Department of Computer Science
Technical University of Cluj-Napoca

Introduction to Artificial Intelligence

Laboratory activity 2019-2020

Project title: Heuristic search
Tool: Pacman agent

Name: Mateiu Bianca
Group: 30434
Email: mateiu.bianca@gmail.com

Contents

Chapter 1

Installing the tool (W_2)

The tool I have chosen to work with is `Pacman agent`, a tool developed by the Berkeley University, for their artificial intelligence course.[4] Also, the operating system I have worked with is Windows 10, so the following instructions for installing the tool apply to this particular os.

Prior to installing the tool, one has to make sure the computer has the following prerequisites installed and working properly.

Firstly, verify that Python 2.7 is installed on the computer. It is important that the version is 2.7, because the tool is not compatible with any newer versions of python. To check the version simply run the Windows PowerShell, and type the command: `python --version`. If you do not have python installed, simply go to their website: www.python.org/downloads/, and search for a release version starting with 2.7. The next step is to download either the *Windows x86-64 MSI installer* or the *Windows x86 MSI installer*. After the download is complete, run the installer and follow the instructions it provides.

Secondly, it would be very helpful to have a dedicated IDE for python programming, or even just a code editor. Some good options are *Atom, Visual Studio Code, Eclipse*. The IDE I used is *PyCharm*, therefore I will give indications on using the tool in this environment. Of course, this step is not mandatory, since python files can be ran using the Windows command line, but int the case of projects having a large number of files, this dedicated IDE's make it easier to keep track of your work.

After making sure that this step is completed, we can proceed to installing the tool. In order to do this, one has to follow a few simple steps:

1. Go to Berekley's CS188 Introduction to AI website
2. Download the *search.zip* archive containing the source code and supporting files
3. Extract the files to the preferred working directory
4. Launch PyCharm and go to `File -> Open`, and locate the directory you extracted the archive *search.zip* to
5. In order to run the tool, we need to configure the project interpreter to Python 2.7, that you have already installed. For this, go to `File -> Settings`, and in the newly opened window, click on `Project -> Project Interpreter`. The next step is to choose from the dropdown list Python 2.7, or click on *Show more...* if that option is not on the list. If you still can not find this option, than you must add a new Project Interpreter: click on the + sign, and locate the directory of the Python 2.7 executable file.

After following this steps, you should be able to use the tool: click right on the *pacman.py* file and run it. A new window should open up, with the classical game of Pacman.

Chapter 2

Running and understanding examples (W_3)

At a closer inspection of the files in this project, we will find a file called *commands.txt* that contains a list of parameters that can be used to run the pacman program. This tool has many options, that can be studied by running the program with the *-h* parameter. In order to supply this parameters to the program in PyCharm, we have to go to `Run -> Edit Configuration`, and type them in the `Parameters` input box.

The most options we are going to use most often are:

- *-l* or *-layout* followed by the name of a file found in the *layouts* directory. This changes Pacman's environment, more precisely the maze.
- *-p* or *-pacman* followed by the name of the agent chooses the agent type that will be used.
- *-a* or *-agentArgs* followed by comma sepparated values represents the arguments that will be sent to the agent. This are often functions or heuristic to be used by the agent.
- *-q* shuts down the graphical interface part of the tool

Before we continue, we should clarify the terms *agent* and *environment* and their meaning in the world of Artificial Intelligence. According to AIMA, "An **agent** is anything that can be viewed as perceiving its **environment** through sensors and acting upon that environment through actuators"[3]. Therefore, our pacman is in fact an agent, and its environment is the maze he finds himself into.

Next, I will exemplify some of the capabilities of this tool:

2.1 Classic pacman

If we simply run the *pacman.py* file, without any configuration, the classical game of Pacman will appear in a new window. You can even play the game: control it using the arrow keys. Figure 2.1 shows a snapshot of the game.

In this example, Pacman is a Keyboard Agent. The implementation of this agent can be found in the file *keyboardAgents*. Other agents are implemented in *searchAgents.py*.

2.2 West Agent

By using the configuration:

```
--layout testMaze --pacman GoWestAgent
```

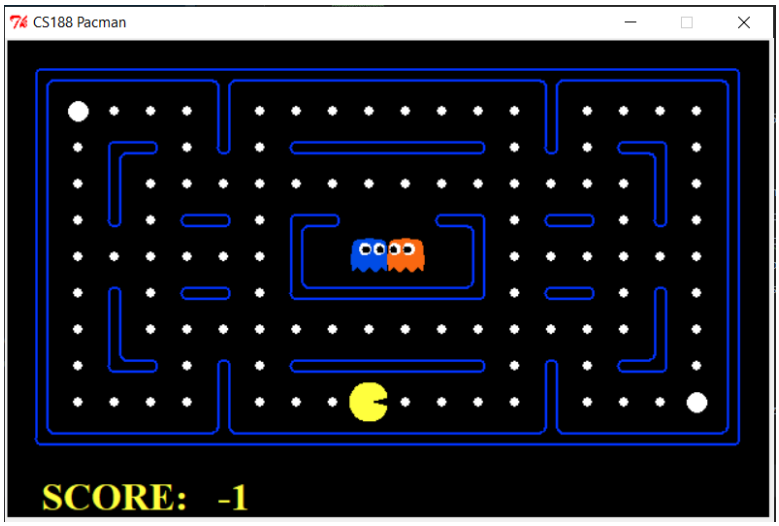
we can see how another Pacman agent behaves. This is a fairly simple agent, and all it does is go west. See 2.1. Of course, when we are dealing with more imbricated layouts that require turning to get to the goal, this agent will fail.**??**

