CS 165a – Artificial Intelligence
Lab 1 Report

Search & Pathfinding

By: Hiram A Rios, Matthew S Montoya
ID: 80552404, 88606727
Instructor: Dr. Christopher Kiekintveld
TA: Khandoker Rahad

1 Introduction

In this lab, a program will utilize three search methods (Breadth-First Search, Iterative Deepening-Depth-First Search, and A* Search) on three different tree (map) sizes; that is, 5x5, 10x10, and 20x20, which are imported from a .txt file. The given starting positions are indicated within the text files. After the search function successfully executes, the results with regards to timing and the number of expanded nodes will be displayed for the user to view.

2 Proposed Solution & Design Implementation

To achieve the end goal, we will implement two different classes. The first class will be a node class that stores the coordinates and all necessary helper data. This node class will be used for all three searches inside of our second class, which also doubles as our runner file and will contain all of the helper logic for the searches, including calculating the Manhattan Distance, expanding our tree nodes to reveal any children nodes, as well as generating the 2-D map imported from the text file.

2.1— `generate_map_space()`

Called from our `main()` method, this method will import the specified text file the user requests and create the 2D map space via a 2D array utilizing `FileReader()` and `BufferedReader()`, which is why the contents of this function are wrapped inside of a try-catch statement. After creating the map space, it will then go ahead and populate the map space with the integer values from the text file. This will then return the populated 2D map to the `main()` method.

2.2— `search_algorithms()`

Called from `main()`, this method will invoke the three different search functions, `breadth_first_search()`, `iterative_deepening_search()`, and `a_star_search()`. This is a non-return method.

2.3— `breadth_first_search()`

Called from `search_algorithms()`, this method will accept the map space as a 2D array. It will then initialize a 3-minute timer and begin implementing the BFS algorithm. Within the algorithm, it will search and expand the current node to reveal any child nodes, while also checking if this is a goal node, that is, we reached the goal. Any children nodes will then be added to memory. This is a non-return method.

2.4— `iterative_deepening_search()`

Called from `search_algorithms()`, this method will accept the map space as a 2D array. Using a Stack of nodes as the fringe, it will begin pushing in our starting node. It will then initialize a 3-minute timer and begin implementing the IDS algorithm. Within the algorithm, it will search and expand the current node by popping the node from the stack to reveal any child nodes, while also checking if this is a goal node, that is, we reached the goal. Any children nodes will then be added to memory. This is a non-return method.

2.5— `a_star_search()`

Called from `search_algorithms()`, this method will accept the map space as a 2D array. Using a `PriorityQueue()` of nodes, it will begin pushing in our starting node. It will then initialize a 3-minute timer and begin implementing the A* Search algorithm. Within the algorithm, it will search and expand the current node to reveal any child nodes, while also checking if this is a goal node, that is, we reached the goal. Any children nodes will then be added to memory. This is a non-return method.

2.6— `falsify_visited_nodes()`

Called from `iterative_deepening_search()`, this method will accept the nodes we've visited as a 2D Boolean array. This non-return method sets all the visited nodes in the 2D map to false, resetting the map from `breadth_first_search()`.

2.7— `is_goal_node()`

Called from all three search methods, this function takes in the node we're currently exploring and determines whether or not it is our goal node. It will return a Boolean value indicating whether or not the node we're currently exploring is our goal node.

2.8— `manhattan_distance()`

Invoked from all three search functions, as well as when we generate our successor nodes, this function returns the integer value of the Manhattan Distance.

2.9— `display_path_information()`

Invoked from all three search functions, this function will display the cost of the path of the current search algorithm. Taking in our current node and path cost and is a non-return method.

2.10— `generate_successor_nodes()`

Invoked from all three search functions, this function expands our current node and generates its child nodes in our tree. Taking in our current node, our goal node, the 2D map space, as well as the 2D Boolean array of the visited nodes, this function returns a node array of successor nodes.

