Michael Della Donna
CS 4341 – Artificial Intelligence
Project 0

README

The project is divided into 3 packages. Land contains the data structure, search contains search algorithm implementations, and project0 contains files that run the test cases.

Data Structure

The data structure consists of three classes, Land.java, Cell.java, Point.java. A graphical model is located in appendix A. Land consists of a two dimensional array of Cell. Cells contain Points as well as descriptive properties. Cells enforce that they can only hold either a vehicle or an obstacle.

Search implementation

Depth First Search

- push start cell to a stack
- while the stack is not empty
    - pop a cell from the stack
    - check to see if it is the destination
        - if it is the destination
            - push it into another stack to return
            - while the current cell has a parent
                - push the parent onto the stack
                - set the current cell to its parent
            - return this new stack
        - if it is not the destination
            - place it into a blacklist
            - for each adjacent cell
                - if not occupied and not on the stack and not in the blacklist
                    - add them to the stack with the current cell as the parent
- if the stack is empty
    - no destination was found

Breadth First Search

- add start cell to a queue
- while the queue is not empty
    - queue a cell from the queue
    - check to see if it is the destination
        - if it is the destination
            - push it into a stack to return
            - while the current cell has a parent
                - push the parent onto the stack
                - set the current cell to its parent
            - return this new stack
        - if it is not the destination
            - place it into a blacklist
            - for each adjacent cell
                - if not occupied and not in the queue and not in the blacklist
                    - add them to the queue with the current cell as the parent

- if the queue is empty
    - no destination was found

Uniform Cost Search

- add start cell to a priority queue ordered by cost, using 0 as priority for the start
- while the queue is not empty
    - queue a cell from the queue
    - check to see if it is the destination
        - if it is the destination
            - push it into a stack to return
            - while the current cell has a parent
                - push the parent onto the stack
                - set the current cell to its parent
            - return this new stack
        - if it is not the destination
            - place it into a blacklist
            - for each adjacent cell
                - if not occupied and not in the blacklist
                    - if they are in the stack
                        - if they have a higher cost, replace them
                    - If they are not in the stack
                        - add them to the queue with the current cell as the parent, ordered by the cost from the current cell into this new cell
- if the queue is empty
    - no destination was found

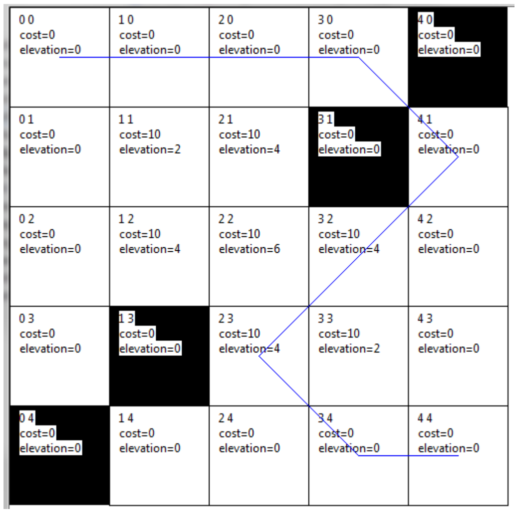
Test Cases

<u>First test case</u>

Depth First Search

```
160 steps executed
14 cells visited
Total cost of this path: 32.0
```

Required the least number of instructions to execute
Visited the number of cells
Produced an indirect path



0 0
cost=0
elevation=0
	1 0
	cost=0
	elevation=0
		2 0
		cost=0
		elevation=0
			3 0
			cost=0
			elevation=0
				4 0
				cost=0
				elevation=0

0 1
cost=0
elevation=0
	1 1
	cost=10
	elevation=2
		2 1
		cost=10
		elevation=4
			3 1
			cost=0
			elevation=0
				4 1
				cost=0
				elevation=0

0 2
cost=0
elevation=0
	1 2
	cost=10
	elevation=4
		2 2
		cost=10
		elevation=6
			3 2
			cost=10
			elevation=4
				4 2
				cost=0
				elevation=0

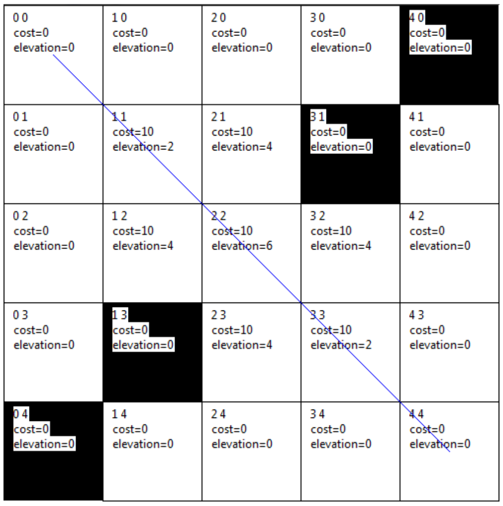
0 3
cost=0
elevation=0
	1 3
	cost=0
	elevation=0
		2 3
		cost=10
		elevation=4
			3 3
			cost=10
			elevation=2
				4 3
				cost=0
				elevation=0

