



Intelligent Systems

Laboratory activity 2018-2019

Project title: Pacman - *Heuristic Search* and *Adversial Search*
Tool: Pacman agent

Name: Blaga Cristian-Ioan
Group: 30433
Email: blaga.cristi23@gmail.com

Assoc. Prof. dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro



Contents

1	AI projects and tools (W_1)	4
1.1	Exercises	4
2	Installing the tool (W_2)	8
2.1	Exercises	8
3	Running and understanding examples (W_3)	10
3.1	Exercises	10
4	Understanding conceptual instrumentation (W_4)	13
4.1	Exercises	13
5	Project description (W_5)	14
5.0.0.0.1	What will the system do?	14
5.0.0.0.2	Scope of the program	14
5.1	Narrative description	14
5.2	Facts	15
5.3	Knowledge acquisition	16
5.4	Exercises	16
6	Results (W_7)	18
6.1	Introduction	18
6.2	General Search Techniques	18
6.2.1	Uninformed Search Techniques	18
6.2.1.1	Breadth-first Search	19
6.2.1.2	Uniform Cost Search	20
6.2.1.3	Depth First Search	23
6.2.1.4	Finding all the corners	26
6.2.2	Informed Search Strategies	28
6.2.2.1	Greedy Best First Search	28
6.2.2.2	A* Search ALgorithm	29
6.2.2.3	Corner Problem Heuristic	30
6.2.2.4	Food Search Heuristic	31
6.3	Adversarial Search Techniques	33
6.3.1	Reflex Agent	34
6.3.2	MinMax Agent	35
6.3.2.1	AlphaBeta pruning	38
6.3.2.2	Expectiminimax algorithm	38
6.3.2.3	Evaluation function	43
6.4	General Implementation Details	44
6.5	Grading	45

6.6 Graphs and Experiments 45

A Your original code 48

B Quick technical guide for running your project 57

Chapter 1

AI projects and tools (W_1)

My personal objectives for this class are:

1. Acknowledging the purpose of this laboratory.
2. Choosing an interesting semestrial project to work on.

1.1 Exercises

1. Compile the `is.tex` file in order to start writing your notes. Recall that this documentation is also a *support for you*, during the design and implementation of you ideas. Make an habit in writing down your ideas from the first week in a professional manner.
2. Identify 3 Web resources with ideas on student AI projects.
3. Think at one of your hobbies. Would be possible to develop something on that line?
4. Identify a media source for AI-news. Investigate interesting ideas in that news. Do the AI-technologies behind these ideas appear in the AIMA book?
5. Identify AI journals in Science Direct and Springer Verlag. Browse some abstracts from these journals.
6. Identify an AI competition. Consider making a team of 2-3 students to participate at that competition.
7. Imagine that you are the founder of a start-up. What kind of innovative project would be feasible for you company?
8. Write a short list (3 to 5) of possible projects for this lab. Be ambitious.
9. Display network information for your workstation.
10. Connect via `ssh` to another workstation.

Solution to exercise 2

Three Web resources i have found with ideas on student AI projects are:

1. Machine learning project suggestions at School of Computer Science, Carnegie Mellon University
2. Pacman projects from the Berkeley University.
3. MITOpenCourseware Collection of projects at MIT

Solution to exercise 3

One of my main hobbies that I enjoy doing in order to relax is listening to music. What is interesting about this hobby is that I do not identify with one single music genre, but rather associate the music I listen to with the feelings or the mood at that specific moment. I think this could be the premise of an interesting AI project in mapping a relation between the music genre and human's psychology based on different subjects, feelings and emotions. This mapping would be relative, of course, because each individual is different, but I think that it could give an interesting insight in how humans react and manifest their inner emotions.

Solution to exercise 4

One reliable media source I have found regarding AI news is Science Daily. On this website, I have seen some interesting AI applications that are being used, like:

1. Combining multiple images in order to identify and catch suspects
2. Helping computers in filling gaps between 2 frames in a video

Solution to exercise 5

1. Artificial Intelligence in Cardiology:

Artificial intelligence and machine learning are poised to influence nearly every aspect of the human condition, and cardiology is not an exception to this trend. This paper provides a guide for clinicians on relevant aspects of artificial intelligence and machine learning, reviews selected applications of these methods in cardiology to date, and identifies how cardiovascular medicine could incorporate artificial intelligence in the future. In particular, the paper first reviews predictive modeling concepts relevant to cardiology such as feature selection and frequent pitfalls such as improper dichotomization. Second, it discusses common algorithms used in supervised learning and reviews selected applications in cardiology and related disciplines. Third, it describes the advent of deep learning and related methods collectively called unsupervised learning, provides contextual examples both in general medicine and in cardiovascular medicine, and then explains how these methods could be applied to enable precision cardiology and improve patient outcomes.

2. Optimizing immune cell therapies with artificial intelligence:

(Only a part of the abstract)

(...)We rely on the mathematical model of Soto-Ortiz and Finley (Soto-Ortiz and Finley, 2016) for the interactions between the tumor growth, angiogenesis and immune system reactions. Our optimization algorithm belongs to the class of Monte-Carlo tree search algorithms. The objective consists in finding the minimal total drug doses for which an injection pattern yields tumor eradication. Our results are twofold. First, optimized injection protocols enable to significantly reduce the total drug dose for tumor elimination. (...) Our second result is that administering a dose equal to the maximal standard dose allows for later diagnosis date compared to standard protocol.

Solution to exercise 6

I would participate in the Student StarCraft AI Tournament together with Buda Vlad - Cristian and Bogdan Darius.

Solution to exercise 7

I think that a tool that would help almost any start-up company is a tool that is capable of identifying different customer needs with foresight of how these needs would eventually evolve. Of course, this tool would be implemented with an underlying Artificial Intelligence technology and algorithm.

Solution to exercise 8

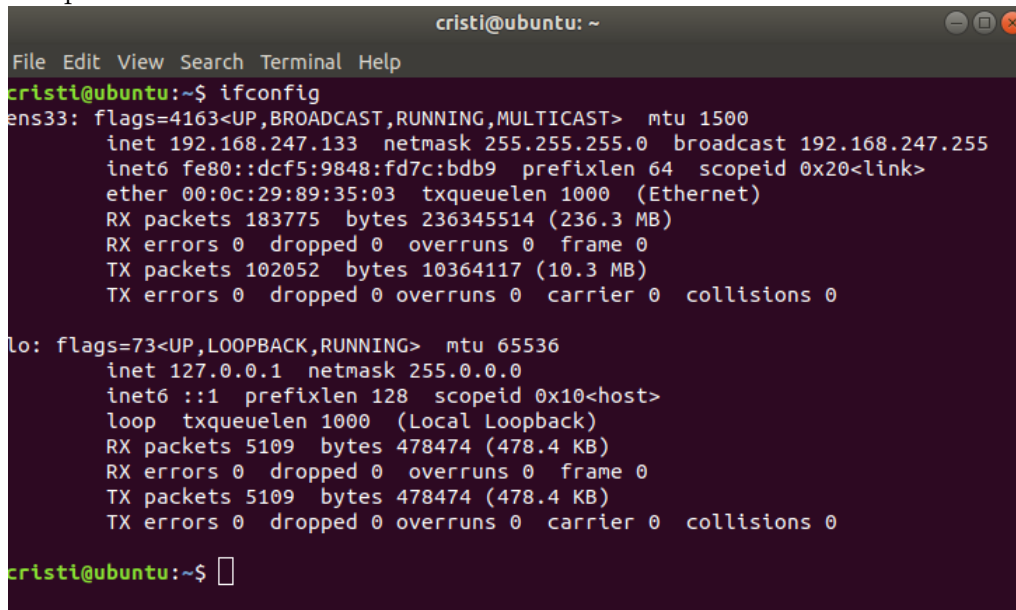
Some project ideas for this lab are:

1. Fake opinion detection system.
2. Usage of autotune in songs.
3. Usage of photoshop in photos/CGI in videos.
4. Developing an agent that would be capable of writing a bestselling novel.

Solution to exercise 9

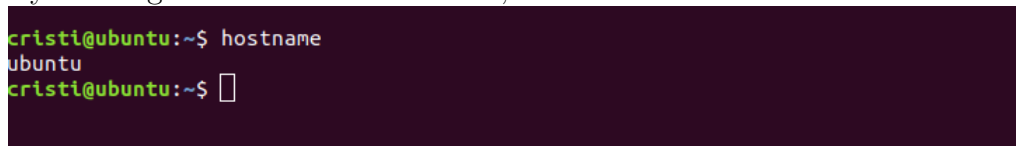
The network information for my laptop is:

1. By running the *ifconfig* command, we can see the network configuration of the computer we work on:



```
cristi@ubuntu: ~  
File Edit View Search Terminal Help  
cristi@ubuntu:~$ ifconfig  
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500  
    inet 192.168.247.133  netmask 255.255.255.0  broadcast 192.168.247.255  
    inet6 fe80::dcf5:9848:fd7c:bdb9  prefixlen 64  scopeid 0x20<link>  
    ether 00:0c:29:89:35:03  txqueuelen 1000  (Ethernet)  
    RX packets 183775  bytes 236345514 (236.3 MB)  
    RX errors 0  dropped 0  overruns 0  frame 0  
    TX packets 102052  bytes 10364117 (10.3 MB)  
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0  
  
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536  
    inet 127.0.0.1  netmask 255.0.0.0  
    inet6 ::1  prefixlen 128  scopeid 0x10<host>  
    loop txqueuelen 1000  (Local Loopback)  
    RX packets 5109  bytes 478474 (478.4 KB)  
    RX errors 0  dropped 0  overruns 0  frame 0  
    TX packets 5109  bytes 478474 (478.4 KB)  
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0  
  
cristi@ubuntu:~$
```

2. By running the *hostname* command, we can see the hostname:



```
cristi@ubuntu:~$ hostname  
ubuntu  
cristi@ubuntu:~$
```

Solution to exercise 10

Connecting to another computer via ssh is done with the following command:

ssh is@ip-of-computer-to-connect-to

Sending a file to another computer via ssh(after the connection has been established) is done with the following command:

scp file-name is@your-ip:path

Closing the ssh connection is done with the following command:

exit

Chapter 2

Installing the tool (W_2)

My personal objectives for this class are:

1. Installing Pycharm on my laptop in order to be able to run the program.
2. Understanding the running parameters specified on the Berkeley website.

2.1 Exercises

1. List the steps done for installing the tool.
2. List the exact commands needed to run the tool.
3. What other AI tools can use the output of your tool? Can you identify such tools on the Web? Try to assess the difficulty level and risks of integrating such tools for your project.

Solution to exercise 1

The steps I have taken for installing the tool are:

1. Installing python 2 using the command:
sudo apt install python2.7 python-pip
2. Installing Pycharm using the commands:
sudo add-apt-repository ppa:mystic-mirage/pycharm
sudo apt-get install pycharm
3. Setup the project in pycharm by creating a new project, including the source files and setting the python interpreter to the one previously installed.

Solution to exercise 2

I am running the project from pycharm, but, the only difference between running it from pycharm or from the command line is that, for any of the commands that I will list, the keyword *python* should be added in front of them. In pycharm, the arguments to run a file are set as a configuration before running the file.

Thus, in order to run different functionalities of the program, the following commands can be used:

1. To run the autograder, just run the *autograder.py* file. No arguments are required.
2. To run the original pacman game, just run the *pacman.py* file. No arguments are required.
3. In order to run the pacman game with a layout and an agent of your choosing, run the *pacman.py* file with the *-layout layout-name -pacman agent-name* arguments (the layout file needs to be placed in the *layouts* folder).
4. You can also specify the frame time of the game by adding the *-frameTime value* argument.
5. In the *search.py* file, different functions are left unimplemented (functions that decide the behavior of the pacman). The *dfs* function is the one implicitly used, but, if you want to use another function, just add the *-a fn=function-name-abbreviation* argument. Also, the heuristic can be specified by adding to the previous argument *, heuristic=heuristic-name*. In the same manner, the problem that the pacman has to 'solve' can be specified by adding to the previous argument *prob=problem-name*.
6. In order to see all the possible arguments, their possible values and all the combinations available, run the *pacman.py* file with the *-h* argument.

Solution to exercise 3

I do not think that precisely the output of my program could be used by other AI tools, but rather the algorithms that are implemented in order to achieve the goals of the project would have applicability. I believe that any AI tool that needs to do a search in a large universe would make use of these algorithms (or even better algorithms) and heuristics. The applicabilities of these algorithms are endless, but, of course, one has to think about the efficiency and how the context and the problem (or problems) an AI agent is trying to solve affect these parameters before making a wise decision regarding the approach they chose in order to overcome the problem. Mostly all the time, the particularity of data is the one that dictates the efficiency of a program, not the algorithm itself.

Thus, in order to be able to integrate other algorithms and heuristics in my project, I should completely comprehend the use cases this program tries to achieve in order make a correct decision. However, different tools may be integrated in order for their performance to be tested on this particular context and problem.