2.3 Search Agent

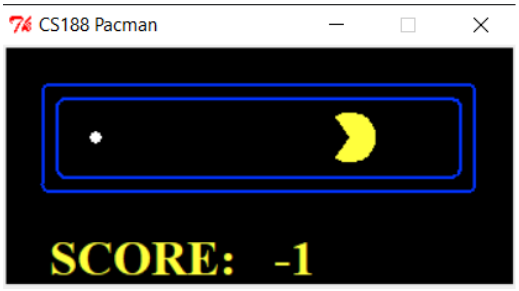
In order to use this tool to its full potential, we have no interest in using an agent that we have to control, or that is capable of one thing only, as we did before. That is why in this example we will be using the Search Agent, that is fully implemented for us and can be found in the file *searchAgents.py*. For this, you will be using the following configuration:

```
-l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

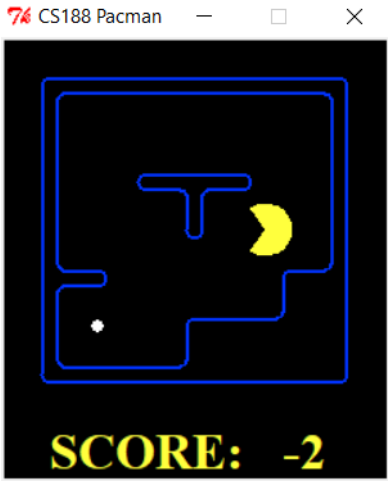
You will notice that this search agent also receives a function that it will use in navigating the maze: *tinyMazeSearch* (found in *search.py*). This function simply hardcodes the solution for one particular maze. This function describes the behaviour of the agent. (See 2.1)



(a) A game of Pacman.



(b) West Agent.



(c) Search Agent.

Figure 2.1: Different Pacman configurations

Chapter 3

Understanding conceptual instrumentation (W_4)

The tool has many functioning modes, that can be controlled by the means of the sent parameters. For example, you could have training games for Pacman, or record the results.

The next algorithm describes the main part of the tool, the runGames function:

Algorithm 1: Run Games.

Input: *layout* - an object containing information about the maze, such as the position of walls, food and agents;
pacman - the search agent;
ghosts - a list of ghost agents;
display - an object containing information about the graphicla interface of the game;
numGames - the number of games to play;
record - save game data to a file;
numTraining - the number of episodes that are training(surpess output);
catchException - turns on exception handling during games;
timeout - maximum time an agent can spend computing in a game;
Output: *games* - a list of all the ran games

