrayList;

public class Fringe {

		Node root;
		ArrayList<Node> Tree;

		public Fringe()
		{
			root = new Node(new State(4), null);
			Tree = new ArrayList<Node>();
			Tree.add(root);
		}

		public void display()
		{
			for(Node it : Tree)
				it.s.display();
		}

		public void expandNode(Node node)
		{
			node.children = new ArrayList<Node>();
			if(node.s.move("up") == 0)
			{
				Node c = new Node(node.s, node);
				node.children.add(c);
				Tree.add(c);
				node.s.move("down");
			}
			if(node.s.move("down") == 0)
			{
				Node c = new Node(node.s, node);
				node.children.add(c);
				Tree.add(c);
				node.s.move("up");
```

```
		}
		if(node.s.move("left") == 0)
		{
			Node c = new Node(node.s, node);
			node.children.add(c);
			Tree.add(c);
			node.s.move("right");
		}
		if(node.s.move("right") == 0)
		{
			Node c = new Node(node.s, node);
			node.children.add(c);
			Tree.add(c);
			node.s.move("left");
		}
	}

	public class Node
	{
		State s;
		Node parent;
		ArrayList<Node> children;
		int depth;

		public Node(State ps, Node p)
		{
			s = new State(ps.size);
			for(int i = 0; i < s.size; i++)
				for(int j = 0; j < s.size; j++)
					s.board[i][j].name = ps.board[i][j].name;

			parent = p;
			if(parent == null)
				depth = 0;
			else
				depth = p.depth + 1;
		}
	}
}
```

The BreadthFirstSearch class

```
import java.util.ArrayList;

public class BreadthFirstSearch {

	Fringe f = new Fringe();
	int j = 0;

	public void findSolution()
	{
		for(int i=0; i < f.Tree.size();i++)
		{
			Fringe.Node it = f.Tree.get(i);

			if(it.s.isFinal() != true)
			{
				f.expandNode(it);
```

```
                }
                else
                {
                        j = i;
                        break;
                }
        }

        f.Tree.get(j).s.display();
        System.out.println("BFS: node " + f.Tree.size()  + " at depth " +
f.Tree.get(j).depth + "\n");

    }

    public void displaySolution()
    {
        ArrayList<Fringe.Node> solution = new ArrayList<Fringe.Node>();
        Fringe.Node n = f.Tree.get(j);
        while(n.depth != 0)
        {
                solution.add(n);
                n = n.parent;
        }
        f.root.s.display();
        for(int i = solution.size() - 1; i >=0 ; i--)
        {
                solution.get(i).s.display();
        }
    }
}
```

The DepthFirstSearch class

```
import java.util.Random;

public class DepthFirstSearch {

    Fringe f = new Fringe();
    Random r = new Random();
    int nodeNumber = 1;

    public int findSolution()
    {

        Fringe.Node n = f.root;
        int pos = 0;
        int child = 0;
        int i = 0;

        while(n.s.isFinal() != true)
        {
                f.expandNode(n);
                for(Fringe.Node nodes : n.children)
                {
                        nodeNumber++;
                }
                //we pick a random child
                child = r.nextInt((f.Tree.size() - pos - 2) + 1) + pos + 1;
```

```
                    //we remove from the tree all children that were not picked for
search
                    i = pos + 1;
                    while(i < child)
                    {
                          f.Tree.remove(i);
                          child --;
                    }
                    i = child + 1;
                    while(i <= (f.Tree.size() - 1))
                    {
                          f.Tree.remove(i);
                    }
                    //the child becomes the next parent
                    n = f.Tree.get(child);
                    pos = child;
             }
             f.Tree.get(f.Tree.size()-1).s.display();
             System.out.println("DFS: node " + nodeNumber + " at depth " +
f.Tree.get(f.Tree.size()-1).depth);
             return nodeNumber;