Chapter 3

Running and understanding examples (W_3)

My personal objectives for this class are:

1. Run the different examples that are presented on the Berkeley website.
2. Identify and understand the real-life problems that the algorithms that I implement may solve.

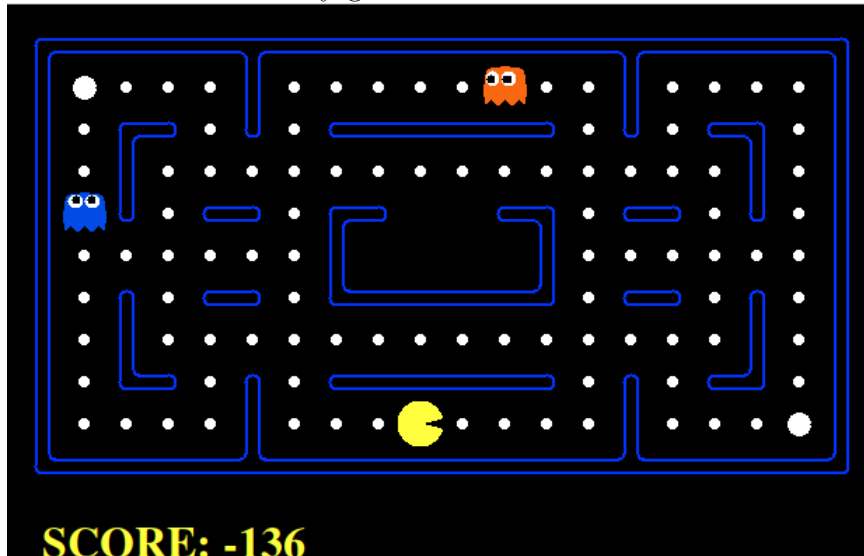
3.1 Exercises

1. Detail one example that you run. Which are the input and the output? Describe the structure of the code.
2. Describe the real world problems that can be solved by your tool/algorithm.

Solution to exercise 1

I will run two examples in order to show two different functionalities of the pacman program:

1. By running the *pacman.py* file with no arguments, the original pacman game will be run. The items present are: pacman, the ghosts and the food. The pacman can be controlled with the arrows. There is also the score present on the screen, that is a function of the time that has passed since the game began, how much food have we eaten and how many ghosts have we defeated.



2. By running the *pacman.py* file with the arguments *-layout testMaze -pacman GoWestAgent*, a window will open in which we will have a layout with a single row and a pacman that moves by itself west. After eating all the dots on the line, the game will close. The *-layout* argument specifies the map in which the pacman will run (these maps are located in the *layouts* folder) The *-pacman* specifies the agent that will control the pacman.



Solution to exercise 2

As specified before, not the tool I am using, but the algorithms that I implement have real world applicability. These algorithms have as ultimate goal to complete a search. Some of them are more efficient than others (on real world data) because they rely on heuristics. Heuristic search is a fundamental problem-solving method in artificial intelligence. For most AI problems, the sequence of steps required for solution is not known from the start but must be determined by a systematic trial-and-error exploration of alternatives. Thus, some applicabilities are (the information has been taken from :

1. Automotive design - designing composite materials and aerodynamic shapes for race cars and regular means of transportation.
2. Optimized telecommunications routing - development of dynamic and anticipatory routing of circuits for telecommunications network.
3. Trip, traffic and shipment routing - applications of the *travelling salesman problem* can be used to plan efficient routes and schedules.
4. Gene expression profiling - in order to make analysis of gene expression profiles much quicker and easier with the goal of establishing which genes produce certain diseases.
5. Encryption and code breaking - creating encryption for sensitive data as well to break those codes.

Chapter 4

Understanding conceptual instrumentation (W_4)

4.1 Exercises

1. Big O complexity
2. Which are the latex options to write algorithms? Describe in one paragraph the main features of one such package for algorithms.

Solution to exercise 1

Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows. Thus, it is able to characterize and classify functions based on their growth rates. Of course, this notation is delimited by the best case and the worst case scenario of data, without taking into consideration the particularities of data and how probable certain cases are to appear.

Solution to exercise 2

Using latex in order to write algorithms, we have different options available, some of them being:

1. `\caption` used to add a short description of the algorithm
2. `\KwIn` used to specify data
3. `\KwOut` used to specify output data
4. `\KwIn` used to specify data
5. `\mathcal` in order to use mathematical notations
6. `\ForEach $o \in \mathcal{O} \delta $` used to create a for loop in which we can place instruction that would be executed iteratively. Of course, these for loops can be imbricated.

Chapter 5

Project description (W_5)

My personal objectives for this class are:

1. Find resources from where I can learn and understand the algorithms that I have to study and implement.
2. To identify different projects/scenarios in which the algorithms are used.

5.0.0.0.1 What will the system do?

My main interest in the development of the program is the study of general search algorithms with the final goal of helping Pacman traverse and finish a maze world whilst also collecting food efficiently. Afterwards, I will also study general adversarial search algorithms with the goal of helping Pacman traverse, finish and win a game of Pacman in which there may be food, capsules and ghosts.

5.0.0.0.2 Scope of the program

All of the algorithms and principles I study and implement in these projects have real world applicabilities, but, in order for them to be understood and to be more easily implemented, they are studied on a toy problems, such as:

1. Pacman solving a maze
2. Pacman eating all the food in a maze
3. Pacman beating the game

5.1 Narrative description

As previously specified, the main purpose specified by the website of the Berkely University regarding these projects are:

1. For the first project (study of general search techniques):
(...) your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.
2. For the second project (study of general adversarial search techniques):

In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.

The algorithms that will be implemented along the way are:

1. For the first project:
 - (a) Depth First Search
 - (b) Breadth First Search
 - (c) Uniform Cost Search
 - (d) A Star Search
 - (e) Greedy Best Search
2. For the second project:
 - (a) Minimax
 - (b) Alpha Beta Prunning
 - (c) Expectimax

Obs. Besides these algorithms, there will also be some evaluation or heuristic functions implemented.

5.2 Facts

After a first look over the project written as a base for the Pacman project by the teachers at Berkeley University, one can see that you do not have to worry about:

1. Pacman maps interpretation - if you want to have a new map, all you have to do is to create a new file in the layouts folder with the following meaning:
 - (a) % represents a wall.
 - (b) *P* represents the Pacman entity.
 - (c) . represents a dot of food.
 - (d) *o* represents the empowering food dot.
 - (e) *G* represents the ghost entity(this will not be used in the current. project)
2. Computation of the score - when the game is run, the score is automatically calculated by the program. This score is a function of the time that has passed since the game has begun and the amount of food Pacman has eaten, whilst also taken into consideration if a scared ghost has been eaten.
3. The implementation logic for creating the game.
4. Different data structures that need to be used in different search algorithms are already predefined.
5. An autograder also exists that is capable of grading your solution.

Thus, the only thing you have to worry about is the implementation of the different search functions(while being consistent with the way the data is transmitted in the entire program). These search functions are called automatically by the pacman agent. When the program runs, it is run with different arguments, one of which specifies the search function that is to be used.

5.3 Knowledge acquisition

My main source for knowledge acquisition will be the book *Artificial intelligence: a modern approach, volume 3* by Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards.

5.4 Exercises

1. Sum up what is the aim of your project in a Twitter-sized phrase (140 characters)
2. Recall or identify an engineering methodology to write specifications. Cite this methodology and employ it for specifying your project.
3. Which are the differences between requirements and specifications?
4. Identify similar scenarios proposed by your colleagues. Think at some form of collaboration with one of your colleagues having similar interests.

Solution to exercise 1

The aim of the first project is the study and implementation of different search algorithms with the main purpose of helping Pacman to solve a maze world whilst also collecting food efficiently. The aim of the second project is the study and implementation of different adversarial search algorithms with the purpose of helping Pacman to beat the game.

Solution to exercise 2

An engineering methodology of writing requirements specification is by using *Natural language sentences*. In this methodology, requirements are written using numbered sentences, in natural language, avoiding technical words that an inexperienced person may not understand. Also, each sentence should express only one requirement.

Solution to exercise 3 Based on the book *Software Engineering* by Ian Sommerville, requirements are of two types:

- Functional requirements - statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Non-functional requirements - constraints on the services or functions offered by the system(like time constraints).

On the other hand, specifications are descriptions of the requirements for the system users who do not have detailed technical knowledge. Ideally, the specifications should be clear, unambiguous, easy to understand, complete and consistent in order to specify mainly only the external behavior of the system, without details of the system architecture or design. They may be specified in different ways:

- Natural language sentences
- Structured language sentences
- Design description languages
- Graphical notations
- Mathematical specifications.

Solution to exercise 4 One of my colleagues, Domuta Darius, has chosen the same project as I did, but, because of the nature of the project(that is, implementation and study of different search algorithms), almost any form of collaboration is impossible unfortunately.

Chapter 6

Results (W_7)

Observation

From this point forward, I will start documenting and explaining the algorithms (together with examples and analysis) I have implemented in this project rather than following the guideline of the initial documentation. Also, the book *Artificial intelligence - a modern approach* will be intensively used to provide examples or explanations. I am saying this here because it would be too redundant to specify this at each use.

6.1 Introduction

As I have mentioned before, the scope of this project is the study, implementation and analysis of different algorithms in the fields of **Search** and **Adversial search** problems.

6.2 General Search Techniques

The agents that implement the general search techniques are called *Problem-solving agents*. These techniques are mainly used in order to solve problems that have a precise goal definitions and the actions that the agent implementing them are not influenced by other factors than solely the steps taken in order to achieve the goal. Also, despite the adversial search techniques, these algorithms can not work unless they explore the entire search-state space (in worst case scenario).

Of course, these algorithms differ, based on their implementation and strategy they apply, but, as we shall see, for large problems, even the best one of them can not do it efficiently. Even so, some of them are still better than the other, depending on the strategy they chose to get to the goal.

6.2.1 Uninformed Search Techniques

The uninformed search techniques is a term that defines algorithms that use strategies that do not use any additional information regarding the states of a problem except than the one given in the problem definition. In the Pacman case, these strategies rely solely on:

- The initial state - in our case, this is the initial position of Pacman.
- The goal state - in our case, is the position of the food dot in the maze world(or the food dots, as we shall see).
- The world representation - in our case, the structure of the maze.

Based on only this information, the algorithms that use uninformed techniques can do only successor generation and be able to distinguish between a goal state and a non-goal state. The sole difference between these algorithms is the way in which they expand the nodes of the search-state space.

6.2.1.1 Breadth-first Search

Breadth-first search algorithm 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. The key idea of this approach is that all the nodes at a given depth (distance from the root node) are expanded before any nodes at the next level are expanded. The idea of always expanding the shallowest unexpanded node is easily implemented by the use of a queue (FIFO data structure). The pseudocode of the algorithm is shown below:

Algorithm 1: Breadth First Search Algorithm

Input: \mathcal{P} - the current problem to be solved
Output: $\langle \text{solution path} \rangle$, a set of nodes that represent the path from the root node to the goal node or *failure*

```

1  $Node \leftarrow$  a node with state  $InitialState(\mathcal{P})$ 
2 if  $GoalState(\mathcal{P}, Node)$  then
3    $\quad$  return  $Solution(Node)$ 
4  $Frontier \leftarrow$  a FIFO queue with  $Node$  as the only element
5  $Explored \leftarrow$  an empty set
6 while  $Frontier$  is not empty do
7    $Node \leftarrow Pop(Frontier)$ 
8    $\quad$  add  $State(Node)$  to  $Explored$ 
9    $\quad$  foreach  $action \in Actions(\mathcal{P}, State(Node))$  do
10     $\quad$   $Child \leftarrow ChildNode(\mathcal{P}, action, Node)$ 
11     $\quad$  if  $State(Child)$  not in  $Explored$  or  $Frontier$  then
12     $\quad$   $\quad$  if  $GoalState(\mathcal{P}, Child)$  then
13     $\quad$   $\quad$   $\quad$  return  $Solution(Child)$ 
14     $\quad$   $\quad$  add  $Child$  node to  $Frontier$ 
15 return failure

```

As it can be seen, each time a node is expanded (in other words, we generate all its successors), we verify (for the nodes that have not been explored yet), if the node in question represents a goal state. If so, then the algorithm computes the solution and returns it. Otherwise, the node is added to the LIFO queue in order to be expanded later. If no solution has been found and there are no more nodes to expand (the queue is empty), then the algorithm will return failure.

An interesting thing about this algorithm is that the LIFO queue actually represents a frontier between the explored nodes and the unexplored nodes. In other nodes, there are no two nodes connected with one of them being explored and the other being unexplored.

It can be easily seen that the algorithm is complete (that is, if there is a goal node at a certain depth, BFS algorithm will eventually find it). It is also easy to see that the algorithm returns an optimal solution - that is because we always explored the nodes at a depth d from the root node before any node at a depth $(d+1)$. Thus, we will always get the shortest path from the root to the goal node (of course, each step/edge has a constant cost of 1).

The problem with the algorithm is that both the time complexity and the space complexity

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Figure 3.12 Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.