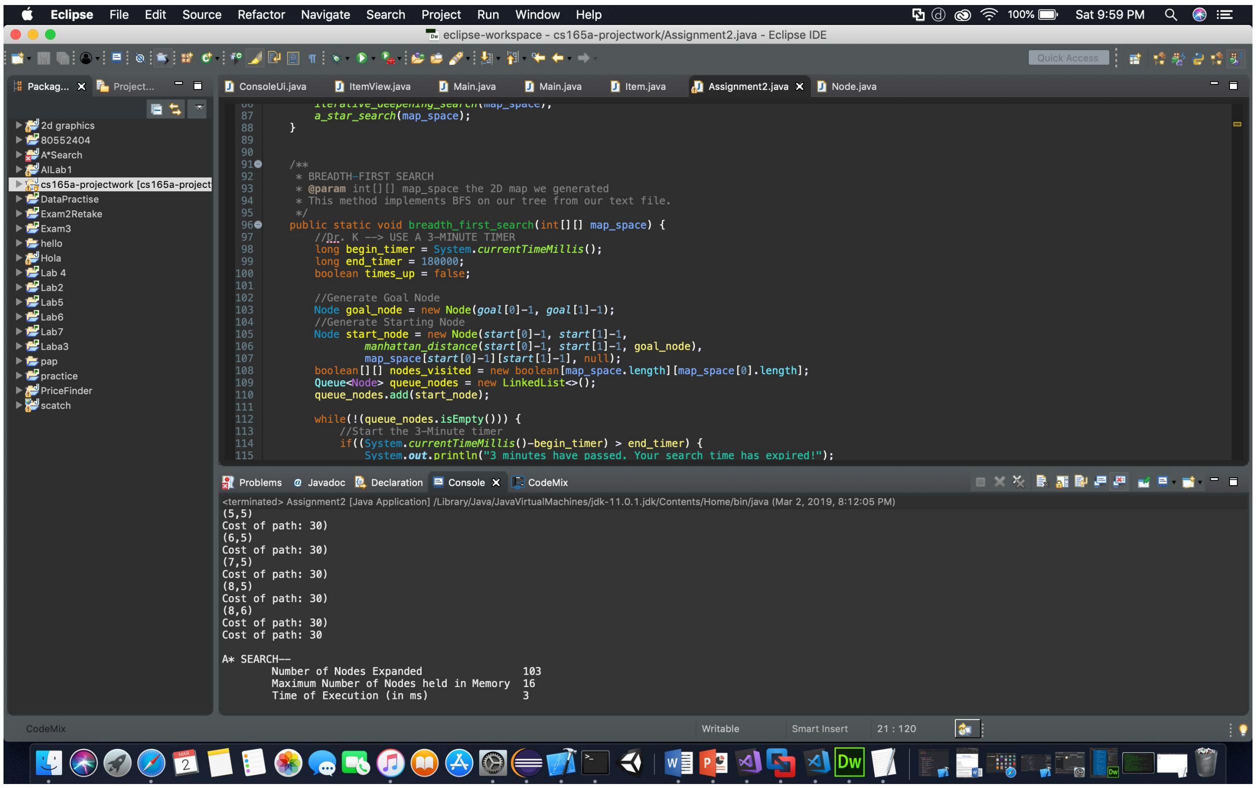
2.11— `print_search_results()`

Invoked from all three search functions, this function prints the results for the search functions, including the total number of nodes expanded, the nodes in memory, and the length of time it took to complete the search. This is a non-return function.

3 Experimental Results

3.1— Program Output

The output of the code (found in the appendix) is shown below. Note: This image includes the beginning of the code (which can also be found in the *Appendix* section), as proof the executed code is not the work of others.



Execution using A* Search.