0 4
cost=0
elevation=0
	1 4
	cost=0
	elevation=0
		2 4
		cost=0
		elevation=0
			3 4
			cost=0
			elevation=0
				4 4
				cost=0
				elevation=0

Breadth First Search

```
211 steps executed
18 cells visited
Total cost of this path: 39.0
```

Visited the most cells
Required a medium number of instructions to execute
Produced the shortest, most direct path



0 0
cost=0
elevation=0
	1 0
	cost=0
	elevation=0
		2 0
		cost=0
		elevation=0
			3 0
			cost=0
			elevation=0
				4 0
				cost=0
				elevation=0

0 1
cost=0
elevation=0
	1 1
	cost=10
	elevation=2
		2 1
		cost=10
		elevation=4
			3 1
			cost=0
			elevation=0
				4 1
				cost=0
				elevation=0

0 2
cost=0
elevation=0
	1 2
	cost=10
	elevation=4
		2 2
		cost=10
		elevation=6
			3 2
			cost=10
			elevation=4
				4 2
				cost=0
				elevation=0

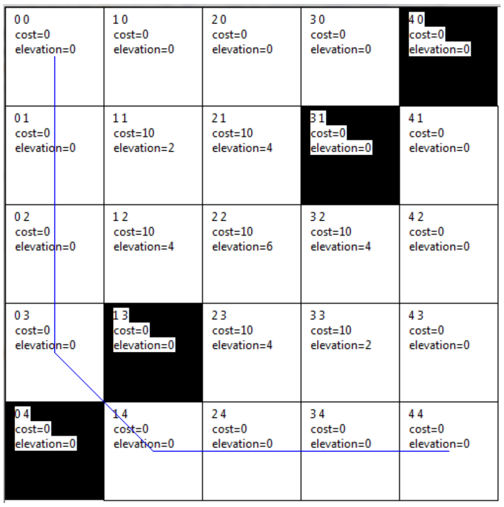
0 3
cost=0
elevation=0
	1 3
	cost=0
	elevation=0
		2 3
		cost=10
		elevation=4
			3 3
			cost=10
			elevation=2
				4 3
				cost=0
				elevation=0

0 4
cost=0
elevation=0
	1 4
	cost=0
	elevation=0
		2 4
		cost=0
		elevation=0
			3 4
			cost=0
			elevation=0
				4 4
				cost=0
				elevation=0

Uniform Cost Search

```
271 steps executed
15 cells visited
Total cost of this path: 8.0
```

Required the most instructions to execute
Only visited one more cell vs DFS
Produced the least costly path by far



0 0
cost=0
elevation=0
	1 0
	cost=0
	elevation=0
		2 0
		cost=0
		elevation=0
			3 0
			cost=0
			elevation=0
				4 0
				cost=0
				elevation=0

0 1
cost=0
elevation=0
	1 1
	cost=10
	elevation=2
		2 1
		cost=10
		elevation=4
			3 1
			cost=0
			elevation=0
				4 1
				cost=0
				elevation=0

0 2
cost=0
elevation=0
	1 2
	cost=10
	elevation=4
		2 2
		cost=10
		elevation=6
			3 2
			cost=10
			elevation=4
				4 2
				cost=0
				elevation=0

0 3
cost=0
elevation=0
	1 3
	cost=0
	elevation=0
		2 3
		cost=10
		elevation=4
			3 3
			cost=10
			elevation=2
				4 3
				cost=0
				elevation=0

0 4
cost=0
elevation=0
	1 4
	cost=0
	elevation=0
		2 4
		cost=0
		elevation=0
			3 4
			cost=0
			elevation=0
				4 4
				cost=0
				elevation=0

Second Test Case

Consisted of a 5x5 grid of cells, start at (0,0), destination at (4,4).
Obstacles at (3,4) (4,3) (4,4)

Depth First Search	Breadth First Search	Uniform Cost Search
249 steps executed 22 cells visited No destination found	249 steps executed 22 cells visited No destination found	485 steps executed 22 cells visited No destination found

In this scenario there was no destination to find, it was inaccessible. By looking at both DFS and BFS we can see that they visited every other cell and took the same amount of steps to reach the conclusion that no path existed. UCS reached the same conclusion, but require more steps (work) to give that answer. The extra steps come from the periodic refactoring of the parents of each node. While DFS and BFS simple drop new nodes into a queue or stack, UCS attempts to up update nodes that have already been discovered with new parents if they represent a shorter path.