```
games ← empty list
for i ← 2 to numGames do
    beQuiet ← (i ≥ numTraining)
    if beQuiet then
        surpress output and graphics
    else
        gameDisplay ← display
    game ← newGame()
    game.run()
    if ! beQuiet then
        games ← games + game
    if record then
        save data to file
if numGames - numTraining ≥ 0 then
    print statistics of the games
return games
```

Chapter 4

Project description (W_5)

4.1 Narrative description

I plan on using this tool to experiment with different searching algorithms, and see how different algorithms behave in terms of optimality and time complexity with a given problem. The problem is the following: start from an initial position in a maze and reach a goal position in the minimum number of steps, and with the minimum number of explored states (therefore, in the smallest amount of time).

The search problem has been extensively discussed before, and dozens of algorithms have been developed to solve it. Also, it can be easily translated into real life problems: finding a route from a town to another, planning a trip by taking into consideration transportation, finding a path through a maze (like in Pacman), finding solutions for games that have a finite set of states, transitions and one single goal (such as eight puzzle). More that that, the pathfinding problem is one of the most popular AI problem in the rising industry of computer games, where often this problem has to be solved in real time, and with limited resources of memory and CPU resources. [**?**]

Therefore, I will implement a number of searching algorithms I find suitable for the task, use the tool to run them multiple times on randomly generated mazes of various sizes, store the data, then query and manipulate it and record my findings.

4.2 Facts

I will consider two searching strategies:

1. uninformed search
2. informed search

I will start with uniformed search first. This are algorithms that only use the information provided to them by the problem. They can distinguish a non-goal state from a goal state, but they can not tell if one state is closer to the goal than another, therefore if that state should be prioritized. Whereas, informed search algoritms can, and use heuristics an priority queues in order to do that. [3]

The uninformed search algorithms that I will take into consideration are *Breadth First Search, Depth First Search, Uniform Cost Search, Bidirectional Search.* The informed search algorithms that I will take into consideration are *Astar* and *Dynamic Weighting Astar.* Also, as an heuristic for the informed search, I will be using the classical Manhattan distance, and I will also develop my own heuristic, appropriate for the maze problem.

4.3 Specifications

In my approach of using the Pacman tool, the input parameters will be the following ones: a Position search problem, and a function that is to be used to solve that problem. There will be a number of functions, some using heuristics, some not. The relevant information that can be found in the problem is:

- a list containing information about the grid's cells, which contain a wall and which don't. This information will be read and processed from a layout file
- the staring position in the grid
- the goal
- the number of explored states, or, in other words, the number of visited cells

In regards to the output of the program, we will not be concerned with the grapihical capabilities of the tool, so we will only take into consideration the information about the performance of the searching algorithm used, namely the cost of the found solution and the number of expanded nodes. This information will be stored in a .csv file, and manipulated later on, after the program has been ran on a large enough set of input data (layouts and functions) so that the result will have credibility.

Moreover, another objective I have set is to learn using another tool that will help me manipulate the data and transform it into charts, so that the conclusion is easier to draw. I will be using *Jupyter Notebook* for this, along with the libraries *Pandas* and *Matplotlib*.

4.4 Knowledge acquisition

How do represent knowledge? The knowledge my system requires is a simple layout file, conatining a perfect maze, represented by symbols such as '*%*' for walls and spaces for hallways. The initial position and goal also have to be represented: with '*P*' and '.'. Of course, in order to see which algorithms find the optimal solution (minimum cost) and which don't, there should be more than one path between start and destination.

Where are you getting the required knowledge/data I used an algorithm[7] that creates a perfect maze[1], and which is based on the Depth first search algorithm. I adjusted the algorithm in order to obtain a layout compatible with the Pacman tool, and made the maze not be a perfect one, since we do not want that.

4.5 Related work

There are hundreds of papers that can be found on the subject of search problems. The one source of inspiration and knowledge that I used more than others is the book *Artificial Intelligence A Modern Approach*[3]

[1] "A so called 'perfect' maze has every path connected to every other path, so there are no unreachable areas. Also, there are no path loops or isolated walls. There is always one unique path between any two points in the maze." [5]

4.6 Assumptions

My assumptions are that informed search will prove to be quicker to reach the goal first, but that this will sometimes happen at the expense of not always finding the optimal solution.

Chapter 5

Implementation details (W_9)

As I have mentioned before, my approach to this project is a rather experimental one. I focused on researching the strong and weak point of informed and uninformed search strategies, and as a result, this project does not model on a particular real life situation. Despite this, the route finding problem is something people face every day, and my findings can be easily translated in a real life scenario.

This being said, since I am using the Pacman tool, my project is based on developing search algorithms that are meant to help Pacman find its way through the maze.

5.1 Relevant code

5.1.1 Searching algorithms

The algorithms I implemented in this project can be divided into two categories:

1. uninformed search:
    - Breadth First Search
    - Depth First Search
    - Uniform Cost Search
    - Bidirectional Search
2. informed search:
    - Astar
    - Dynamic Weighting Astar

I have also created a *Node* structure, where I store the information about each state the Search Agent finds himself into. This structure has a general implementation, so that it can be used in other problems, not just the *Position search problem*. The *Node* class has the following fields:

- state - the current state
- cost - the cost of getting to this state
- action - a list of the necessary actions to get to this state
- parent - the Parent node

- depth - the length of the path from the current node to the starting node

The *Node* class also has a method, *getNeighbors*, that uses the problem to generate the list of states that can be reached from the current state

I will now briefly explain every algorithm:

Breadth First Search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The data structure used for the frontier is the Queue (LIFO). [3]

Depth First Search always expands the *deepest* node in the frontier. The data structure used for the frontier is the Stack (FIFO). [3]

Uniform Cost Search is similar to the BFS algorithm, but instead of using a queue to store the frontier nodes, UCS uses a priority queue, to make sure that the nodes with the lowest path cost are expanded first.

Bidirectional Search can be used for problems where we know both the initial position and the goal. The idea is to run two BFS algorithms, one from the initial state and one from the goal, and build the solution when the frontier of this 2 algorithms intersect. [3]

Astar is a form of BFS, but the evaluating function of the nodes is different:

$$f(n) = g(n) + h(n)$$

where g(n) is the cost to reach the node, and h(n) is the cost to get to the goal. [3] H(n) is an heuristic function, and depending on this function, we can improve this algorithm even further.

Dynamic Weighting Astar is a form of the Astar algorithm that uses dynamic weights for the heuristic function. The idea in this algorithm is that at the beginning, it is important to get anywhere fast, so the heuristic, or the approximated cost to get to the goal is not as important. But as the nodes increase in depth, the heuristic becomes more and more important. So, the function evaluating the nodes becomes:

$$f(n) = g(n) + (1 + \sigma * w(n)) * h(n),$$

where

$$w(n) = \begin{cases} 1 - \frac{d(n)}{N}, & \text{if } d(n) \leq N \\ 0, & \text{otherwise} \end{cases}$$

and $d(n)$ is the depth of the search and N is the anticipated length of the solution path [6]

I used two different heuristics for the informed search strategies: the Manhattan Distance, and an heuristic developed by me, that combines the Manhattan Distance value with the number of walls found between the current state and the goal state. I tried this heuristic in multiple forms, by changing the weight of the walls number.

5.1.2 Maze generation

Another algorithm I used is the one that randomly generates the maze layouts, and it is based on the Depth first search algorithm.[7]

The idea is to start with a grid of cells having all the walls between them. Then choose a random location from where to start the DFS algorithm. For each cell, mark it as visited, choose a random neighbor, and if it is not visited, remove the adjacent wall and continue the DFS with this node.

This algorithm generates a perfect maze (a maze in which for each 2 cells, there is a single path between them), but since we need to have more than one possible path from Pacman to the goal, we choose a number of random walls from the ones left and remove them.

Lastly, the string of the generated maze is written to a file.

5.1.3 Generating graphs

And last but not least, I used the online environment *Jupyter Notebook*, and the libraries *pandas* and *matplotlib* to interpret the results.

I configured the Pacman tool to run all the search algorithms on the same maze, and write the results in a .csv file. Then, in Jupyter Notebook, i used *Pandas* to parse the data and save it into a structure called *Data frame*, that enabled me to easily manipulate it. The next step was to use *matplotlib*, a plotting library for python, to generate line graphs and bar graphs.

To give you an example of the capabilities of this tool, the graph in figure 5.1 was generated by the following piece of code:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv("C:/Users/Bianca/Desktop/resultsSmall.csv")

df = pd.DataFrame(data)
barAll = pd.DataFrame(data, columns=['Function', 'Cost', 'Expanded'])
                .groupby('Function')
                .sum()
                .sort_values(by = ['Expanded', 'Cost'], ascending=True)

barAll.plot(kind='barh')
```

In the above code, I used pandas to create the data frame *df*. Now, we have to manipulate this data frame in order to have the desired structure to display the graph. So, in the variable *barAll*, I selected only the relevant collumns: *Function, Code* and *Expanded*. Using the function *groupby* I grouped the data by the collumn *Function*, and the values in the rest of the collumns are summed, and then sorted ascendingly by the *Expanded* collumn and the *Cost* collumn. After all this has been done, the data can be plotted.