      }
}
```

The IterativeDeepeningSearch class

```
import java.util.ArrayList;

public class IterativeDeepeningSearch {

      Fringe f = new Fringe();
      Fringe.Node found;
      int nodeNumber = 0;

      public void findSolution()
      {
             for(int i=0; ;i++)
             {
                    f.Tree.clear();
                    f.Tree.add(f.root);
                    found = DepthLimitedSearch(f.root,i);
                    if(found != null)
                    {
                          found.s.display();
                          System.out.println("IDS: node " + nodeNumber + " found
at depth " +  found.depth + " with limit = " + i + "\n");
                          break;
                    }
             }
      }

      public Fringe.Node DepthLimitedSearch(Fringe.Node node, int limit)
      {
             nodeNumber++;
             if(limit == 0 && node.s.isFinal())
                    return node;
             if(limit > 0)
             {
```

```
                f.expandNode(node);
                for(Fringe.Node child : node.children)
                {
                        Fringe.Node result = DepthLimitedSearch(child, limit -
1);
                        if(result != null)
                        {
                                return result;
                        }
                }
        }
        return null;
    }

    public void displaySolution()
    {
        ArrayList<Fringe.Node> solution = new ArrayList<Fringe.Node>();
        Fringe.Node nodeAdded = found;
        while(nodeAdded.depth != 0)
        {
                solution.add(nodeAdded);
                nodeAdded = nodeAdded.parent;
        }
        f.root.s.display();
        for(int i = solution.size() -1; i >= 0; i--)
                solution.get(i).s.display();
    }
}
```

The AStar class

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class AStar {

    int nodeNumber = 1;
    State solution;
    Fringe f = new Fringe();
    PriorityQueue<Fringe.Node> queue = new PriorityQueue<Fringe.Node>(1, new
NodeComparator());


    public AStar()
    {
        solution = new State(4);
        for(int i = 0; i < 4; i++)
                for(int j = 0; j < 4; j++)
                        solution.board[i][j].name = "/";
        solution.board[1][1].name = "A";
        solution.board[2][1].name = "B";
        solution.board[3][1].name = "C";

        queue.add(f.root);
    }

    public void findSolution()
    {
```

```
		Fringe.Node node = queue.poll();
		while(node.s.isFinal() == false)
		{
			f.expandNode(node);
			for(Fringe.Node child : node.children)
			{
				queue.add(child);
				nodeNumber++;
			}
			node = queue.poll();

			//clear the fringe since we don't need it
			f.Tree.clear();
		}
		node.s.display();
		System.out.println("A*: node " + nodeNumber + " found at depth " +
node.depth);
	}

	public void displaySolution()
	{

	}

	public int calcManDis(State state)
	{
		int a = 0;
		int b = 0;
		int c = 0;

		a = Math.abs((state.find("A").col) - solution.find("A").col) +
Math.abs((state.find("A").row) - solution.find("A").row);
		b = Math.abs((state.find("B").col) - solution.find("A").col) +
Math.abs((state.find("B").row) - solution.find("B").row);
		c = Math.abs((state.find("C").col) - solution.find("C").col) +
Math.abs((state.find("C").row) - solution.find("C").row);

		return (a + b + c);
	}

	public class NodeComparator implements Comparator<Fringe.Node>
	{
		public int compare(Fringe.Node n1, Fringe.Node n2)
		{
			int dis1 = n1.depth + calcManDis(n1.s);
			int dis2 = n2.depth + calcManDis(n2.s);

			if(dis1 < dis2)
			{
				return -1;
			}
			else if (dis1 > dis2)
			{
				return 1;
			}
			return 0;
		}
	}
}
```

The Main class

```
public class Main
{
	public static void main(String[] args)
	{
		BreadthFirstSearch b = new BreadthFirstSearch();
		DepthFirstSearch d = new DepthFirstSearch();
		IterativeDeepeningSearch i = new IterativeDeepeningSearch();
		AStar a = new AStar();

		d.findSolution();
		b.findSolution();
		i.findSolution();
		a.findSolution();
	}
}
```