4 Conclusions

In this lab, we learned the logic in utilizing different tree search algorithms (BFS, IDS, A*). Our results are posted below.

```
        Path cost:                          24

ITERATIVE DEEPENING DFS--
        Number of Nodes Expanded:           9
        Maximum Number of Nodes held in Memory: 6
        Time of Execution (in ms):          1
        Path Cost:                          7

A* SEARCH--
        Number of Nodes Expanded:           18
        Maximum Number of Nodes held in Memory: 6
        Time of Execution (in ms):          1

Congrats! Your program has finished executing!
hiramrios@Hirams-MBP ~/D/c/test_case_files> java -jar AIHW2Lab.jar test_case_10_10.txt IDS
Map Size: 10 10
Starting Position: 8 6
Goal Position: 1 1
Path cost: 38

BREADTH-FIRST SEARCH--
        Number of Nodes Expanded:           12
        Maximum Number of Nodes held in Memory: 83
        Time of Execution (in ms):          6
        Path cost:                          42

ITERATIVE DEEPENING DFS--
        Number of Nodes Expanded:           21
        Maximum Number of Nodes held in Memory: 16
        Time of Execution (in ms):          2
        Path Cost:                          30

A* SEARCH--
        Number of Nodes Expanded:           103
        Maximum Number of Nodes held in Memory: 16
        Time of Execution (in ms):          1

Congrats! Your program has finished executing!
hiramrios@Hirams-MBP ~/D/c/test_case_files> java -jar AIHW2Lab.jar test_case_20_20.txt IDS
Map Size: 20 20
Starting Position: 2 18
Goal Position: 20 2
Path cost: 167

BREADTH-FIRST SEARCH--
        Number of Nodes Expanded:           21
        Maximum Number of Nodes held in Memory: 366
        Time of Execution (in ms):          11
        Path cost:                          542

ITERATIVE DEEPENING DFS--
        Number of Nodes Expanded:           151
        Maximum Number of Nodes held in Memory: 172
        Time of Execution (in ms):          1
        Path Cost:                          82

A* SEARCH--
        Number of Nodes Expanded:           515
        Maximum Number of Nodes held in Memory: 172
        Time of Execution (in ms):          4

Congrats! Your program has finished executing!
hiramrios@Hirams-MBP ~/D/c/test_case_files>
```

5 Appendix

5.1— Documentation & Disclosure

- Methods are documented using the following format—

```
/**METHOD NUMBER:
*
* BRIEF MENTION OF METHOD'S PURPOSE
*/
public static void method_name(parameters) {
}
```

- Non-obvious code is documented using the following format—

```
for(int i=0; i<x.length-1; i++){//summarizes the function of the loop
        x[i] = i;       //may include brief logical reasoning
}
```

- Disclosure: Though keeping the original digital format of the code (indention lengths, spaces, etc.) the use of Microsoft Word has reformatted the code in the document to be of different space and tab lengths. The font size has been reduced to help alleviate some visual misrepresentations.