Figure 6.1: Time and space complexity of BFS

have an exponential value. If we consider that each node has a number b of successors (the branching factor), and the goal node is situated at a depth d , it can easily be seen that, overall, we explore $\mathcal{O}(b^d)$ nodes. Regarding the space complexity, it can easily be seen that, for a depth d , the frontier stores roughly around $\mathcal{O}(b^d)$ nodes.

In order to truly comprehend the huge values of the BFS complexity, I will use the efficiency example from the *Artificial intelligence - a modern approach*. The table is built with a variable d , a branching factor of $b = 10$ and with the assumptions that:

- One million nodes can be generated in a second
- A node takes 1000 bytes of storage

Please look at 6.1

The lesson to be learned from this table is that *in general, exponential-complexity problems can not be solved by uninformed methods for any but the smallest instances*.

Please look at 6.2.

Implementation

The python implementation is presented below.

I will not enter implementation details, because the algorithm is just a translation of the strategy from pseudoco to python, with some differences regarding the compatibility with the problem at hand and the tool that uses the algorithm. Besides this, the code is also commented and self explanatory.

Below, an example of the BFS algorithm is presented. From it, we can see all the nodes that have been expanded in the search for the goal, thus giving us an understanding of the space size regarding the time complexity. Please look at 6.3

6.2.1.2 Uniform Cost Search

When all steps are equal, Breadth-first search algorithm is optimal because it always expands the shallowest unexpanded node. However, when the step-cost function changes, this algorithm is no longer optimal, but, with a simple extension, we can find an algorithm that is optimal with any step-cost function.

This strategy is based on the use of a *Min Priority Queue* instead of just a simple queue. In this type of queue, elements are extracted based on their priority, that is, the element with the smallest priority will be extracted first. In our case, we translate the priority of an element to the distance from the root node to that specific node. Thus, UCS always expands the node that has the *lowest path cost* $g(n)$. There are two other major differences from the BFS, mainly:

```

def breadthFirstSearch(problem):
    # initialization part of the algorithm
    from util import Queue
    queue = Queue()
    queue.push(problem.getStartState())
    visited = {problem.getStartState(): (None, None)}
    solution = []

    while not queue.isEmpty() and not solution:
        # extract a node from the queue
        node = queue.pop()
        # if the node is a goal state, then compute solution
        if problem.isGoalState(node):
            solution = computeSolution(node, visited)
        if not solution:
            # take each successor of the current node and add it to the queue if not visited
            for successor in problem.getSuccessors(node):
                if successor[0] not in visited:
                    visited[successor[0]] = (node, successor[1])
                    queue.push(successor[0])
    return solution

```

Figure 6.2: BFS algorithm

- Instead of testing a node against the goal state when the node is selected, we will test it when the node is expanded. The reason for this is that the node may be on a suboptimal path. In other words, only when the algorithm expands a node, we are sure that the lowest cost path to that node has been found. This is because of the difference explained below.
- The second difference from the BFS is that, while we expand a node, nodes that are already in the priority queue (that means, they have not yet been explored, so we are not sure yet of their lowest cost path), may have their priority (cost) changed. That is because, while we generate the successors of a node, we generate new paths that may have a lower cost than the ones we have found so far for the nodes in the priority queue.

The pseudocode of the algorithm can be seen below:

Optimality of the algorithm (and the idea that when we select a node, we are sure we have found the lowest cost path to that node) is built on the idea that if there was a node n' that could create a lower cost path to the node n in question, then that node n' would have sure been expanded before the node n , and, in case of a lower path, the path of the node n would have been updated.

From the implementation of the algorithm it can be seen that, when the cost of an action is constant and equal to 1, the algorithm behaves similarly to the BFS algorithm, the only difference being where the goal test is places (thus, UCS does some unnecessary work when finding a solution).

Complexity of the algorithm is quite hard to be calculated because the algorithm is guided by the path costs rather than the depths. Thus, we will just present the final formula, $\mathcal{O}(b^h)$ where h is $1 + C/\mathcal{E}$, where C is the cost of the optimal solution and \mathcal{E} is the smallest cost of an action. From this formula, it can easily be seen that UCS may have a worst case time and space complexity far bigger than the one of the BFS, because UCS explores firstly large trees of small steps before exploring paths involving large, useful steps.

Implementation

As before, we will not enter implementation details. The implementation can be seen below.

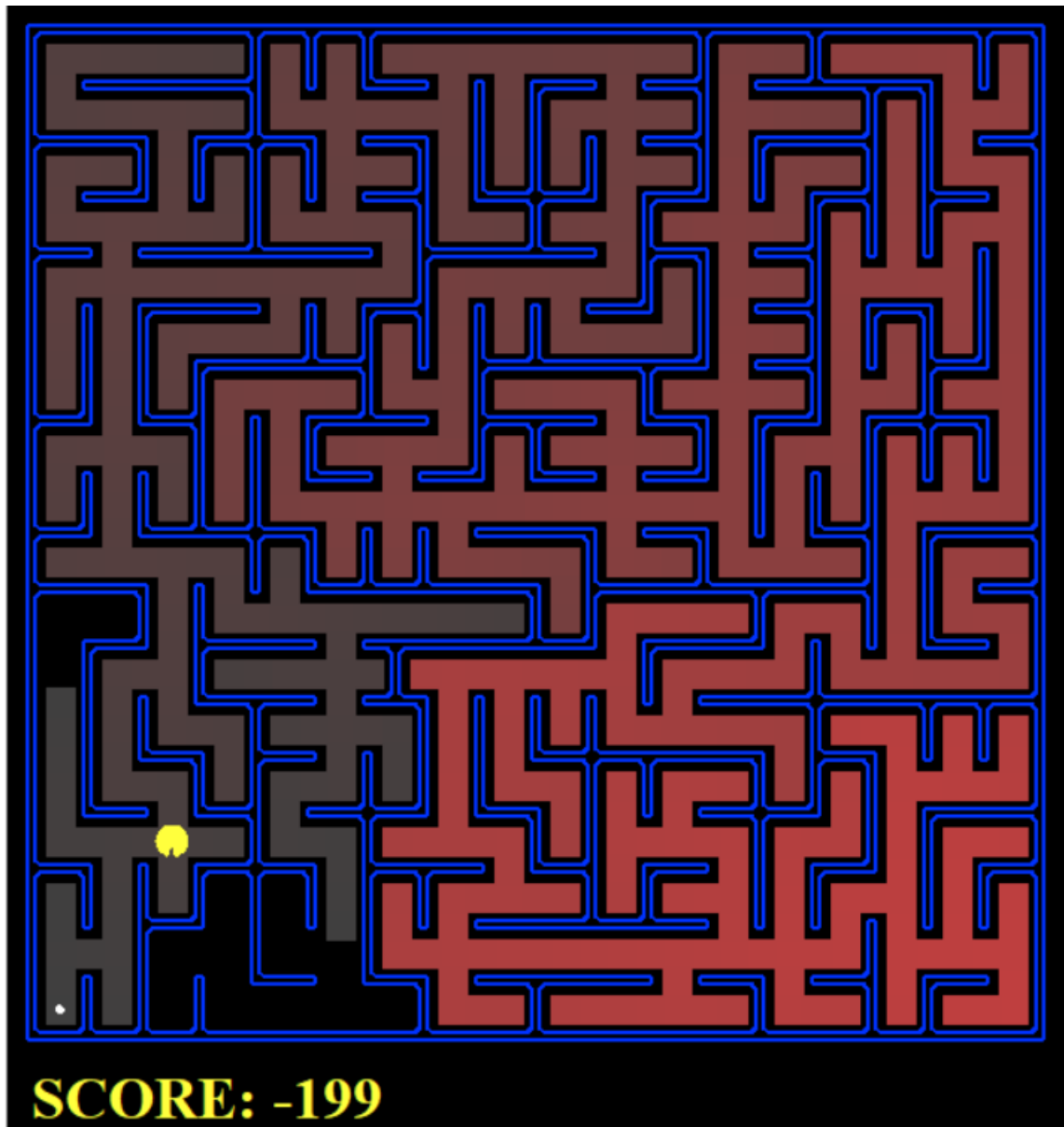


Figure 6.3: States explored by the BFS algorithm

Algorithm 2: *Uniform Cost Search Algorithm*

Input: \mathcal{P} - the current problem to be solved

Output: $\langle \text{solution path} \rangle$, a set of nodes that represent the path from the root node to the goal node or *failure*

```
1  $Node \leftarrow$  a node with state  $InitialState(\mathcal{P})$  and  $Path - Cost = 0$ 
2  $Frontier \leftarrow$  a Priority queue ordered by  $Path - Cost$  with  $Node$  as the only element
3  $Explored \leftarrow$  an empty set
4 while  $Frontier$  is not empty do
5    $Node \leftarrow Pop(Frontier)$ 
6   if  $GoalState(\mathcal{P}, State(Node))$  then
7      $\_ return Solution(Node)$ 
8    $add State(Node)$  to  $Explored$ 
9   foreach  $action \in Actions(\mathcal{P}, State(Node))$  do
10     $Child \leftarrow ChildNode(\mathcal{P}, action, Node)$ 
11    if  $State(Child)$  not in  $Explored$  or  $Frontier$  then
12       $\_ add Child$  node to  $Frontier$ 
13    else
14      if  $State(Child)$  in  $Frontier$  with higher  $Path - Cost$  then
15         $\_ replace that Frontier node with Child$ 
16  $\_ return failure$ 
```

Please look at 6.4

Also, here we can see an example of the nodes that have been explored by the UCS on a problem that has a variable cost path function. The function returns values that are exponentially higher the closer the step takes you to a ghost. Thus, we can see in this example that UCS algorithm searches the search space by avoiding the ghosts and, thus, finding an optimal solution. Please look at 6.5

6.2.1.3 Depth First Search

The depth first search algorithm, as the name suggests, uses a strategy that always expands the node at the deepest level. When the algorithm does not find a node to expand from the current node, it *backs* up to the first node that still has unexpanded successors. The implementation of the DFS algorithm uses a FIFO queue (a stack) to store the nodes and expand them. The implementation is usually done recursively, with a function that calls itself on each of its children in turn.

It is obvious that the algorithm is not optimal, solely because it always chooses to expand the deepest node. The time complexity, in the worst case scenario, it's $\mathcal{O}(b^m)$, where m is the maximum depth of any node. This value can be greater than d . However, DFS has an advantage over BFS because of space complexity because the stack size can not increase more than the maximum depth of a node, m . Because of this, *DFS is the basic workhorse of many areas of AI*.

Implementation As before, we will not enter implementation details. The implementation can be seen below. Please look at 6.6

If we were to compare this algorithm to BFS by using the same example, we can clearly see that, in this case, it has found a solution by expanding fewer nodes than BFS. Thus, even

```

def uniformCostSearch(problem):
    # initialization part of the algorithm
    from util import PriorityQueue
    priorityQueue = PriorityQueue()
    priorityQueue.push((problem.getStartState(), [], 0))
    solution = []
    visited = []
    while not priorityQueue.isEmpty() and not solution:
        # take the element with the most priority and put it into elem
        elem = priorityQueue.pop()
        node = elem[0]
        # only when we pop the element we are sure we have reached it OPTIMALLY
        visited.append(node[0])
        priority = elem[1]
        # if the node found is a goal state, we return the solution
        if problem.isGoalState(node[0]):
            solution = node[1]
        else:
            # for each successor of the node, if it has not been visited (it has not been popped from the PQ), add it to the PQ
            for successor in problem.getSuccessors(node[0]):
                if successor[0] not in visited:
                    movementList = node[1][:]
                    movementList.append(successor[1])
                    # the element is "updated" in the queue:
                    # - if it's not present, add it
                    # - if it's present, update the priority in case the new one is lower
                    priorityQueue.update((successor[0], movementList), priority + successor[2])
    return solution