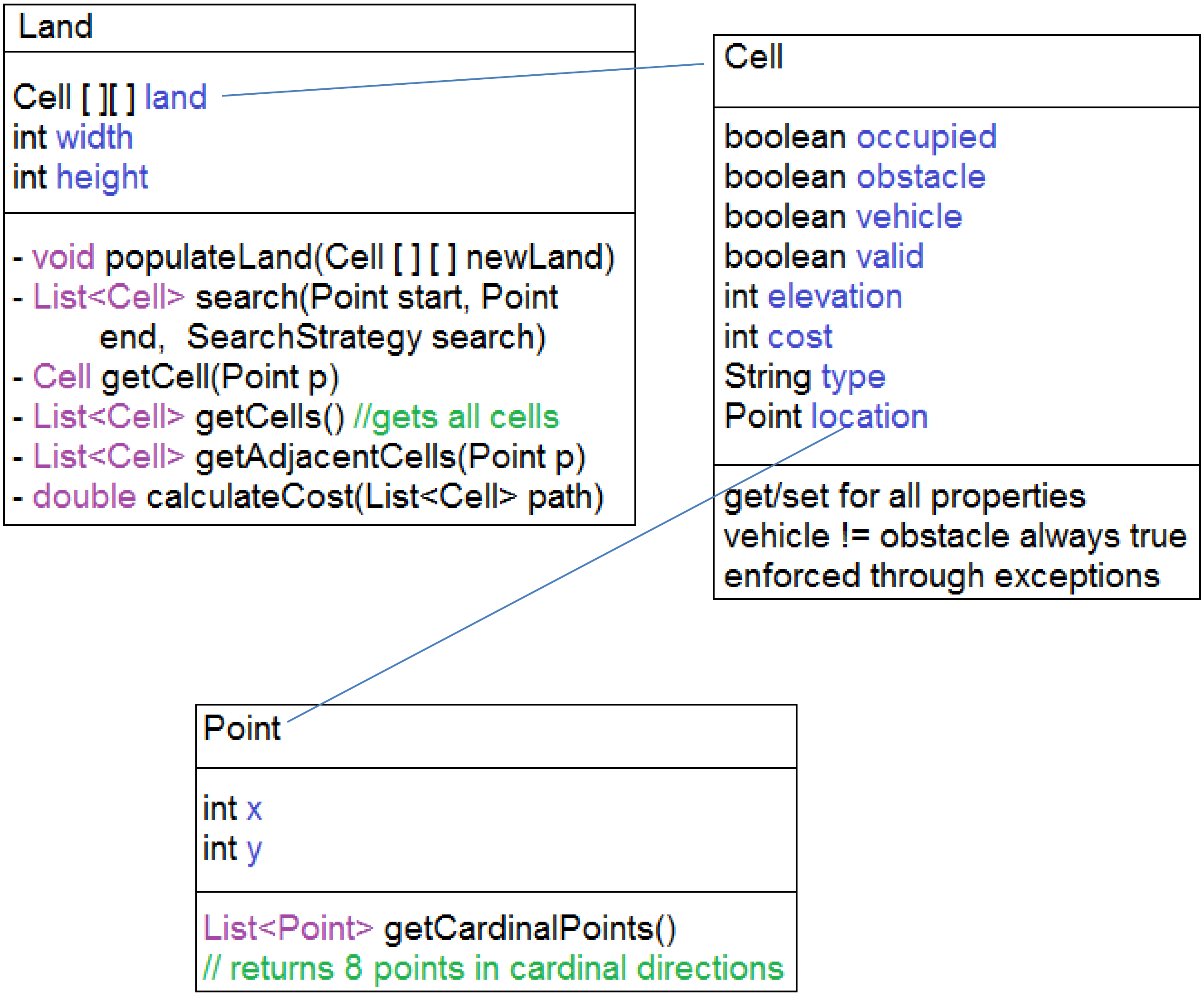
Third Test Case

Consisted of 4x4 grid of cells, start at (0,0), destination at (0,1)

Depth First Search	Breadth First Search	Uniform Cost Search
165 steps executed 16 cells visited Total cost of this path: 2.0	43 steps executed 4 cells visited Total cost of this path: 2.0	21 steps executed 3 cells visited Total cost of this path: 2.0

While all three search algorithms correctly identified the same path from start to destination, Depth First Search visited every cell in the structure, while BFS and UCS only need to visit 4 cells and 3 cells respectively. The worst case for DFS is when the destination is the first thing to be pushed to the stack, as it will be the last to be removed

Appendix A


Land
Cell [ ][ ] land
int width
int height
- void populateLand(Cell [ ] [ ] newLand)
- List<Cell> search(Point start, Point end, SearchStrategy search)
- Cell getCell(Point p)
- List<Cell> getCells() //gets all cells
- List<Cell> getAdjacentCells(Point p)
- double calculateCost(List<Cell> path)
Cell
boolean occupied
boolean obstacle
boolean vehicle
boolean valid
int elevation
int cost
String type
Point location
get/set for all properties
vehicle != obstacle always true
enforced through exceptions
Point
int x
int y
List<Point> getCardinalPoints()
// returns 8 points in cardinal directions