5.2— Student Code

5.2.1— Assignment2.java

```
/**
 * CS 165a -- Artificial Intelligence
 * Lab 01
 * Assignment2.java
 * By: Hiram A Rios && Matthew S Montoya
 * Instructor: Dr. Chistopher Kiekintveld
 * TA: Khandoker Rahad
 * Purpose: To practice implementing BFS, DFS, and A* Search
 * Last Modified: March 2, 2019
 */

import java.io.*;          //For fileReader() & bufferedReader()
import java.util.*;        //For scanner when TA executes .jar file

public class Assignment2 {

       public static final long startTime = 0;
       public static int children_expanded = 0;        //Counter for the number of expanded
successor nodes
       public static int memory_nodes = 0;             //Counter for our nodes in memory
       public static int[] goal;
       public static int[] start;

       /**
        * GENERATE MAP SPACE
        * This method reads the test case text file
        * @return int[][] the 2D Map Space.
        */
       public static int[][] generate_map_space() {
              try{
            FileReader file_reader = new FileReader("/Users/hiramrios/Desktop/cs165a-
projectwork/test_case_files/test_case_10_10.txt");
            BufferedReader buffered_reader = new BufferedReader(file_reader);
            String current_line;

            //Get Map size
            current_line = buffered_reader.readLine();//gets the size of the map and
creates the matrix for it
            String[] string_split = current_line.split(" ");
                     int[][] map_space = new
int[Integer.parseInt(string_split[0])][Integer.parseInt(string_split[1])];

                     //Get Starting Position
                     current_line = buffered_reader.readLine();
                     string_split = current_line.split(" ");
            start = new int[]{Integer.parseInt(string_split[0]),
Integer.parseInt(string_split[1])};

            //Get Goal Position
            current_line = buffered_reader.readLine();
            string_split = current_line.split(" ");
            goal = new int[]{Integer.parseInt(string_split[0]),
Integer.parseInt(string_split[1])};
```

```
                     int row = 0;
                     System.out.println("Map Size: "+ map_space.length+ " " +
map_space[0].length);
                     System.out.println("Starting Position: " +start[0]+ " " +start[1]);
                     System.out.println("Goal Position: " +goal[0]+ " " +goal[1]);

                     //Create + Populate the Map Space
            while((current_line = buffered_reader.readLine()) != null) {
              string_split = current_line.split(" ");
                for(int column=0; column<string_split.length; column++) {
                     map_space[row][column] = Integer.parseInt(string_split[column]);
                }
                row++;
            }
            //Prevent Leakage + Return Map Space
              buffered_reader.close();
            return map_space;
        }
              catch (IOException e){
            System.out.println(e.getMessage());
            System.exit(0);
        }
        return null;
    }

       /**
        * PRINT SEARCH RESULTS
        * @param times_up
        * @param start_timer
        */
       public static void print_search_Results(boolean times_up, long start_timer)
{//prints results of search
              if(!(times_up)) {
                     System.out.println("\tNumber of Nodes Expanded:\t\t"
+children_expanded);
                     System.out.println("\tMaximum Number of Nodes held in Memory:\t" +
memory_nodes);
                     System.out.println("\tTime of Execution (in ms):\t\t" +
(System.currentTimeMillis() - start_timer));
              }

              //Our search timed out
              else {
                     System.out.println("Null");
                     System.out.println("\tCost of Path:\t-1");
                     System.out.println("\tMaximum Number of Nodes Held in Memory:\t"
+memory_nodes);
                     System.out.println("\tTime of Execution (in ms):\t\t" +
(System.currentTimeMillis() - start_timer));
              }
       }

       /**
        * SEARCH ALGORITHMS
        * @param int[][] map_space the 2D map we generated
        * This method calls the various search algorithms we're tasked with implementing.
```

```
	 */
	public static void search_algorithms(int[][] map_space) {
		breadth_first_search(map_space);
		iterative_deepening_search(map_space);
		a_star_search(map_space);
	}

	/**
	 * BREADTH-FIRST SEARCH