```

Figure 6.4: UCS algorithm

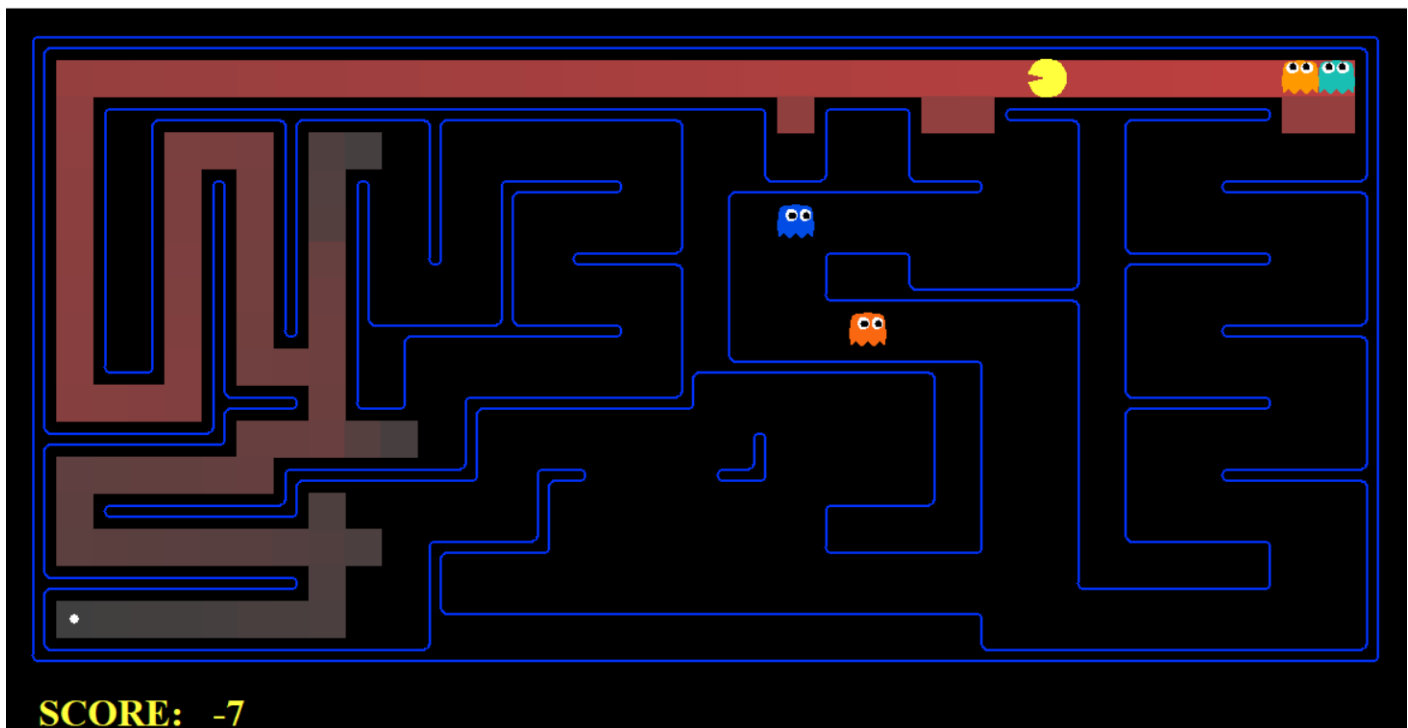


Figure 6.5: Ucs with non-constant step-cost function

Algorithm 3: *Depth First Search Algorithm*

Input: \mathcal{P} - the current problem to be solved

Output: $\langle solution\ path \rangle$, a set of nodes that represent the path from the root node to the goal node or *failure*

```
1 Function DFS(problem):
2   return recursiveDFS(MakeNode(InitialState(problem), problem))
3 Function recursiveDFS(node, problem):
4   if GoalState( $\mathcal{P}$ , State(Node)) then
5     return Solution(Node)
6   else
7     foreach action  $\in$  Actions( $\mathcal{P}$ , State(Node)) do
8       Child  $\leftarrow$  ChildNode( $\mathcal{P}$ , action, Node)
9       Result  $\leftarrow$  recursiveDFS(Child, Problem)
10      if Result  $\neq$  failure then
11        return Result
12  return failure
```

```
def depthFirstSearch(problem):
    # Initialization part of the search Algorithm
    from util import Stack
    stack = Stack()
    stack.push(problem.getStartState())

    visited = {problem.getStartState(): (None, None)}
    solution = depthFirstSearchRecursive(problem, stack, visited)

    return solution

def depthFirstSearchRecursive(problem, stack, visited):
    # take the top of the stack and put it into the node variable
    node = stack.pop()
    solution = []
    # if the node is a goal state, we compute the solution
    if problem.isGoalState(node):
        solution = computeSolution(node, visited)
    else:
        """
        DFS is a deterministic algorithm but behaves differently based on the order
        in which the neighbour list of a node is given. In some cases, certain path lengths may differ
        (as is the case for the MediumMaze) depending on whether the neighbour list is reversed or not.
        """
        # take each of the successors of the current node and add it to the stack if not visited yet
        for successor in problem.getSuccessors(node):
            if successor[0] not in visited and not solution:
                stack.push(successor[0])
                visited[successor[0]] = (node, successor[1])
                # each node is explored in the moment of discovery
                solution = depthFirstSearchRecursive(problem, stack, visited)
    return solution
```

Figure 6.6: DFS algorithm

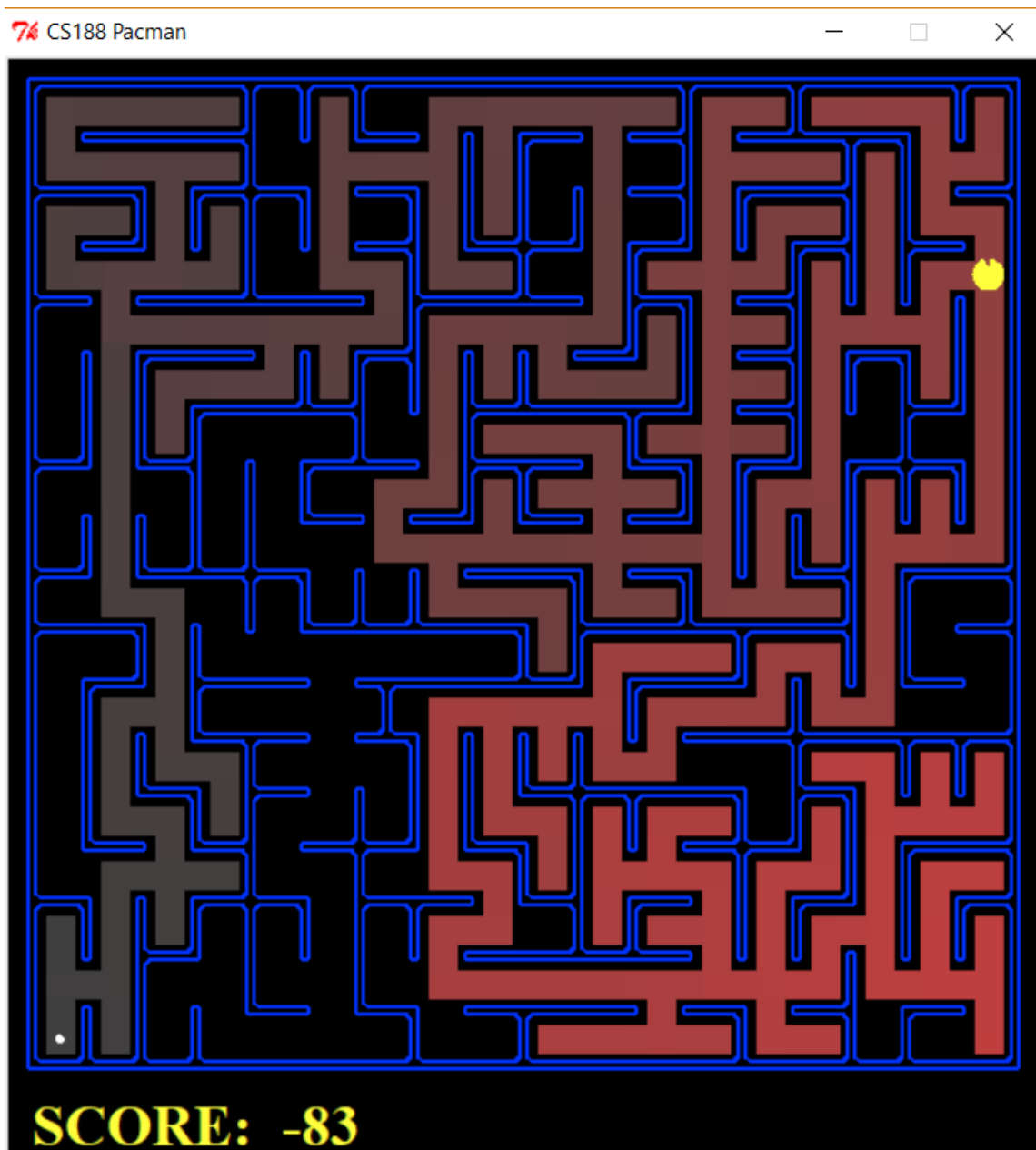


Figure 6.7: States explored by DFS

though the solution is not optimal, it's still a solution that was found in far less time and memory usage than the BFS solution. Please look at 6.7

6.2.1.4 Finding all the corners

In the project, after implementing the 3 uninformed strategies, we are asked to solve the problem of finding a path that passes through all the four corners of the maze by using the BFS algorithm. Even though, at first glance, it may seem impossible to solve the problem by altering the algorithm (for example, thinking of adding and checking for 4 subgoals that, when all 4 have been reached, will represent the goal of the problem), the solution is actually much more simpler than that. The secret lies in the state representation. In the initial implementation of BFS, a state was represented as a position on the 2d plane of the game but that encoding does not facilitate our purpose anymore. Actually, we can alter the state representation and

```

def getSuccessors(self, state):

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        # Add a successor state to the successor list if the action is legal
        # Here's a code snippet for figuring out whether a new position hits a wall:
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            cornersList = list(state[1])
            if (nextx, nexty) in cornersList:
                cornersList.remove((nextx, nexty))
            nextState = ((nextx, nexty), tuple(cornersList))
            cost = 1
            successors.append((nextState, action, cost))

    self._expanded += 1 # DO NOT CHANGE
    return successors

```

Figure 6.8: GetSuccessor method for the FourCorners problem

maintain the BFS algorithm as follows:

- Each state is now represented not only by the position it represents, but also by the corners that it has yet to reach. As it can be seen in the implementation below, the code that is altered in order to solve this problem is the *getSuccessors* method of the *CornersProblem* class. Thus, the state representation now is (position, [cornersToBeReached]). Whenever we generate a successor, we look if the respective node is a corner or not. If it is a corner, we remove that node from the corners list. Otherwise, we pass the corner list of the initial node to its successors.
- The goal check is also altered now. As I have said before, what we are trying to do is to achieve four subgoals that ultimately represent our goal. With the representation explained previously, the goal check can be done as just a simple check of the corners list. Basically, we have reached the goal when that list is empty.
- The initial state, besides the initial position, will also contain a list of all the four corners.

With this representation in mind, the problem can easily be solved by the BFS algorithm we have explained before. Of course, in order to be correct, the order of the corners in the list does not matter. What actually matters is the order in which the nodes are explored. Thus, a position that used to represent just a state before, can represent now exactly 16 states theoretically (practically, they are 15).

This problem is a good example of why choosing a good representation of the problem at hand is crucial in developing an efficient (and yet simple) solution. The algorithm has the same complexity as the BFS algorithm, only with a constant factor greater in magnitude (if we think about the nodes as positions), or the same constant factor (if we think about the nodes as states). The code can be seen below (I have skipped the implementation of the goal test and the creation of the initial state because they are quite trivial). Please look at 6.8

An example can be seen below. Please look at 6.9

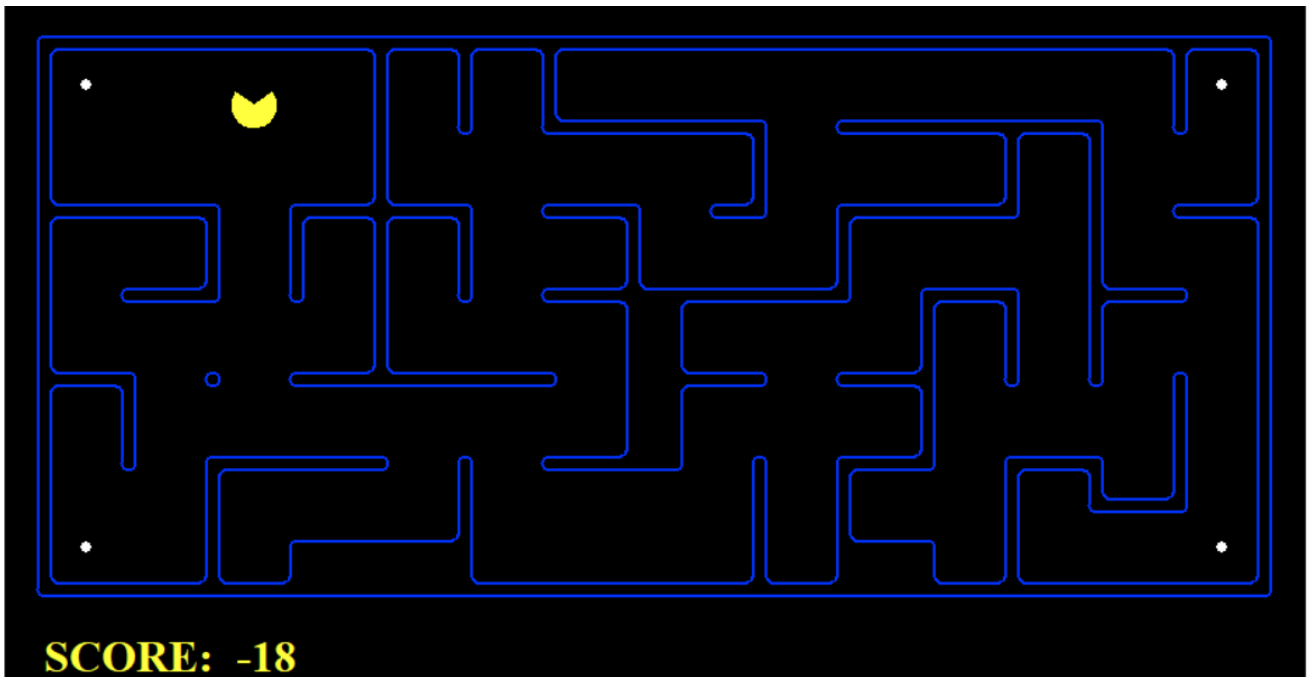


Figure 6.9: Example of the four corners problem

6.2.2 Informed Search Strategies

Informed search strategies use problem specific knowledge beyond the definition of the problem itself in order to find solutions more efficiently than an uninformed strategy. Informed strategies use an evaluation function that returns a cost estimate and the node with the lowest cost estimate is expanded further. Also, these strategies make heavy use of *heuristic* functions which are just an estimation of the cost from the current state to the goal state. **Heuristics always represent a lower bound of the actual cost.** The two algorithms, at their core, have the sole difference of how they chose to implement the evaluation function.