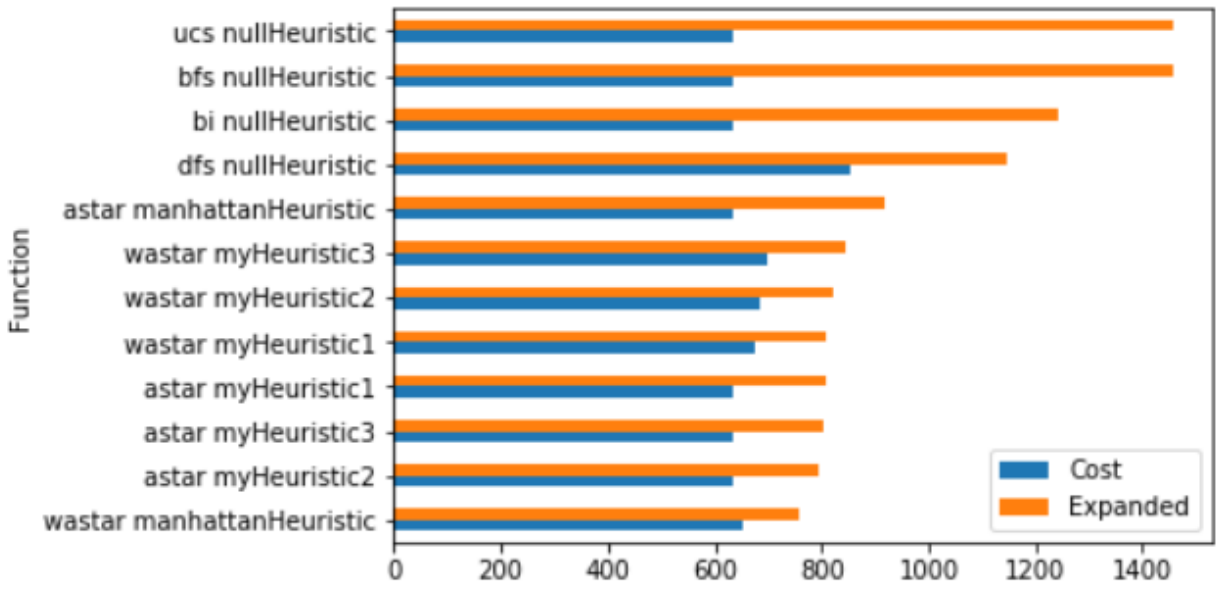


Figure 5.1: The results registered by the program ran on small sized mazes

Chapter 6

Tool expressivity (W_{10})

This tool can be used for more than just for a Pacman game. It gives the user the possibility to use it with diferent Search Agents and different Search Problems. So essentially, if a route finding problem can be described in terms of a Search Problem, than it can be solved by this tool.

The game is the main part of this tool. There are also some *GameRules*, where the definitions of winning and losing can be found. The game also defines the layout of the maze, the Search Agent, the Ghosts Agents, and other parameters meant to configure the game.

The Search Agent receives a search algorithm and a search problem, and returns actions to follow for a path from an initial state and a goal (defined in a problem)

The Search Problem defines the state space, start state, goal test, successor function and cost function. For example, for the Pacman game, the state space consists of (x, y) positions in a pacman game. For the eightpuzzle, the states are defined by different arrangements of the 8 digits.

The Ghost Agent is another type of Agent, but instead of searching a path to a specific goal, it simply roams the maze.

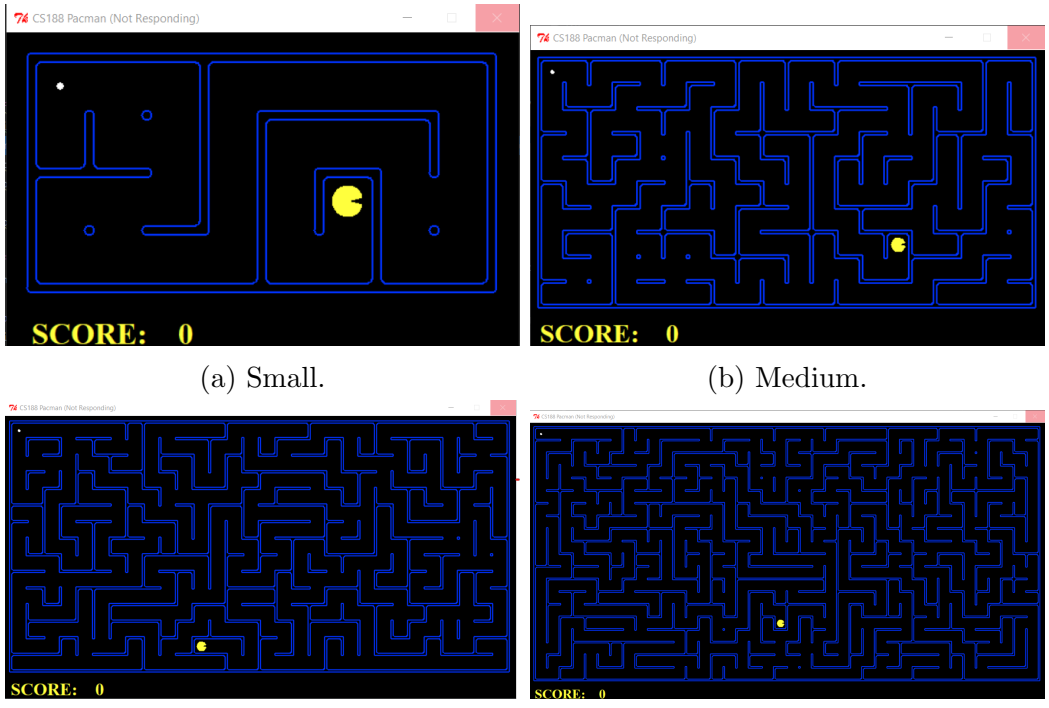
In my program, I changed the configuration of the Search Agent, in order to provide it with different search algorithms to use.