	 * @param int[][] map_space the 2D map we generated
	 * This method implements BFS on our tree from our text file.
	 */
	public static void breadth_first_search(int[][] map_space) {
		//Generate Timer
		long begin_timer = System.currentTimeMillis();
		long end_timer = 180000;
		boolean times_up = false;

		//Generate Goal Node
		Node goal_node = new Node(goal[0]-1, goal[1]-1);
		//Generate Starting Node
		Node start_node = new Node(start[0]-1, start[1]-1,
				manhattan_distance(start[0]-1, start[1]-1, goal_node),
				map_space[start[0]-1][start[1]-1], null);
		boolean[][] nodes_visited = new
boolean[map_space.length][map_space[0].length];
		Queue<Node> queue_nodes = new LinkedList<>();
		queue_nodes.add(start_node);

		while(!(queue_nodes.isEmpty())) {
			//Start the 3-Minute timer
			if((System.currentTimeMillis()-begin_timer) > end_timer) {
				System.out.println("3 minutes have passed. Your search time has
expired!");
				times_up = true;
				break;
			}

			//Search && Expand the current node
			Node current_node = queue_nodes.poll();
//Removes + Returns head of queue
			nodes_visited[current_node.x][current_node.y] = true;
//Call .x & .y position in Node class
			memory_nodes++;

			//Check if this is a goal node
			if(is_goal_node(current_node)){
				int path_cost = current_node.accumulated_path_cost;
				System.out.println("Path cost: "+path_cost);
				break;
			}

			//Generate Successor Nodes && add to the queue
			Node[] children_nodes_array = generate_successor_nodes(current_node,
goal_node, nodes_visited, map_space);
			for(int i=0; i<children_nodes_array.length; i++){
				queue_nodes.add(children_nodes_array[i]);
```

```
                        }

                        //Add Successor Nodes to memory
                        if(children_expanded<queue_nodes.size())
                                children_expanded = queue_nodes.size();
                }
                System.out.println("\nBREADTH-FIRST SEARCH--");
                print_search_Results(times_up, begin_timer);
        }

        /**
         * ITERATIVE DEEPENING SEARCH
         * @param int[][] map_space the 2D map we generated
         * This method implements IDS on our tree from our text file.
         */
        public static void iterative_deepening_search(int[][] map_space) {
                //Generate Timer
                long begin_timer = System.currentTimeMillis();
                long end_timer = 180000;
                boolean times_up = false;

                //Generate Goal Node
                Node goal_node = new Node(goal[0]-1, goal[1]-1);
                //Generate Starting Node
                Node start_node = new Node(start[0]-1, start[1]-1,
                manhattan_distance(start[0]-1, start[1]-1, goal_node),
                map_space[start[0]-1][start[1]-1], null);
                boolean[][] nodes_visited = new
boolean[map_space.length][map_space[0].length];
                int total = 0, limit = 0, infitite_count = 1;
                Stack<Node> the_fringe = new Stack<Node>();             //USE STACK AS A
FRINGE (FILO)


                for(limit = 0;limit<infitite_count; limit++){
                        //Reset variables from any previous search
                        the_fringe.push(start_node);
                        falsify_visited_nodes(nodes_visited);
                        children_expanded = 0;
                        memory_nodes = 0;

                        //Build the stack
                        while(!(the_fringe.isEmpty())) {
                                //Start the timer
                                if((System.currentTimeMillis()-begin_timer) > end_timer) {
                                        System.out.println("3 minutes have passed! Your search
time has expired!");

                                        times_up = true;
                                        break;
                                }
                                //Expand current node by popping from the fringe
                                Node current_node = the_fringe.pop();
                                children_expanded++;

                                //Check if this is a goal node
                                if(is_goal_node(current_node)) {
                                        int path_cost = current_node.accumulated_path_cost;
```

```
                                        //IDS Limit
                                        limit = infitite_count+1;
                                        System.out.println("\tPath cost:\t\t\t\t"+path_cost);