6.2.2.1 Greedy Best First Search

The GBF algorithm is an algorithm that always chooses to expand the closest node from the current state. Thus, it is quite easy to determine that the evaluation function it implements is just an heuristic function that determines what node it should expand next. In the project, the implementation of the heuristic was on the idea of always eating the closest food dot. The closest food dot (considering that the step cost was constant) was determined by using the BFS algorithm.

It is easy to see that the algorithm does not obtain an optimal solution. By always choosing to go to the node that it is closest to it, we can easily find an example that shows that it does not found an optimal solution.

Even though there are algorithms that use heuristics that really improve their performance whilst also finding an optimal solution, they still take a lot of time in big problems. Thus, sometimes, we may be interested in finding a solution rather than an optimal solution. I am not saying that a greedy algorithm can never find an optimal solution. I am just saying that a greedy algorithm, at its core, with no constraints or relaxations from a specific problem in a particular context can never promise optimality in a general context. However, greedy strategies are usually employed in applications to find a good-enough solution to the goal and, afterwards, a more complex algorithm that promises an optimal solution will find it based on

```

def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """

    # we use bfs algorithm to find the path to the closest food dot
    prob = AnyFoodSearchProblem(gameState)
    return search.bfs(prob)

```

Figure 6.10: Greedy Best First Search Algorithm

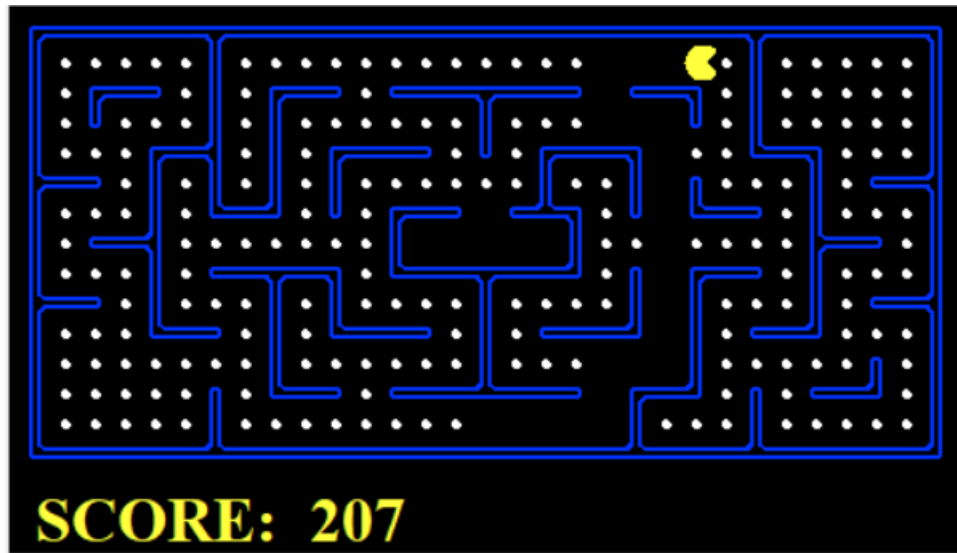


Figure 6.11: Example of a problem that implies collecting all the food

the solution provided by the greedy approach.

Also, we have chosen to not adapt the problem of the four corners to this problem because, even though that the worst case scenario of the BFS algorithm is as bad as the one of the GBF, GBF has higher chances of actually finding a nonoptimal solution because of the heuristic it uses. Thus, on average, GBF will perform way better than the adapted solution of the Corners Problem.

Implementation

The implementation was done as follows. We have implemented the class *AnyFoodSearchProblem* which is basically just a search problem but with the goal test changed. The goal now is just to encounter a food dot. By taking this problem and applying the bfs algorithm, we get a solution which is returned to the pacman. All of this subsearches are concatenated to form the final sequence of actions. The implementation can be quite trivial as it can be seen below. Please look at 6.10

An example can be seen below. Please look at 6.11

6.2.2.2 A* Search Algorithm

A* Star Search algorithm is basically the UCS algorithm but with an important change. The change I am talking about is the way in which the priority of a node/state is computed. Up until now, that priority was just the sole distance from the root node to current node, and, as

we have seen before, this assured optimality. A^* comes with the following improvement: the priority is now the value of an *evaluation function*. This evaluation function has the following value:

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

Where $g(n)$ represents the actual shortest path from the root node to the current node and $h(n)$ represents the approximation of the path from the current node to the goal state.

In order to achieve the optimality of the A^* algorithm, the heuristic needs to be:

- **Admissible**

An admissible heuristic is an heuristic that never overestimates the path from the current node to the goal state. As a consequence, $f(n)$ never overestimates the true cost of a solution through the current node n .

- **Consistent**

This requirement is stronger than the previous one. A heuristic $h(n)$ is consistent if, for every node n and every successor n' generated by any action a , the estimated cost of reaching the goal from n is not greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

$$h(n) \leq c(n, a, n') + h(n')$$

This creates a triangle inequality, which can be used to prove that **any consistent heuristic is also admissible**.

Thus, with a consistent heuristic, we get an optimal solution and drastically reduce the search space of the algorithm. The better the heuristic, the less it is explored. However, there is a tradeoff. In order to make a good heuristic, we need to carry additional computation, which may increase the time it takes to find the solution. One may think that a good, consistent heuristic may be to compute the actual distance to the goal state, but this would decrease the efficiency drastically. Thus, one has to find a balance in order to create a good heuristic. We will study two heuristics in the next two subsections.

6.2.2.3 Corner Problem Heuristic

In this subsection, we will develop an heuristic that enables the A^* search algorithm to solve the Corners Problem.

As we have said before, we need to find a balance when developing an heuristic. We want to get as close as we can to actual distance with as little as possible computation. Thus, the solution I propose is the following one:

- By making the heuristic to be the smallest of the lengths of the lines from the current node to the corners that have yet to be visited, we will not achieve the limits imposed by the autograder.
- By making the heuristic a little bit more realistic and, instead of the straight line between the two, we use the manhattan distance, we still do not reach the limits imposed by the autograder.
- The final heuristic I came up with that gives me a maximum score to this problem whilst also being consistent is the following one:

We start from the previously defined heuristic. From just a little bit of observation, we can see why the heuristic is not good enough. The problem is that we do not take into account the number of corners left to visit. Thus, a state with 3 corners left and a MD of 2 to the closest corner will be more priority than a node with 1 corner left and a MD of 3 to that corner. From this observation, we can do the following change:

```

corners = list(state[1]) # these are the corners that have yet to be discovered
heuristicValue = 0
pointOnGrid = state[0]
while corners:
    manDistances = []
    for corner in corners:
        manDistances.append(manhattanDistance(pointOnGrid, corner))
    heuristicValue = heuristicValue + min(manDistances)
    pointOnGrid = corners[manDistances.index(min(manDistances))]
    corners.remove(pointOnGrid)

return heuristicValue

```

Figure 6.12: Corner Problem Heuristic

- Compute the MD to the closest corner
- Move to that corner and compute the MD to the closest corner
- Repeat the last step until there are no more corners left
- The heuristic value is the sum of the distances computed previously

First thing we are going to do is trying to prove that the heuristic is consistent. It is obvious that, if the heuristic is consistent on a map with no walls, it is also consistent on a map that contains walls because it does not care about the walls. Thus, we imagine a simple map, with no corners at all, and pacman being somewhere in it. If pacman is on the edge, then the heuristic is consistent, because pacman just has to follow the layout of the map going from corner to corner (it does not matter where he is on the line. He will chose to go to the closest corner). Consistency is also ensured by the MD in this case (the path from one corner to the oposite corner can be anything as long as, at each step, you decrease the MD). If pacman is somewhere inside the map, the consistency can still be proved by the MD: he has to go to the closest edge, and apply the same reasoning as before (he can also go to the closest corner, because pacman moves according to the principle of MD. He can only go in 4 directions).

Thus, we have developed a consistent heuristic that uses few computations to resolve the problem.

Implementation

The implementation can be seen below. Please look at 6.12

6.2.2.4 Food Search Heuristic

In this subsection, we will develop an heuristic that allows the A* algorithm to collect the food from a map efficiently. My reasoning in developing the heuristic is the following one:

- First, I have thought to use the same heuristic as before but, after a short period of consideration, I have found out that it is quite easy to find an example that proves that our heuristic is inconsistent. Consider having 3 food dots on the OX axis, where pacman is on the position 0, and the food dots are at -2,1,7. The optimal solution would have a length of 11, but our heuristic says that it is 13. The heuristic was consistent solely because of the positioning of the *goals*.

- At this point, it was clear that this was not a good approach. My next approach was finding the closest food dot with MD, computing the line equation between that node and the current node, and somehow try to find a relationship between the length of this line and the number of wall it intersects. Needless to say, it was a dead end.
- Thus, I tried to find another approach. I have realised that the problem I had in finding the solution is that I was trying to keep the time it takes to compute the heuristic as low as possible, so I have decided to try and relax that constraint.
- The solution I have come up with (which even gives me the bonus points from the auto-grader) is the following one:

Find the closest food dot using the MD. For this food dot, compute the optimal solution from the current node to the food dot. Afterwards, follow this solution and, for each new position, count if the encountered position contains a food dot. Our final heuristic will be:

$$h(n) = \mathcal{Length}(solution) + (totalFoodDots - encounteredFoodDots)$$

Now, why is this heuristic consistent? I am sure that any person that thinks a little bit about this heuristic will see that it is a little bit counterintuitive to go to the closest food dot based on the MD, but actually, this part is trickier because our final goal is not going to the closest food dot, but eating all the food dots actually. We need to remember that an heuristic is a *lowerbound* of the actual distance from the current node to the goal node (the lower bound condition comes from the admissibility of the heuristic, which is implied by the consistency). No one says how low it has to be. Any heuristic that is consistent (consistency is required in order to achieve optimality) and returns a value strictly greater than 0 will make A* to have a far better performance than UCS.

Thus, for the moment let's just say that our heuristic consists solely of the length of the BFS solution. This length represents the length of the shortest path from the current node to that food dot. If that food dot is the only one, the heuristic holds true. If there are two food dots, the second food dot can either be on the path, and thus the heuristic returns the actual value of the length of the path to the goal, or the food dot is not on the path, case in which, considering that the step-cost function in this problem is 1, we will need to do extra steps in order to reach for the goal. In both cases, the heuristic holds true. It is easy to see that the explanation given before holds true for any number of food dots. Thus, for the moment, this heuristic is consistent and, implicitly, admissible.

One may wonder why does applying BFS to going to the *closest* (based on MD) food dot helps us in this case and achieves optimality but it doesn't behave the same in the case of the GBF search. The reason lies in the evaluation function. In GBF, we blindly follow just current percepts, without considering any past percepts (the evaluation function is the value of the heuristic). Here, however, the evaluation function is composed between the heuristic and the value of the shortest path between the root node and the current node, and this is the reason why this approach assures optimality and that one doesn't. The next thing one may wonder is why we have chosen to go to the MD closest food dot and not to the actual closest food dot. There are two reasons for this:

- First, the code itself was a problem. I would have needed to write code (classes and functions) in order to have a problem compatible to apply the algorithm. Considering that I was advised to compute the heuristic inside that function, I have chosen not to implement a problem class that could solve this.
- The second reason is the fact that I wanted to be able to add into the computation of the heuristic the magnitude of the food dots remaining (the second part of the heuristic). If we were to go to the actual closest food dot, then the number of


```

position, foodGrid = state
foodGridList = foodGrid.asList()
distances = []
for food in foodGridList:
    distances.append(manhatanDistance(position, food))
if distances:
    closest = foodGridList[distances.index(min(distances))]
    pathToClosest = mazeDistance(position, closest, problem.startingGameState)
    distanceToClosest = len(pathToClosest)

    # add all the points that we still have to reach and we are sure they are not reachable
    pointsLeftOut = len(foodGridList)
    x, y = position

    for action in pathToClosest:
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if (x, y) in foodGridList:
            pointsLeftOut = pointsLeftOut - 1

    return distanceToClosest + pointsLeftOut
else:
    return 0

```

Figure 6.13: Food Search Heuristic

remaining food dots will have always decrease by 1. Basically, it had no effect on the heuristic. But, on our current heuristic, remembering that it is consistent, we can make the following modification whilst also preserving consistency. This observation is extrapolated from the explanation involving the consistency of the heuristic from before. Each food dot that it is not present on the path computed by our heuristic is one more step we have to take (at least) to reach the goal. Thus, the new and final heuristic is consistent.

