Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name:Bodea Razvan-Marius
Group:30432
Email:razvan.bodea2700@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search

1.1 Introduction

A short video with the visualization of the algorithms on the maps can be seen here:
(https://youtu.be/td4RjsOg280)
Problem:
The proposed problem was designing difficult grids/maps for the algorithms we worked with and analyzing/comparing the performance of the algorithms on those maps.

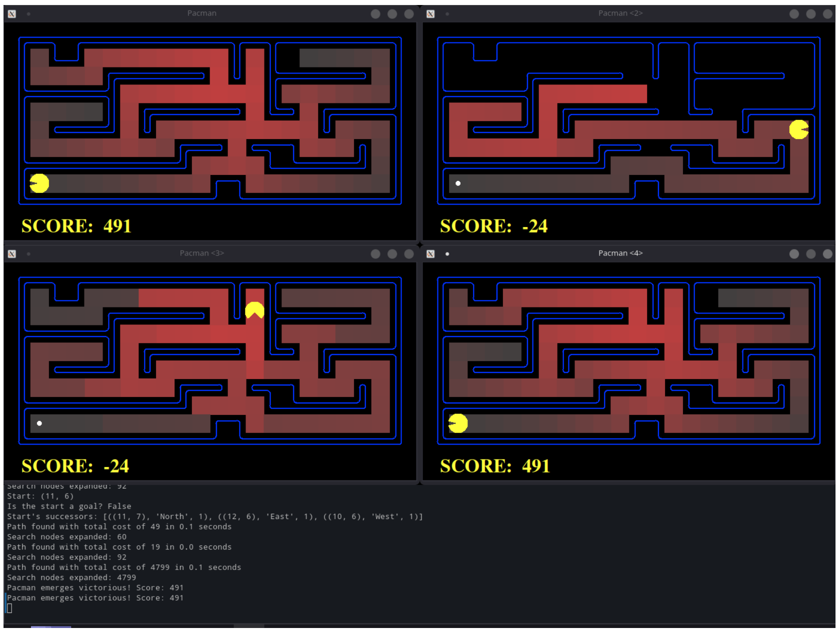
1.2 Approach

The approach was split in 2 parts:

One was creating by hand some grids that are difficult for the algorithms, based on the knowledge I have about them and also based on the implementation of them in the program (ex: the order in which the algorithms prioritize the movement is important: the priority North-Left-Down-Right != Down-Left...)

The other part was creating a simple code to generate some random maps to test the algorithms on. The maps should be as random as possible, but keep some aspects the same (like starting as close to the top right corner and having to go somewhere on the bottom left corner)

I also introduced a simple random search algorithm to have a comparison between algorithms that try to solve the problem as good as possible and just moving randomly until you find the solution. To make the process more fun, I started the algorithms in parallel on the same map to see which one would finish first, organizing a competition between them.

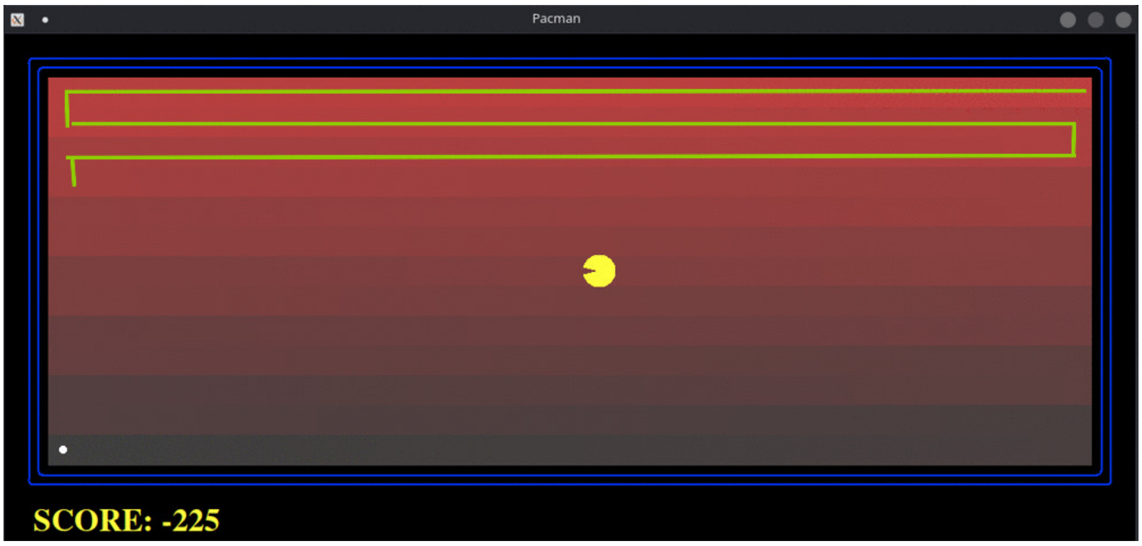


```
Search nodes expanded: 92
Start: (11, 6)
Is the start a goal? False
Start's successors: [((11, 7), 'North', 1), ((12, 6), 'East', 1), ((10, 6), 'West', 1)]
Path found with total cost of 49 in 0.1 seconds
Search nodes expanded: 60
Path found with total cost of 19 in 0.0 seconds
Search nodes expanded: 92
Path found with total cost of 4799 in 0.1 seconds
Search nodes expanded: 4799
Pacman emerges victorious! Score: 491
Pacman emerges victorious! Score: 491
```