Chapter 7

Graphs and experiments (W_{11})

My approach was the following: divide the experiment by the size of the maze. This way, I defined 4 classes of mazes:

1. small maze (16 x 8)7.1
2. medium mazes (40 x 20)7.1
3. large mazes (60 x 30)7.1
4. extra large mazes (80 x 40)7.1



(a) Small. (b) Medium.

(c) Large. (d) Extra Large.

Figure 7.1: Maze Sizes

Since the maze layout is a random one, I would not obtain trust worthy results if I only ran the game once for each search algorithm. So, for each of this maze classes, I ran the program 30 times on 30 different layouts, for each of the 6 search algorithms. More than that, I wanted

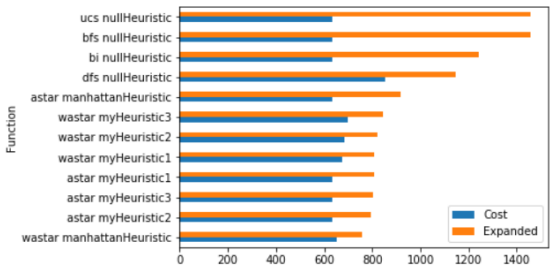
to test the informed search strategies with different heuristics. So I ran a different version of Astar and Weighted Astar, with the following heuristics:

1. Manhattan Distance Heuristic
2. the heuristic I defined, weighing in the number of walls between the position and goal with a weight of 1
3. the heuristic I defined, weighing in the number of walls between the position and goal with a weight of 2
4. the heuristic I defined, weighing in the number of walls between the position and goal with a weight of 3

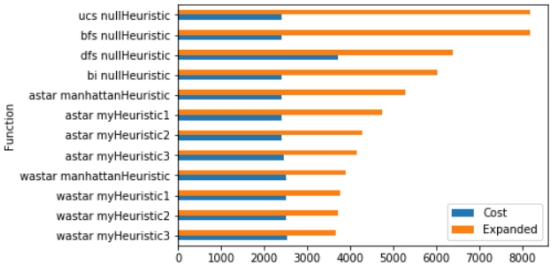
Therefore, for each layout I ran 12 algorithms.

The output I recorded was the *Cost* of the found path and the number of nodes that were *Expanded* to get to the goal.

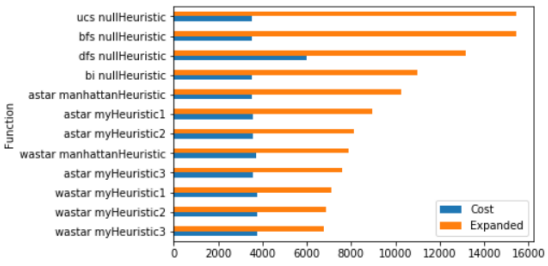
The results I obtained were, in some ways, what I expected them to be:



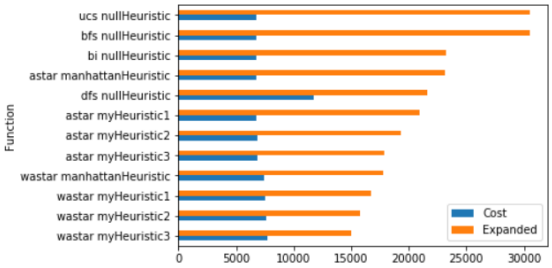
(a) Cost and Explored for Small mazes.



(b) Cost and Explored for Medium mazes.



(c) Cost and Explored for Large mazes.



(d) Cost and Explored for Extra large mazes.

Figure 7.2: Results

And for a more precise analysis, here are the values based on which the previous graphs were generated:

Results for small mazes:

Function	Cost	Expanded
wastar manhattanHeuristic	652	756
astar myHeuristic 2	632	794
astar myHeuristic 3	636	805
astar myHeuristic 1	632	808
wastar myHeuristic 1	676	809
wastar myHeuristic 2	684	823
wastar myHeuristic 3	700	843
astar manhattanHeuristic	632	917
dfs	852	1147
bi	632	1243
bfs	632	1458
ucs	632	1458

Results for medium mazes:

Function	Cost	Expanded
wastar myHeuristic 3	2540	3663
wastar myHeuristic 2	2512	3713
wastar myHeuristic 1	2508	3761
wastar manhattanHeuristic	2520	3901
astar myHeuristic 3	2448	4145
astar myHeuristic 2	2404	4276
astar myHeuristic 1	2404	4752
astar manhattanHeuristic	2396	5289
bi	2396	6031
dfs	3716	6395
bfs	2396	8192
ucs	2396	8192

Results for large mazes:

Function	Cost	Expanded
wastar myHeuristic 3	3776	6748
wastar myHeuristic 2	3756	6885
wastar myHeuristic 1	3764	7088
astar myHeuristic 3	3568	7597
wastar manhattanHeuristic	3700	7897
astar myHeuristic 2	3548	8137
astar myHeuristic 1	3548	8959
astar manhattanHeuristic	3532	10255
bi	3532	10966
dfs	5996	13150
bfs	3532	15456
ucs	3532	15456

Results for extra large mazes:

Function	Cost	Expanded
wastar myHeuristic 3	7738	15029
wastar myHeuristic 2	7650	15808
wastar myHeuristic 1	7578	16708
wastar manhattanHeuristic	7398	17791
astar myHeuristic 3	6882	17900
astar myHeuristic 2	6818	19319
astar myHeuristic 1	6766	20913
dfs	11786	21615
astar manhattanHeuristic	6762	23161
bi	6762	23270
bfs	6762	30532
ucs	6762	30532

In terms of optimality (*Cost*), the BFS algorithm is the best in all 4 cases. This was expected, since by the nature of this algorithm, it always finds the shortest path between a root and any other node. But the number of the *Expanded* nodes registered by this algorithm is significantly higher than other algorithms.