Implementation

The implementation can be seen below. Please look at 6.13

6.3 Adversial Search Techniques

Adversial search techniques are employed in competitive environments, in which the agent's goals are in conflict (these environments are often known as games). The environments we are going to deploy these techniques are deterministic and fully observable, in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite (that is, when one wins, the other one loses for sure).

Observation

Usually, an *evaluation function* in an adversarial search problem is a function that assigns a value to a state of the game, without taking into consideration any percept that lead us into this state (an important property of an evaluation function is that, if applied to the same state a number of times, it will always return the same value). However, the teachers that have created this project decided to use an evaluation function that also takes into consideration the action that is being done and, only at the end of the project, to implement an evaluation function of our own that uses only the current game state.

6.3.1 Reflex Agent

The first agent we are going to implement is a simple reflex agent that chooses, based on an evaluation function, an action that it thinks its the best to perform in that moment. The agent has a limited *vision*, that is, it only looks one steap ahead, which it can easily be seen from the fact that he does not perform well in a competitive environment.

The agent's pseudocode can be seen below.

Algorithm 4: Simple Reflex Agent

Input: \mathcal{P} - the current problem to be solved

Output: *failure* or *success*

```
1 while True do
2   if Success(problem) then
3     return success
4   if Failure(problem) then
5     return failure
6   CurrentState ← GetState(problem)
   Action ← GetBestAction(CurrentState)
7   problem ← NewProblem(problem, Action)
```

AS it can be seen, the algorithm is quite a basic one. At each step, we chose an action that we think it will help us, and we perform it. The algorithm stops when we have reached success or when we have failed. The evaluation function we have talked before is inside the *GetBestAction* function of the algorithm. This function looks at all the possible actions our agent can take (Up,Down,Left,Right,Stop), and, for each of them, calls an evaluation function with the currentGameState and the action. Based on the value of this evaluation function, the function returns the action that had the highest score (the action that is the most likely to succeed).

As with the heuristic, an evaluation function should not take a lot of time to compute, so we can not afford any expensive computations. Thus, we are going to build a simple evaluation function (to be honest, the purpose is satisfying the autograder, because I believe you could spend days in creating a close-to-perfection evaluation function), in which we are going to take into account some parameters, and each of these will contribute to the function based on some weight. In other words, the evaluation function was built by tweaking the values of the weights until we have reached our desired goal of performance.

I am just going to specify the way the parameters influence the value of the evaluation function, without giving all of the implementation details or the actual weights (they can be seen in the code).

The parameters we take into consideration are presented below.

- If the action that pacman wants to take is a *Stop* action (that is, he considers to do nothing), we will apply a penalty. We do not want the pacman to stay still only if absolutely necessary.
- If we intersect with a ghost that is not scared, we will apply a huge penalty.
- For each scared ghost, we will give a bonus.
- For each food dot, we compute the MD. This value is then inversed and added to the value of the function. Thus, pacman will be influenced to go to areas with more food rather than areas with less food.
- If pacman has eaten a capsule, we will give him a bonus.
- For each capsule that has not been eaten, we apply the same logic as the one regarding food dots.

Implementation

The actual implementation can be seen below. Please look at 6.14

6.3.2 MinMax Agent

In a normal search problem, the optimal solution is represented by a sequence of actions leading to a goal state, where only one agent has to say something about how the goal is reached. In a competitive environment, there are other agents that alter our solution to the goal.

Basically, in a competitive environment, each agent wants to maximize its performance. This can also be seen as minimizing the performance of the other agents. With this in mind, we can develop an algorithm called MinMax algorithm, which is an algorithm that basically look from the perspective of two players, one called Min, the other called Max, each one of them with the goal of minimizing and, respectively, maximize the score.

Therefore, the Max agent must find a contingent strategy, which specifies Max's move in the initial state, then Max's moves in the states resulting from every possible response by Min and so on.

One of the key ideas of the MinMax algorithm is that each of the two agents plays optimally. The tree that is created by applying the rule from before is called a game tree. In order to find an optimal strategy, we assign a *minmax* value to each of the nodes. The minmax value is computed as follows:

- If we have reached a terminal state (the game is a win or a lose in the pacman case), we assign an utility value (given by an utility function that defines a final numerical value for a game that ends in a terminal state).
- If it's Max's turn, we choose the maximum of all the values of the minmax function applied on the states we can reach by applying the possible actions from the current state.
- If it's Min's turn, we choose the minimum of all the values of the minmax function applied on the states we can reach by applying the possible actions from the current state.

After applying this function on all the nodes of the game tree, we can compute the minmax decision at the root node - we choose the optimal action for Max based on the game tree.

As we have said before, the algorithm assumes that both players play optimally. In other words, the algorithm maximizes the worst-case outcome for the MAX player. If MIN does not play

```

successorGameState = currentGameState.generatePacmanSuccessor(action)
newPos = successorGameState.getPacmanPosition()
newFood = successorGameState.getFood()
newGhostStates = successorGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

foodEaten = 100
capsuleEaten = 200
scaredGhostEaten = 300
intersectWithGhost = -10000

ifStopped = -1000

score = 0
if action == 'Stop':
    score += ifStopped

# ghostDistances = []
for ghostState in newGhostStates:
    if newPos == ghostState.getPosition():
        if newScaredTimes[newGhostStates.index(ghostState)] > 0:
            score += scaredGhostEaten
        else:
            score += intersectWithGhost

if currentGameState.hasFood(newPos[0], newPos[1]):
    score += foodEaten

for food in newFood.asList():
    score += 1.0 / self.manhattanDistance(food, newPos) * 10

for capsule in currentGameState.getCapsules():
    if newPos == capsule:
        score += capsuleEaten

for capsule in successorGameState.getCapsules():
    score += 1.0 / self.manhattanDistance(capsule, newPos) * 20

return score

```

Figure 6.14: Reflex agent evaluation function implementation

optimally, then it can easily be shown that MAX will do even better.

The complexity of the algorithm is bounded on the depth of the game tree. It can easily be seen that the algorithm traverses (whilst building the game tree) in a depth first manner. If we assume that the depth of the tree is m and there are b legal moves at each point, then the time complexity of the algorithm is $\mathcal{O}(b^m)$, and the space complexity is $\mathcal{O}(m)$ for an algorithm that generates actions one at a time.

Because of this, we need to take into consideration that the algorithm has to be applied at each step of pacman. Thus, we need to chose a relatively small depth of the game tree when we build it. For real games, of course, the time cost is totally impractical, but this algorithm can serve as the basis for the mathematical analysis of games and for more practical algorithms.

Algorithm 5: *MinMax algorithm*

Input: \mathcal{P} - the current state to be solved

Output: *returns an action*

```

1 Function Minimax-Decision(currentState):
2   finalAction  $\leftarrow$  None
3   foreach  $a \in$  Actions(state) do
4     action  $\leftarrow$  Min-Value(Result(state,  $a$ ))
5     finalAction  $\leftarrow$  Max(action, finalAction)
6   return finalAction
7 Function Max-Value(state):
8   if terminalTest(state) then
9     return Utility(state)
10  finalAction  $\leftarrow$   $-\infty$ 
11  foreach  $a \in$  Actions(state) do
12    action  $\leftarrow$  Min-Value(Result(state,  $a$ ))
13    finalAction  $\leftarrow$  Max(action, finalAction)
14  return finalAction
15 Function Min-Value(state):
16  if terminalTest(state) then
17    return Utility(state)
18  finalAction  $\leftarrow$   $\infty$ 
19  foreach  $a \in$  Actions(state) do
20    action  $\leftarrow$  Max-Value(Result(state,  $a$ ))
21    finalAction  $\leftarrow$  Min(action, finalAction)
22  return finalAction

```

Implementation In our implementation, however, we need to add more min levels (one for each ghost). Thus, for a gametree depth of 1, we will actually have n levels, one for each entity. So, basically, the depth of the tree from the previous explanation m is multiplied by the number of entities. And so on. In terms of depth numbers, if there are k entities (pacman + ghosts), and the current game state has depth -1 , we have the following:

- at depth mod $k == 0$, we have the pacman moves (the sole maximizer level)
- at depth mod $k == 1$, we have the 1st ghost moves (the 1st minimizer level)
- at depth mod $k == 2$, we have the 2nd ghost moves(the 2nd minimizer level) and so on.

```

def getAction(self, gameState):
    return self.maxTurn(-1, gameState)[1]

"""
MaxTurn represents the moment that the pacman moves. On this level, we always choose
the action that maximizes the profit.
"""

def maxTurn(self, actualDepth, gameState):

    actualDepth += 1
    # if we have reached a terminal node, return the utility value
    # return utility value also if we have surpassed the intended depth
    if gameState.isLose() or gameState.isWin() or actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:
        # get Pacman's legal actions
        legalActions = gameState.getLegalActions(0)
        utilityValues = []
        for action in legalActions:
            # for each possible action, we develop the gameState, and call the minTurn function for the 1st ghost
            val, action = self.minTurn(1, actualDepth, gameState.generateSuccessor(0, action))
            utilityValues.append(val)
        return (max(utilityValues), legalActions[utilityValues.index(max(utilityValues))])

```

Figure 6.15: MinMax algorithm part1

Thus, if the algorithm searches to a depth h , the search tree will actually have $k \cdot h$ levels. The implementation can be seen below. Please look at 6.15 and 6.16.

6.3.2.1 AlphaBeta pruning

The problem with MinMax search is that the number of game states it has to examine is exponential in the depth of tree. Even though we can't reduce the complexity, by making a simple observation, we can drastically reduce the gameTree by avoiding redundant computations. Take the following example. Please look at 6.17

Assume we are in node E, a maximizing node. Node B, a minimizing node, already has an option of value 5. Thus, in node E, when we encounter the value 6, we can stop because we know that node E has a minimum output of 6, which will not influence the decision of node B. This can be implemented by using 2 variables that represent an upper and lower boundary. It is important to say that AlphaBeta pruning returns the exact same solution as MinMax, but with a lot less computation.

Implementation

Implementation can be seen below. Please look at 6.18 and 6.19

6.3.2.2 Expectiminimax algorithm

Because of the way MinMax algorithm works (and, implicitly, AlphaBeta), we encounter two problems:

- First problem is that MinMax algorithm may sometimes get stuck because of its limited *vision* in the future.
- Second problem is that MinMax algorithm will sometimes lead the Pacman entity to commit suicide. There are scenarios where, if the algorithm assumes that the other entities

```
def minTurn(self, ghostAgentIndex, actualDepth, gameState):

    # if we have reached a terminal node, stop
    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    # if we have expanded the gameState for every ghost, go on to the next "depth" level
    # next depth level is represented by pacman performing a new move
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        utilityValues = []
        for action in legalActions:
            # for each possible action, we develop the gameState, and call the minTurn function for the next ghost
            val, action = self.minTurn(ghostAgentIndex + 1, actualDepth,
                                       gameState.generateSuccessor(ghostAgentIndex, action))
            utilityValues.append(val)
        return (min(utilityValues), legalActions[utilityValues.index(min(utilityValues))])
```