1.3 The new maps

Each search algorithm has it's strengths and weaknesses. The goal of the created maps is to find/exploit those aspects in order to test how the algorithms perform in every situation.

1.3.1 Map1: DifficultDFS

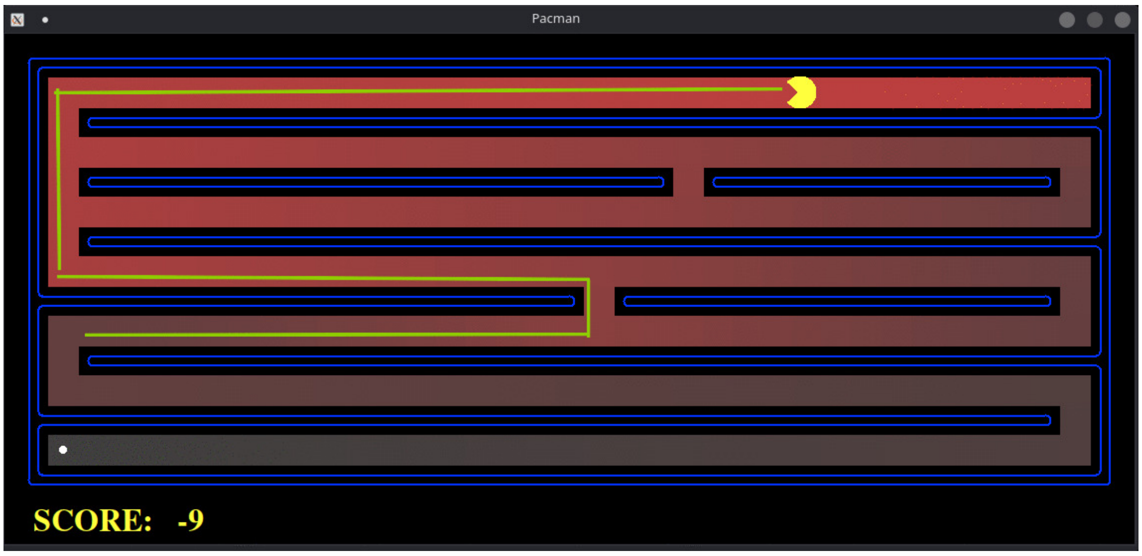


As weird as it looks, the map that DFS takes the longest to solve, is a simple map with just the outside walls. The starting position should be top right and finish should be bottom left (this is because of the previously mentioned order of prioritization of the movement directions). This is a good example of : more options do not necessarily lead to a better/faster solution.

DFS will choose the longest path on this map due to the way it works: it will always choose the 1st option it has thus creating the longest path to the end.