                                        break;
                                }

                                //Check if we've reached our limit
                                if(total==limit) {
                                        total++;
                                        the_fringe.removeAllElements();
                                        break;
                                }

                                //Generate Child Nodes && push to the fringe (stack)
                                Node[] children_nodes_array =
generate_successor_nodes(current_node, goal_node, nodes_visited, map_space);
                                for(int i = 0; i < children_nodes_array.length; i++){
                                        the_fringe.push(children_nodes_array[i]);
                                }

                                //Add Children for counting Nodes in Memory
                                if(memory_nodes < the_fringe.size())
                                        memory_nodes = the_fringe.size();
                        }
                        total++;
                        infitite_count++;
                }
                //Print Results
                System.out.println("\nITERATIVE DEEPENING DFS--");
                print_search_Results(times_up, begin_timer);
        }

        /**
         * A* SEARCH
         * @param int[][] map_space the 2D map we generated
         * This method implements A* Search on our tree from our text file.
         */
        public static void a_star_search(int[][] map_space) {
                //Generate Timer
                long begin_timer = System.currentTimeMillis();
                long end_timer = 180000;
                boolean times_up = false;

                //Generate Goal Node
                Node goal_node = new Node(goal[0] - 1, goal[1] - 1);
                //Generate Starting Node
                Node start_node = new Node(start[0] - 1, start[1] - 1,
                                manhattan_distance(start[0] - 1, start[1] - 1, goal_node),
                                map_space[start[0] - 1][start[1] - 1], null);
                boolean[][] nodes_visited = new
boolean[map_space.length][map_space[0].length];
                Queue<Node> queue_nodes = new
PriorityQueue<Node>(map_space.length*map_space[0].length, queue_comparator);
                queue_nodes.add(start_node);

                while(!(queue_nodes.isEmpty())){
                        //Start the timer
```

```
			if((System.currentTimeMillis() - begin_timer) > end_timer){//the 
search has timed out
				System.out.println("Search has timed out");
				times_up = true;
				break;
			}

			//Retrieve First Node in the Queue
			Node current_node = queue_nodes.poll();
			nodes_visited[current_node.x][current_node.y] = true;
			children_expanded++;
			int path_cost = current_node.accumulated_path_cost;

			//Check if this is a goal node
			if(is_goal_node(current_node)){//if goal is reached print information 
and break
				System.out.println("\tPath Cost:\t\t\t\t" +path_cost);
				break;
			}

			//Generate Child Nodes && push to the fringe (stack)
			Node[] children = generate_successor_nodes(current_node, goal_node, 
nodes_visited, map_space);//add children to queue
			for(int i = 0; i < children.length; i++)
				queue_nodes.add(children[i]);

			//Add Children for counting Nodes in Memory
			if(memory_nodes < queue_nodes.size())
				memory_nodes = queue_nodes.size();
		}
		//Print Results
		System.out.println("\nA* SEARCH--");
		print_search_Results(times_up, begin_timer);

	}

	public static void confirm_program_termination() {
		System.out.println("\nCongrats! Your program has finished executing!");
	}


	/**									HELPER METHODS FOR 
SEARCH ALGORITHMS
	*/

	//sets the visited matrix all to false
	/**
	 * SET VISITED NODES TO FALASE
	 * @param boolean[][] nodes_visited the nodes already explored
	 * Sets the visited nodes in 2D Map Space to false
	 */
	public static void falsify_visited_nodes(boolean[][] nodes_visited) {
		for(int i=0; i<nodes_visited.length; i++)
			for(int j=0; j<nodes_visited[i].length; j++)
				nodes_visited[i][j] = false;
	}
```

```
	/**
	 * GOAL NODE CHECKER
	 * @param Node current_node the node we're currently at in the tree
	 * @return boolean indicates if we've reached our goal node
	 */
	public static boolean is_goal_node(Node current_node) {
		if(current_node.distance == 0)
			return true;
		else
			return false;
	}

	/**
	 * MANHATTAN DISTANCE