The UCS algorithm behaves exactly like the BFS algorithm in this case. It is possible we would have seen an improvement in terms of the number of the expanded nodes, if the cost of actions in the maze were different than 1.

The Bidirectional BFS also finds the optimal path, and the number of expanded nodes is lower than the one registered by BFS, so we can already see an improvement.

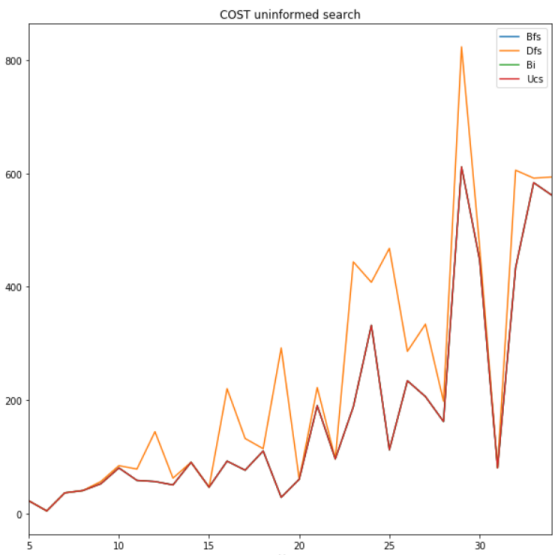
The Depth First Search algorithm is the one that registeres the worst *Cost* number. Combined with the still high number of Expanded nodes, I consider this the worst choice of search algorithm for this kind of problem.

As we have already noticed, the uninformed search strategies tend to explore a number of nodes that is almost double to the one of the informed search strategies. But they do find the shortest path (except for DFS).

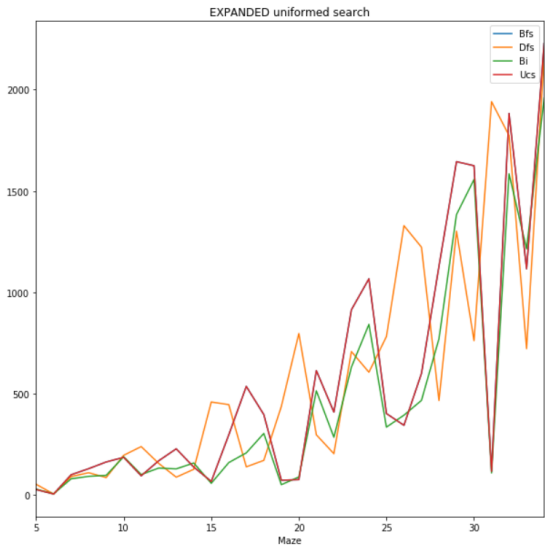
Moving over to the informed search strategies, we notice that for small sized mazes, the Astar algorithm gives a better number of expanded nodes than the Weighted Astar. We can argue than since the path to the goal is shorter, it doesn't matter as much in the beginning to get *anywhere*, as fast as possible, as the Weighted Astar behaves. And is more important to move in the general direction of the goal. But, if we consider the Cost of the path, Astar is closer to the optimal solution in all 4 scenarios.

Also, we have to take into consideration the heuristic that has been used. We can observe that as the size of the maze increases, the Manhattan Distance heuristic becomes less and less useful, and the heuristic I defined proves itself to be better. From this we conclude that the number of walls we approximated to be between the position and the goal becomes more of an important factor as we are dealing with bigger sized mazes. And more than this, the heuristic using the weight = 3 (meaning that the number of walls is multiplied by 3, and then added to the Manhattan distance between points) is the one that leads to the lowest number of expanded nodes.

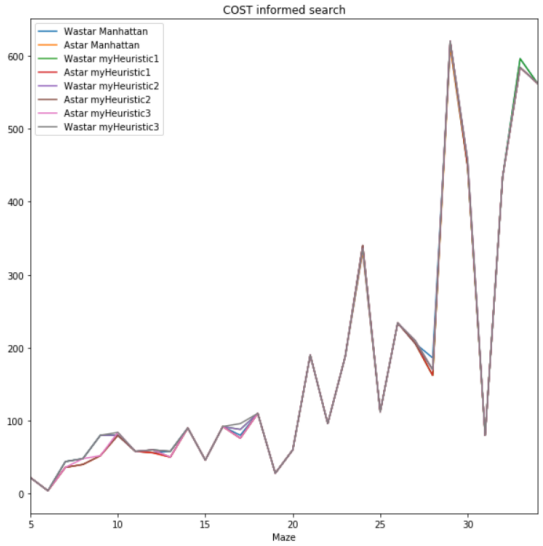
Another scenario I considered is one in which the size of the maze varies. The parameters in which the program was ran remain the same: I generated a random layout, and ran the 12 algorithms on this maze. The experiment was repeated 30 times, and the results were recorded. They can be seen in the following charts.



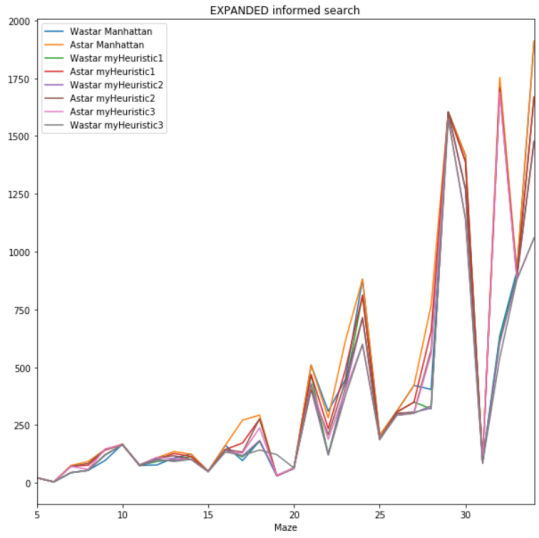
(a) Cost - uninformed search



(b) Explored nodes - uninformed search



(c) Cost - informed search



(d) Explored nodes - informed search

Figure 7.3: Results - varying maze size

As expected, the path cost and number of explored nodes increase with the increase of the maze size.