Figure 6.16: MinMax algorithm part2

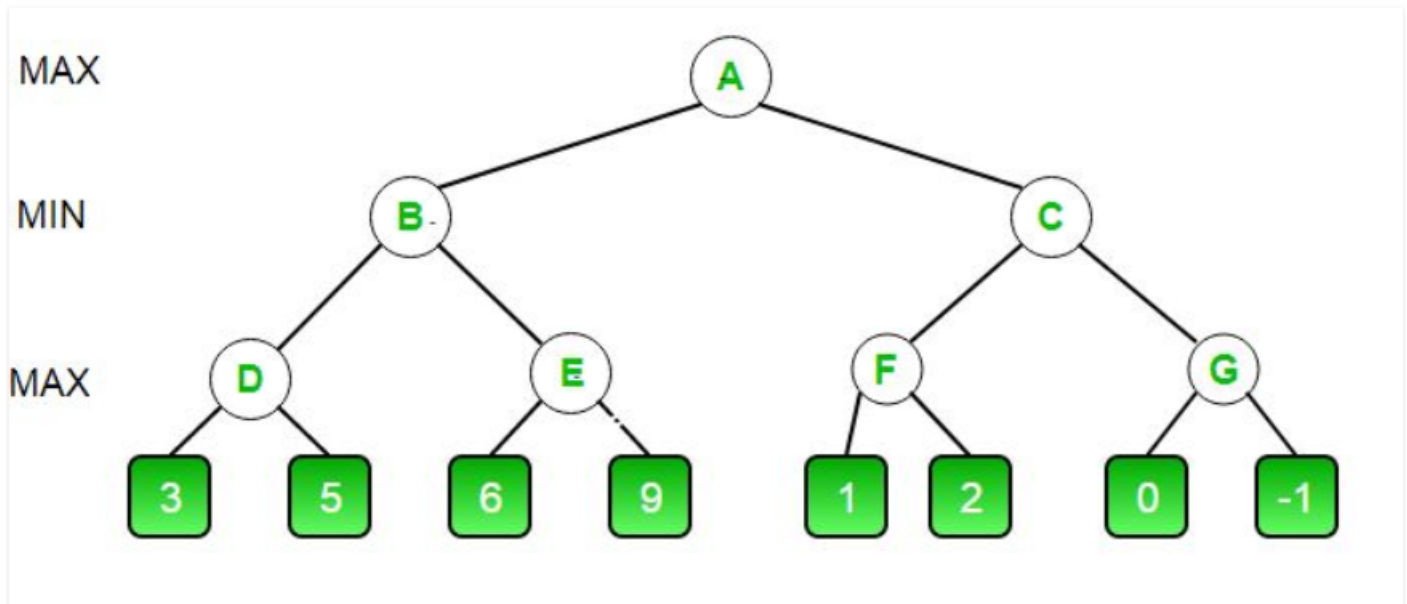


Figure 6.17: AlphaBeta pruning example

Algorithm 6: *AlphaBeta pruning algorithm*

Input: \mathcal{P} - the current state to be solved

Output: *returns an action*

```
1 Function Alpha-Beta Search(currentState):
2   finalAction  $\leftarrow$  None foreach  $a \in \text{Actions}(\text{state})$  do
3     action  $\leftarrow$  Min-Value(Result(state,  $a$ ),  $-\text{infinity}$ , infinity)
4     finalAction  $\leftarrow$  Max(action, finalAction)
5   return finalAction

6 Function Max-Value(state,  $\alpha$ ,  $\beta$ ):
7   if terminalTest(state) then
8     return Utility(state)
9   finalAction  $\leftarrow$   $-\text{infinity}$ 
10  foreach  $a \in \text{Actions}(\text{state})$  do
11    action  $\leftarrow$  Min-Value(Result(state,  $a$ ),  $\alpha$ ,  $\beta$ )
12    finalAction  $\leftarrow$  Max(action, finalAction)
13    if finalAction  $\geq \beta$  then
14      return finalAction
15     $\alpha \leftarrow$  Max( $\alpha$ , finalAction)
16  return finalAction

17 Function Min-Value(state,  $\alpha$ ,  $\beta$ ):
18  if terminalTest(state) then
19    return Utility(state)
20  finalAction  $\leftarrow$  infinity
21  foreach  $a \in \text{Actions}(\text{state})$  do
22    action  $\leftarrow$  Max-Value(Result(state,  $a$ ))
23    finalAction  $\leftarrow$  Min(action, finalAction) if finalAction  $\leq \alpha$  then
24      return finalAction
25     $\beta \leftarrow$  Min( $\beta$ , finalAction)
26  return finalAction
```

```

def getAction(self, gameState):
    """
    Returns the minimax action using self.depth and self.evaluationFunction
    """
    # alpha = value of the best choice found so far for the maximizer
    # beta = value of the best choice found so far for the minimizer
    # this values are consistent along a path from the root node to any node in the search tree
    # "best choice" is relative with respect to the agent's perspective from which we are looking
    self.alpha = -10000
    self.beta = 10000
    return self.maxTurn(-1, gameState, -10000, 10000)[1]

def maxTurn(self, actualDepth, gameState, alpha, beta):

    actualDepth += 1
    # if we have reached a terminal node, return the utility value
    # return utility value also if we have surpassed the intended depth
    if gameState.isLose() or gameState.isWin() or actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:
        # get Pacman's legal actions
        legalActions = gameState.getLegalActions(0)
        # best option so far
        v = -10000
        bestAction = None
        for action in legalActions:
            val, action1 = self.minTurn(1, actualDepth, gameState.generateSuccessor(0, action), alpha, beta)
            # if we have found an action better than one we have found for this level, we update it
            if v < val:
                v = val
                bestAction = action
            # if the action found is greater than the best option for the minimizer, we stop
            if v > beta:
                return (v, action)
            # otherwise, we update the alpha value ( in case it can be updated)
            alpha = max(v, alpha)
        return (v, bestAction)

```

Figure 6.18: AlphaBeta pruning algorithm part1

```

def minTurn(self, ghostAgentIndex, actualDepth, gameState, alpha, beta):

    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState, alpha, beta)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        # the best options so far
        v = 10000
        bestAction = None
        for action in legalActions:
            val, action1 = self.minTurn(ghostAgentIndex + 1, actualDepth,
                                         gameState.generateSuccessor(ghostAgentIndex, action), alpha, beta)
            # if we have found an action better than one we have found for this level, we update it
            if val < v:
                v = val
                bestAction = action
            # if the action found is greater than the best option for the minimizer, we stop
            if v < alpha:
                return (v, action)
            # otherwise, we update the beta value ( in case it can be updated)
            if v < beta:
                beta = v
        return (v, bestAction)

```

Figure 6.19: AlphaBeta pruning algorithm part2

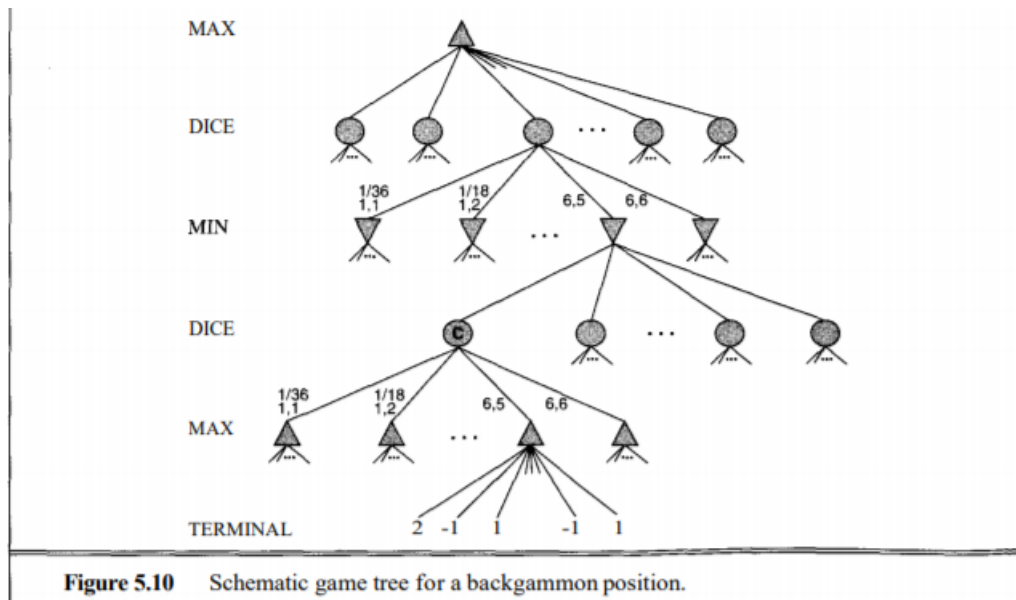


Figure 6.20: Example of how the game tree of expectiminimax algorithm

play optimally, that would be more optimal to commit suicide in order to preserve the point it has. However, in an environment where the other entities may make random moves, it would be better to deploy the algorithm that takes into account the probability to happen of each action.

For the second problem, we may deploy the Expectiminimax algorithm. This algorithm is usually deployed in stochastic games, games in which we have unpredictability, such as throwing dices. In order to improve the MinMax algorithm to take into account, we add one more layer of nodes called *Chance* nodes.

Imagine if, for example, we throw 2 dices. For each of the values from 2 to 12, we have a probability for it to occur. For each of these values, we may make a number of moves. Thus, our game tree would look like this. Please look at 6.20

The minimax value is also updated to contain the chance nodes. Thus, when we compute the expectiminimax value of a chance node, we will compute the expected value. In other words, we compute the sum of the probability of a dice output for example with the value returned by the expectiminimax value of that certain action.

A rough explanation for the time complexity of this algorithm is that, besides doing all the computation and decisions of the MinMax algorithm, it also takes into account all the possible dice outcomes for example. Thus, the time complexity is $\mathcal{O}(b^m * n^m)$, where n is the number of possible outcomes of rolling a dice.

Even if the complexity of MinMax algorithm is far smaller than the complexity of this one. However, MinMax can not be deployed in problems where uncertainty has something to say.

Implementation

For our game, the probability of an action that the ghost agent can choose is $1/\text{numberOfLegalActions}$. Each level of choice nodes is made of one node actually, thus we can combine the choice node with minLevel. Thus, instead of choosing the minimum of a single value, the minLevel will actually compute the expected value. I have chosen to implement it using only one choice node per choice level because each action has exactly one immediate outcome.

When introducing this algorithm, I have talked about the fact that MinMax may lead Pacman to commit suicide in order to preserve its score because it thinks that it's safer that way. One of the experiments we will talk a little bit later proves this. On a trap map, Pacman always commits suicide. However, the Expectimax algorithm, deployed on the same map, has a winning

```

def getAction(self, gameState):
    return self.maxTurn(-1, gameState)[1]

def maxTurn(self, actualDepth, gameState):

    actualDepth += 1
    # if we have reached a terminal node, return the utility value
    # return utility value also if we have surpassed the intended depth
    if gameState.isLose() or gameState.isWin() or actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:
        # get Pacman's legal actions
        legalActions = gameState.getLegalActions(0)
        utilityValues = []
        for action in legalActions:
            val, action = self.minTurn(1, actualDepth, gameState.generateSuccessor(0, action))
            utilityValues.append(val)
        return (max(utilityValues), legalActions[utilityValues.index(max(utilityValues))])

def minTurn(self, ghostAgentIndex, actualDepth, gameState):

    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        utilityValues = []
        for action in legalActions:
            val, action = self.minTurn(ghostAgentIndex + 1, actualDepth,
                                      gameState.generateSuccessor(ghostAgentIndex, action))
            utilityValues.append(val)
        # here is the expected value computed
        return (sum(utilityValues) * 1.0 / len(legalActions), None)

```

Figure 6.21: Expectiminimax algorithm

chance of roughly 50 percent. Please look at 6.21.

6.3.2.3 Evaluation function

The last step of this project is developing an evaluation function. This is the evaluation function we have talked at the beginning of the **Adversial Search**, which assigns a value to a game state without taking into consideration the action that was took in order to get to this game state.

The same reasoning is applied as the one applied beforehand. Thus, we will explain all the parameters we take into consideration. The values and the way the parameters influence the final value have been tweaked until the autograder was satisfied.

The evaluation function written for the reflex agent is good for that agent, but not really good for the minimax (and its acolytes) algorithm. One of the reasons, for example, is that it takes into account the action performed. Thus, I have chosen an approach that starts from that evaluation function, but improves some aspects. The parameters we take into account are the following:

- If the current game state is a win, return a really high value
- If the current game state is a lose, return a really low value
- The score is computed by subtracting values. Even if this is counterintuitive, the reason for this is that this is a really good way to represent some values, such as distance to closest food dot (the smaller the value, the less we subtract).
- If we have eaten a food dot, we can see this as follows: subtract the number of food dots from the score. If, for example, from 4 actions, only one of them eats a food dot, that means that the score of that action is greater with at least 1 from all the scores of the other actions (taking into account only the food dots, with no other constraints). This quantity is weighted with a value (in our case it's a 10) to express how important eating a food dot is.
- We also take into account the distance to the closest food dot. This distance is computed by using the BFS algorithm. This value is subtracted from the score. Thus, the smaller the distance, the higher the score is.
- We also look at the ghost states, but a little bit different than previously. We are going to compensate him if there are ghosts that are scared (implicitly, with this constraint we encourage eating capsules). Also, we are going to give pacman a penalty based on the distance from him to the ghosts (the closer he is, the worse). This was a tricky thing to do, because we do not want to have to much computation time spent in this evaluation function (so I have chosen to use manhattan distance rather than actual distance to the ghosts), and we also have to find a good way to quantitatively represent this information.
- The last part was the hardest thing to do. I have first started with the way I represented the information in the previous evaluation function, but this thing made pacman to have a low score. By varying the respective values, it still did not prove to be more efficient. So i have decided to chose another approach. We still want to give him a penalty based on the distance (the closer, the worse), but we will do it by subtracting a small part of the distance, rather than working with the inverse of the distance.

With a little bit of tweaking all the quantitative values, I have come up with an evaluation function that the autograder evaluates to the maximum grade (based on 10 games played, pacman has a score slightly over 1000, but never less). Please look at 6.22 and 6.23.