	 * This method calculates and @return the Manhattan Distance.
	 * @param x the row in the matrix
	 * @param y the column in the matrix
	 * @param goal_node our goal node
	 */
	public static int manhattan_distance(int x, int y, Node goal_node) {
		return(Math.abs(goal_node.x - x) + Math.abs(goal_node.y - y));
	}

	/**	Comparator For A* Search Priority Queue	*/
	public static Comparator<Node> queue_comparator = new Comparator<Node>() {
		@Override
		public int compare(Node n1, Node n2) {
            return (int) (n1.accumulated_path_cost - n2.accumulated_path_cost);//using cost
so far and comparator
         }
	};

	/**
	 * GENERATE SUCCESSOR NODES
	 * @param node current the current node being traversed
	 * @param path_cost the cost of the path thus far
	 * This method prints the path & cost from the goal node to the start node
	 */
	public static Node[] generate_successor_nodes(Node current_node, Node goal_node,
boolean[][] nodes_visited, int[][] map_space) {

		//We can only move UP, DOWN, LEFT, & RIGHT
		LinkedList<Node> children_nodes_LL = new LinkedList <Node> ();

		//Move Right
		if(current_node.y + 1 < map_space[0].length &&
map_space[current_node.x][current_node.y + 1] != 0 &&
!nodes_visited[current_node.x][current_node.y + 1]){
			children_nodes_LL.add(new Node(current_node.x, current_node.y + 1,
manhattan_distance(current_node.x, current_node.y + 1, goal_node),
current_node.accumulated_path_cost+ map_space[current_node.x][current_node.y + 1],
current_node));
			nodes_visited[current_node.x][current_node.y + 1] = true;
		}
		//Move Left
		if(current_node.y - 1 >= 0 && map_space[current_node.x][current_node.y - 1]
!= 0 && !nodes_visited[current_node.x][current_node.y - 1]){
```

```
                    children_nodes_LL.add(new Node(current_node.x, current_node.y - 1,
manhattan_distance(current_node.x, current_node.y - 1, goal_node),
current_node.accumulated_path_cost + map_space[current_node.x][current_node.y - 1],
current_node));
                    nodes_visited[current_node.x][current_node.y - 1] = true;
              }
              //Move Up
              if(current_node.x - 1 >= 0 && map_space[current_node.x - 1][current_node.y]
!= 0 && !nodes_visited[current_node.x - 1][current_node.y]){
                    children_nodes_LL.add(new Node(current_node.x - 1, current_node.y,
manhattan_distance(current_node.x - 1, current_node.y, goal_node),
current_node.accumulated_path_cost + map_space[current_node.x - 1][current_node.y],
current_node));
                    nodes_visited[current_node.x - 1][current_node.y] = true;
              }
              //Move Down
              if(current_node.x + 1 < map_space.length && map_space[current_node.x +
1][current_node.y] != 0 && !nodes_visited[current_node.x + 1][current_node.y]){
                    children_nodes_LL.add(new Node(current_node.x + 1, current_node.y,
manhattan_distance(current_node.x + 1, current_node.y, goal_node),
current_node.accumulated_path_cost + map_space[current_node.x + 1][current_node.y],
current_node));
                    nodes_visited[current_node.x + 1][current_node.y] = true;
              }
              Node[] children = new Node[children_nodes_LL.size()];
              return children_nodes_LL.toArray(children);
       }

       /** MAIN RUNNER */
       public static void main(String[] args) {
              int[][] map_space = generate_map_space();
              search_algorithms(map_space);
              confirm_program_termination();
       }
}
```

5.2.2— Node.java

```
 /**
 * CS 165a -- Artificial Intelligence
 * Lab 01
 * Assignment2.java
 * By: Hiram A Rios && Matthew S Montoya
 * Instructor: Dr. Chistopher Kiekintveld
 * TA: Khandoker Rahad
 * Purpose: To practice implementing BFS, DFS, and A* Search
 * Last Modified: March 2, 2019
 */

public class Node {
    int x;                              //Position
    int y;
    int distance;                       //Distance to goal node
       int accumulated_path_cost;          //Cost from traversal (so far)
    Node prev;
```

```
    Node next;

    public Node(int x, int y){
        this.x = x;
        this.y = y;
              distance = 0;
              accumulated_path_cost = 0;
        prev = null;
        next = null;
    }


    public Node(int x, int y, int distance, int accumulated_path_cost, Node prev){
        this.x = x;
        this.y = y;
        this.distance = distance;
              this.accumulated_path_cost = accumulated_path_cost;
        this.prev = prev;
    }
}
```