My conclusion after the observations of this experiment is that when deciding what search algorithm to use in a position search problem, one must first ask what does one value more. If it is more important to find the lowest cost path, than an uninformed cost strategy, such as BFS or UCS is recommended. If instead it is more important to get to the goal with the lowest number of explored states, than it is worth looing into informed search strategy. And in order to get the best performance, the used function and heuristic can be decided dynamically at runtime, based on the size of the maze.

Chapter 8

Related work and documentation (W_{12})

My results are similar to other results from papers I have found on the subject. It is wildly accepted that the Astar algorithm is one of the best algorithms to use for the pathfinding problem, and it is to this day used in the world of computer games. For example, this paper[1], Pathfinding in strategy games and maze solving using A search algorithm, describes its use in strategy games, and how this algorithm is considered to be an efficient and inexpensive approach.

I also found other works describing even more search algorithms, such as Maze Solving Algorithms, Paintbrush Algorithm, Breadth First Search Algorithm, Depth First Search Algorithm, Trimming Algorithm and Compass Algorithm[2]. This paper was particularly interesting, since it describes some algorithms developed by the authors. The one that is the highlight of the whole paper is the so called *Paintbrush algorithm*, which is similar in concept to the floodfill in paintbrush (if you color a figure, any space that has contact to the figure will be colored). So, if you consider that you have a maze, each time you have to choose between going left or right, first close the right way, and fill the current cell with paint. If the goal cell is filled with paint as well, it means that by going to the left, you will surely find the path to the goal. If the goal cell is not painted, than the path, if there exists one, must be to the right.

Of course, I also used as guiding the book *Artificial intelligence: a modern approach*[3], which I found to be great help in clarifying AI concepts and in documenting about the chosen search strategies.

Appendix A

Your original code

This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained. Including in this section any line of code taken from someone else leads to failure of IS class this year. Failing or forgetting to add your code in this appendix leads to grade 1. Don't remove the above lines.

A.1 Heuristic function

```
def myHeuristic(position, problem, info={}):

    xy1 = position
    xy2 = problem.goal
    manhattan = abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
    xmin = min(xy1[0], xy2[0])
    xmax = max(xy1[0], xy2[0])
    ymin = min(xy1[1], xy2[1])
    ymax = max(xy1[1], xy2[1])

    list1 = problem.walls.data[xmax][ymin:ymax + 1]
    list2 = problem.walls.data[xmin][ymin:ymax + 1]
    list3 = [el[ymin] for el in problem.walls.data[xmin + 1:xmax]]
    list4 = [el[ymax] for el in problem.walls.data[xmin + 1:xmax]]
    count = list1.count(True) + list2.count(True)
                + list3.count(True) + list4.count(True)
    count /= 2

    return manhattan + int(problem.heuristicWeight) * count
```

A.2 Node class

```
class Node:
    def __init__(self, state, cost=0, action=None, parent=None, depth=0):
        self.state = state
        self.cost = cost
        if action is None:
            self.action = []
```

```
        else:
            self.action = action
        self.parent = parent
        self.depth = depth

    def __repr__(self):
        return "<Node %s>" % (self.state,)

    def getNeighbors(self, problem):
        neighbors = []
        for node in problem.getSuccessors(self.state):
            neighbors.append(node)
        return neighbors
```

A.3 Breadth First Search

```
def breadthFirstSearch(problem):
    from util import Queue

    start = Node(problem.getStartState())
    explored, frontier = [], Queue()
    frontier.push(start)

    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.state):
            return node.getAction()
        if node.state not in explored:
            explored.append(node.state)
            for child in node.getNeighbors(problem):
                if child[0] not in explored and child[0] not in frontier.list:
                    actions = node.getAction()[:]
                    actions.append(child[1])
                    frontier.push(Node(child[0],
                                       node.cost + child[2], actions, node))
    return []
```

A.4 Depth First Search

```
def depthFirstSearch(problem):
    from util import Stack

    start = Node(problem.getStartState())
    explored, frontier = [], Stack()
    frontier.push(start)

    while not frontier.isEmpty():
        node = frontier.pop()
```

```
        if problem.isGoalState(node.state):
            return node.getAction()
        explored.append(node.state)
        for child in node.getNeighbors(problem):
            if child[0] not in explored and child[0] not in frontier.list:
                actions = node.getAction()[:]
                actions.append(child[1])
                frontier.push(Node(child[0],
                            node.cost + child[2], actions, node))
    return []
```

A.5 Uniform Cost Search

```
def uniformCostSearch(problem):
    from util import PriorityQueue

    start = Node(problem.getStartState())
    explored, frontier = [], PriorityQueue()
    frontier.push(start, start.getPathCost())

    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.state):
            return node.getAction()
        if node.state not in explored:
            explored.append(node.state)
            for child in node.getNeighbors(problem):
                if child[0] not in explored:
                    actions = node.getAction()[:]
                    actions.append(child[1])
                    childNode = Node(child[0],
                                     node.cost + child[2], actions, node)
                    frontier.update(childNode, childNode.getPathCost())
    return []
```