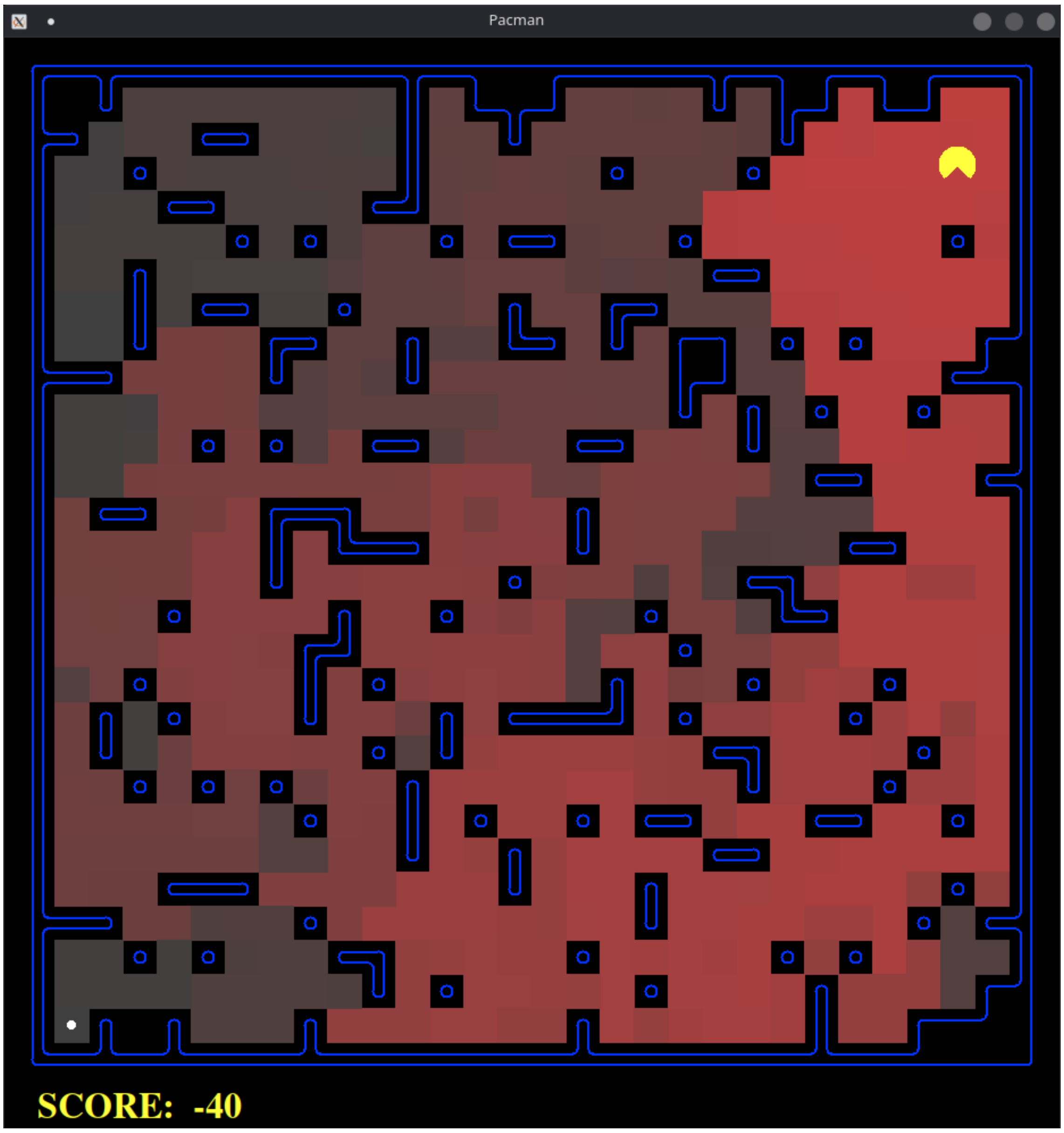
BFS will go straight down then left, finding the best solution available without diagonal walking allowed. On this map, BFS will expand a large number of nodes in order to find the best solution (a lot of computation).

1.3.2 Map2: DifficultBFS



This map is difficult for BFS in terms of time, but the number of expanded nodes is lower than the previous map. Because BFS fainds the optimal solution, the only way to delay it is giving it a long path with no choices and increasing the 'moving' time, or giving it a map with a lot of nodes to expand, thus increasing the computation time.

1.3.3 Map3: RandomlyGenerated



This map was randomly generated by using the algorithm attached at the end. The map is 30x30.