A.6 Bidirectional Search

```
def computePath(nodeS, nodeG):
    actionsStart = nodeS.getAction()
    actionsGoal = nodeG.getAction()
    actualActionGoal = []
    for action in actionsGoal[::-1]:
        if action == 'South':
            actualActionGoal.append('North')
        if action == 'North':
            actualActionGoal.append('South')
        if action == 'East':
            actualActionGoal.append('West')
```

```
        if action == 'West':
            actualActionGoal.append('East')
    return actionsStart + actualActionGoal

def bidirectionalSearch(problem):
    from util import Queue

    start = Node(problem.getStartState())
    goal = Node(problem.getGoalState())
    explored, frontier1, frontier2 = [], Queue(), Queue()
    frontier1.push(start)
    frontier2.push(goal)

    while not frontier1.isEmpty() and not frontier2.isEmpty():
        if not frontier1.isEmpty():
            node1 = frontier1.pop()
            if node1.state == goal.state:
                return node1.getAction()
            if node1.state in (item.state for item in frontier2.list):
                for node in frontier2.list:
                    if node.state == node1.state:
                        return computePath(node1, node)
            if node1.state not in explored:
                explored.append(node1.state)
                for child in node1.getNeighbors(problem):
                    if child[0] not in explored:
                        actions = node1.getAction()[:]
                        actions.append(child[1])
                        frontier1.push(Node(child[0],
                                            child[2], actions, node1))
        if not frontier2.isEmpty():
            node2 = frontier2.pop()
            if node2.state == start.state:
                return node2.getAction()
            if node2.state in (item.state for item in frontier1.list):
                for node in frontier1.list:
                    if node.state == node2.state:
                        return computePath(node, node2)
            if node2.state not in explored:
                explored.append(node2.state)
                for child in node2.getNeighbors(problem):
                    if child[0] not in explored:
                        actions = node2.getAction()[:]
                        actions.append(child[1])
                        frontier2.push(Node(child[0],
                                            child[2], actions, node2))

    return []
```

A.7 Astar Search

```
def aStarSearch(problem, heuristic=nullHeuristic):
    from util import PriorityQueue

    start = Node(problem.getStartState())
    explored, frontier = [], PriorityQueue()
    frontier.push(start, start.cost)

    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.state):
            return node.getAction()
        if node.state not in explored:
            explored.append(node.state)
            for child in node.getNeighbors(problem):
                if child[0] not in explored:
                    actions = node.getAction()[:]
                    actions.append(child[1])
                    childNode = Node(child[0],
                                     node.cost + child[2], actions, node)
                    frontier.update(childNode,
                                    childNode.getPathCost() +
                                    heuristic(childNode.state, problem))
    return []
```

A.8 Weighted Astar Search

```
def weightedAStarSearch(problem, heuristic=nullHeuristic):
    from util import PriorityQueue
    from searchAgents import manhattanHeuristic

    start = Node(problem.getStartState())
    explored, frontier = [], PriorityQueue()
    frontier.push(start, start.cost)

    N = manhattanHeuristic(start.state, problem)

    sigma = 4

    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.state):
            return node.getAction()
        if node.state not in explored:
            explored.append(node.state)
            for child in node.getNeighbors(problem):
                if child[0] not in explored:
                    actions = node.getAction()[:]
```

```
                    actions.append(child[1])
                    childNode = Node(child[0], node.cost + child[2], actions,
                                                    node, node.depth + 1)
                    if childNode.depth > N:
                        weight = 1
                    else:
                        weight = 1 + (1 - childNode.depth / N) * sigma
                    cost = childNode.getPathCost()
                              + weight * heuristic(childNode.state, problem)
                    frontier.update(childNode, cost)
    return []
```

A.9 Generation of graphs

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv("C:/Users/Bianca/Desktop/resultsSmall.csv")

df = pd.DataFrame(data)
barAll = pd.DataFrame(data, columns=['Function', 'Cost', 'Expanded'])
                 .groupby('Function')
                .sum()
                .sort_values(by = ['Expanded', 'Cost'], ascending=True)
barCost = pd.DataFrame(data, columns=['Function', 'Cost'])
                   .groupby('Function')
                   .sum()
                   .sort_values(by = ['Cost'], ascending=True)
barExpanded = pd.DataFrame(data, columns=['Function', 'Expanded'])
                            .groupby('Function')
                            .sum()
                            .sort_values(by = ['Expanded'], ascending=True)

barAll.plot(kind='barh')
barCost.plot(kind='barh')
barExpanded.plot(kind='barh')
```

Appendix B

Quick technical guide for running your project

The requirements and the instructions on installing the tool have already been specified: 1
Step by step technical manual:

1. open the file *pacman.py*
2. depending on which type of maze size you want to run the problem, the parameters of the function *generateMaze* will have to be changed. The values for each category can be found commented in the lines above the function definition
3. run the program
4. the results will be saved to a .csv file on the computer's Desktop

In order to generate the graphs, you must first install *Jupyter, Pandas* and *Matplotlib*. I am using the python manager package, pip. To install them with pip, open a terminal window and type:

```
pip install jupyterlab
pip install matplotlib
pip install pandas
```

After the instalation, open a new terminal window and type `jupyter notebook` and create a new Notebook. You can use the code provided by me in the code section A.9, and generate the graphs based on the data from the .csv file.

Bibliography

[1] Nawaf Hazim Barnouti, Sinan Sameer Mahmood Al-Dabbagh, and Mustafa Abdul Sahib Naser. Pathfinding in strategy games and maze solving using a search algorithm. *Journal of Computer and Communications*, 4(11):15, 2016.

[2] Shantur Rathore Manish Chand, Mayank Goel. Maze solving algorithms, paintbrush algorithm, breadth first search algorithm, depth first search algorithm, trimming algorithm and compass algorithm. page 15.

[3] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[4] UC Berkeley. The pac-man project. `http://ai.berkeley.edu/project_overview.html`.

[5] Wang Tiles. Perfect maze generator. `http://www.cr31.co.uk/stagecast/wang/perfect.html`, 2019.

[6] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=931177297`, 2019. [Online; accessed 31-December-2019].

[7] Wikipedia contributors. Maze generation. `https://rosettacode.org/wiki/Maze_generation`, 2019.