Testing the algorithms on random maps is important because these tests could find defects or qualities related to the algorithms. Randomness creates a lot of interactions that a hard-coded map could not include. One aspect I tried to keep constant is the start and end point for the pacman. The starting position and end point are placed as close as possible to the original problem parmeters.

Even on the randomly generated grids, the BFS algorithm still finds the optimal solution.

1.4 Results of the tests

1.5 Conclusions

DFS algorithm can find a good enough solution for a given problem, with fewer computations on average than other algorithms.

Search nodes expanded	DFS	BFS	A*	Random
DifficultDFS	454	454	454	9200
DifficultBFS	250	253	253	80792
RandomMap	134	608	608	9900

Cost of found solution	DFS	BFS	A*	Random
DifficultDFS	454	46	36	9200
DifficultBFS	250	150	150	80792
RandomMap	98	54	54	9900

BFS algorithm finds the optimal solution but it expands more nodes to find this solution. I would default to using this algorithm unless the problem is huge and it just needs a solution.

Random search algorithm is just a for fun algorithm that should not be used in practice. It was important for my tests because it shows how hard a search problem actually is, if you don't have a plan to solve it and just go by chance.

Appendix A

Your original code

```python
import random

maxX = 15
maxY = 15

# write mode just to clear the file at the start
file1 = open("newGrid.lay", 'w')
open("newGrid.lay", 'a')

mapMatrix = [[0] * maxX for i in range(maxY)]
choiceList = [1, 2, 3, 4]

# generating random locations for the walls
# this generation will be the starting grownd for modifications
for x in range(maxX):
    for y in range(maxY):
        if x == 0 or y == 0 or x == maxX - 1 or y == maxY - 1:
            mapMatrix[x][y] = 1
        else:
            k = random.choice(choiceList)
            if k == 1:
                mapMatrix[x][y] = 1
            else:
                mapMatrix[x][y] = 0

# making sure the map can be played and every place is reachable
# by removing walls that would block the path
for x in range(maxX - 1):
    for y in range(maxY - 1):
        # a space surrounded by 3 walls
        if (mapMatrix[x - 1][y] == 1 and mapMatrix[x + 1][y] == 1 and mapMatrix[
    x][y - 1] == 1) or (
                mapMatrix[x + 1][y] == 1 and mapMatrix[x][y - 1] == 1 and
    mapMatrix[x][y + 1] == 1) or (
                mapMatrix[x - 1][y] == 1 and mapMatrix[x + 1][y] == 1 and
    mapMatrix[x][y + 1] == 1):
            if x != 0 and x + 1 != maxX - 1 and y != 0 and y + 1 != maxY - 1:
                choice = random.choice(choiceList)
                if choice == 1:
                    if x-1 != 0:
                        mapMatrix[x - 1][y] = 0
                if choice == 2:
                    if x + 1 != maxX-1:
                        mapMatrix[x + 1][y] = 0
                if choice == 3:
                    if y-1 != 0:
                        mapMatrix[x][y - 1] = 0
                if choice == 4:
                    if y+1 != maxY-1:
                        mapMatrix[x][y + 1] = 0

for x in range(maxY - 1, 1, -1):
    if mapMatrix[1][x] == 0:
        mapMatrix[1][x] = 2
        break
for y in range(1, maxY - 1, 1):
    if mapMatrix[maxX-2][y] == 0:
        mapMatrix[maxX-2][y] = 3
        break

# writing the matrix in a file as a format that can be understood by the pacman program
ok = 0
for x in range(maxX):
    for y in range(maxY):
        if mapMatrix[x][y] == 1:
            file1.write('%')
        if mapMatrix[x][y] == 0:
            file1.write(' ')
        if mapMatrix[x][y] == 2:
            file1.write('P')
        if mapMatrix[x][y] == 3:
            file1.write('.')
    file1.write('\n')
```

Listing A.1: Generate a random map

```
def randomSearch(problem):
    import random
    current = problem.getStartState()
    path = []
    while not problem.isGoalState(current):
        nextNode = random.choice(problem.getSuccessors(current))
        path.append(nextNode[1])
        current = nextNode[0]
    return path
```

Listing A.2: Random search algorithm