6.4 General Implementation Details

The implementation of the types of search strategies has been done on the base projects provided by the berkeley university, mainly:

- General Search Strategies
- Adversial Search Strategies

On each of these projects, I had to implement methods that were going to be used by the program written by the teachers from berkeley. As I have mentioned before, I have decided to not explain the concrete implementation of the python code, mostly because the algorithms are not defined by the way they are implemented (of course, they have to be implemented efficiently). However, besides the code being self-explanatory with suggestive variable names, I have also included comments to all the code I have written. The projects will be attached with this documentation. Also, the code I have written will be included in the appendix.

```

def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).

    DESCRIPTION: <write something here so we know what you did>
    """
    if currentGameState.isWin():
        return 10000
    elif currentGameState.isLose():
        return -10000

    pacmanPosition = currentGameState.getPacmanPosition()
    foodPositionMatrix = currentGameState.getFood()
    wallPositionMatrix = currentGameState.getWalls()
    ghostStates = currentGameState.getGhostStates()

    score = -10 * len(foodPositionMatrix.asList())

    score -= breadthFirstSearch(pacmanPosition, wallPositionMatrix, foodPositionMatrix.asList())

    for ghostState in ghostStates:
        if ghostState.scaredTimer > 0:
            score += 10
        else:
            score -= 0.2 * manhattanDistance(pacmanPosition, ghostState.getPosition())

    return score

```

Figure 6.22: Evaluation function part1

6.5 Grading

During the development of the code, I have relied on the autograder provided by the teachers at Berkeley. For both of the projects, I have succeeded in receiving maximum points (for the first project, I have also received the bonus point). Please look at 6.24 and 6.25.

6.6 Graphs and Experiments

It is quite impossible to explain the experiments and the scenarios I have tested the algorithms in without a live presentation. However, in the project that I have included, I have a lot of setups done using the maps and suggestions offered by the teachers from Berkeley. Thus, if the project is opened in Pycharm, those experiments can be tested. These setups look like this. Please look at figure 6.26.

Regarding the graphs, I have decided to not include any because all of the algorithms have roughly the same time complexity, that is, *exponential time complexity*.

```

def breadthFirstSearch(pacmanPosition, wallPosition, goalPosition):
    from util import Queue
    queue = Queue()
    queue.push(pacmanPosition)
    visited = {pacmanPosition: (None, None)}
    solution = []
    while not queue.isEmpty() and not solution:
        node = queue.pop()
        if node in goalPosition:
            solution = computeSolution(node, visited)
        if not solution:
            for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
                x, y = node
                dx, dy = Actions.directionToVector(direction)
                nextx, nexty = int(x + dx), int(y + dy)
                if not wallPosition[nextx][nexty]:
                    if (nextx, nexty) not in visited:
                        visited[(nextx, nexty)] = (node, direction)
                        queue.push((nextx, nexty))
    return len(solution)

def computeSolution(node, visited):
    solution = []
    while True:
        if visited[node][1]:
            solution.append(visited[node][1])
            node = visited[node][0]
        else:
            break
    # solution has to be reversed because it is computed starting from the goal
    solution.reverse()
    return solution

def manhattanDistance(pos1, pos2):
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

```

Figure 6.23: Evaluation function part2

```

Provisional grades
=====
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3
-----
Total: 26/25

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

```

Figure 6.24: Autograder result for the General Search project

```
Provisional grades
=====
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 5/5
Question q5: 6/6
-----
Total: 25/25

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Figure 6.25: Autograder result for the Adversial Search project

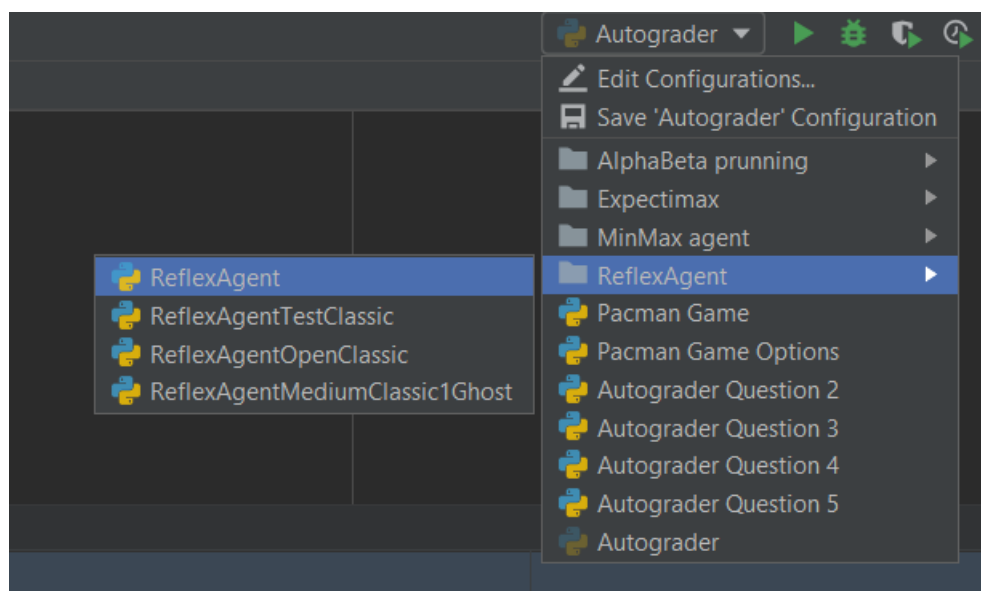


Figure 6.26: Setup example

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.

```
def computeSolution(node, visited):
    solution = []
    while True:
        if visited[node][1]:
            solution.append(visited[node][1])
            node = visited[node][0]
        else:
            break
    # solution has to be reversed because it is computed starting from the goal
    #solution.reverse()
    return solution

def depthFirstSearch(problem):
    # Initialization part of the search Algorithm
    from util import Stack
    stack = Stack()
    stack.push(problem.getStartState())

    """
    This dictionary will help both in reconstructing the solution and
    in keeping track of the already visited nodes
    With the help of this dictionary we also avoid loops
    Structure: dict[node] = (parent, move)
    """

    visited = { problem.getStartState(): (None, None) }
    solution = depthFirstSearchRecursive(problem, stack, visited)

    return solution

def depthFirstSearchRecursive(problem, stack, visited):
```



```

# take the top of the stack and put it into the node variable
node = stack.pop()
solution = []
# if the node is a goal state, we compute the solution
if problem.isGoalState(node):
    solution = computeSolution(node, visited)
else:
    for successor in problem.getSuccessors(node):
        if successor[0] not in visited and not solution:
            stack.push(successor[0])
            visited[successor[0]] = (node, successor[1])
            # each node is explored in the moment of discovery
            solution = depthFirstSearchRecursive(problem, stack, visited)
return solution

def breadthFirstSearch(problem):
    # initialization part of the algorithm
    from util import Queue
    queue = Queue()
    queue.push(problem.getStartState())
    visited = { problem.getStartState(): (None, None) }
    solution = []
    while not queue.isEmpty() and not solution:
        # extract a node from the queue
        node = queue.pop()
        # if the node is a goal state, then compute solution
        if problem.isGoalState(node):
            solution = computeSolution(node, visited)
        if not solution:
            for successor in problem.getSuccessors(node):
                if successor[0] not in visited:
                    visited[successor[0]] = (node, successor[1])
                    queue.push(successor[0])
    return solution

def uniformCostSearch(problem):
    # initialization part of the algorithm
    from util import PriorityQueue
    priorityQueue = PriorityQueue()
    priorityQueue.push((problem.getStartState(), []), 0)
    solution = []
    visited = []
    while not priorityQueue.isEmpty() and not solution:
        # take the element with the most priority and put it into elem
        elem = priorityQueue.pop()
        node = elem[0]
        # only when we pop the element we are sure we have reached it OPTIMALLY
        visited.append(node[0])
        priority = elem[1]

```

```

# if the node found is a goal state, we return the solution
if problem.isGoalState(node[0]):
    solution = node[1]
else:
    for successor in problem.getSuccessors(node[0]):
        if successor[0] not in visited:
            movementList = node[1][:]
            movementList.append(successor[1])
            # the element is "updated" in the queue:
            # - if it's not present, add it
            priorityQueue.update((successor[0], movementList), \
                                priority + successor[2])
return solution

```

```

def aStarSearch(problem, heuristic = nullHeuristic):
    from util import PriorityQueue
    priorityQueue = PriorityQueue()
    priorityQueue.push((problem.getStartState(), [], 0), \
        heuristic(problem.getStartState(), problem))
    solution = []
    visited = []
    while not priorityQueue.isEmpty() and not solution:
        elem = priorityQueue.pop()
        node = elem[0]
        # only when we pop the element we are sure we have reached it OPTIMALLY
        visited.append(node[0])
        actualDistance = node[2]
        if problem.isGoalState(node[0]):
            solution = node[1]
        else:
            for successor in problem.getSuccessors(node[0]):
                if successor[0] not in visited:
                    movementList = node[1][:]
                    movementList.append(successor[1])
                    priorityQueue.update((successor[0], \
                        movementList, actualDistance + successor[2]), \
                        actualDistance + successor[2] + \
                        heuristic(successor[0], problem))
    return solution

```

```

def getStartState(self):
    return (self.startingPosition, self.corners)

```

```

def isGoalState(self, state):
    return len(state[1]) == 0

```

```

def getSuccessors(self, state):

```

```

successors = []
for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,\
Directions.WEST]:
    x, y = state[0]
    dx, dy = Actions.directionToVector(action)
    nextx, nexty = int(x + dx), int(y + dy)
    if not self.walls[nextx][nexty]:
        cornersList = list(state[1])
        if (nextx, nexty) in cornersList:
            cornersList.remove((nextx, nexty))
        nextState = ((nextx, nexty), tuple(cornersList))
        cost = 1
        successors.append((nextState, action, cost))

self._expanded += 1 # DO NOT CHANGE
return successors

def cornersHeuristic(state, problem):
    walls = problem.walls
    corners = list(state[1])
    heuristicValue = 0
    pointOnGrid = state[0]
    while corners:
        manDistances = []
        for corner in corners:
            manDistances.append(manhatanDistance(pointOnGrid, corner))
        heuristicValue = heuristicValue + min(manDistances)
        pointOnGrid = corners[manDistances.index(min(manDistances))]
        corners.remove(pointOnGrid)

    return heuristicValue

def foodHeuristic(state, problem):
    position, foodGrid = state
    foodGridList = foodGrid.asList()
    distances = []
    for food in foodGridList:
        distances.append(manhatanDistance(position, food))
    if distances:
        closest = foodGridList[distances.index(min(distances))]
        pathToClosest = mazeDistance(position, closest,\
problem.startingGameState)
        distanceToClosest = len(pathToClosest)

    pointsLeftOut = len(foodGridList)
    x, y = position

    for action in pathToClosest:

```

```

        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if (x, y) in foodGridList:
            pointsLeftOut = pointsLeftOut - 1

        return distanceToClosest + pointsLeftOut

    else:
        return 0

def findPathToClosestDot(self, gameState):
    prob = AnyFoodSearchProblem(gameState)
    return search.bfs(prob)

def evaluationFunction(self, currentGameState, action):
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
    foodEaten = 100
    capsuleEaten = 200
    scaredGhostEaten = 300
    intersectWithGhost = -10000

    ifStopped = -1000

    score = 0
    if action == 'Stop':
        score += ifStopped

    # ghostDistances = []
    for ghostState in newGhostStates:
        if newPos == ghostState.getPosition():
            if newScaredTimes[newGhostStates.index(ghostState)] > 0:
                score += scaredGhostEaten
            else:
                score += intersectWithGhost
    if currentGameState.hasFood(newPos[0], newPos[1]):
        score += foodEaten

    for food in newFood.asList():
        score += 1.0 / self.manhattanDistance(food, newPos) * 10

    for capsule in currentGameState.getCapsules():
        if newPos == capsule:
            score += capsuleEaten

    for capsule in successorGameState.getCapsules():
        score += 1.0 / self.manhattanDistance(capsule, newPos) * 20

```

```

    return score

def getAction(self, gameState):
    return self.maxTurn(-1, gameState)[1]

def maxTurn(self, actualDepth, gameState):

    actualDepth += 1
    if gameState.isLose() or gameState.isWin() or\
    actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:
        # get Pacman's legal actions
        legalActions = gameState.getLegalActions(0)
        utilityValues = []
        for action in legalActions:
            val, action = self.minTurn(1, actualDepth, \
            gameState.generateSuccessor(0, action))
            utilityValues.append(val)
        return (max(utilityValues),\
        legalActions[utilityValues.index(max(utilityValues))])

def minTurn(self, ghostAgentIndex, actualDepth, gameState):
    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        utilityValues = []
        for action in legalActions:
            val, action = self.minTurn(ghostAgentIndex + 1,\
            actualDepth,gameState.generateSuccessor\
            (ghostAgentIndex, action))
            utilityValues.append(val)
        return (min(utilityValues),legalActions\
        [utilityValues.index(min(utilityValues))])

def getAction(self, gameState):
    self.alpha = -10000
    self.beta = 10000
    return self.maxTurn(-1, gameState, -10000, 10000)[1]

def maxTurn(self, actualDepth, gameState, alpha, beta):

    actualDepth += 1
    if gameState.isLose() or gameState.isWin() or actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:

```

```

    # get Pacman's legal actions
    legalActions = gameState.getLegalActions(0)
    # best option so far
    v = -10000
    bestAction = None
    for action in legalActions:
        val, action1 = self.minTurn(1, actualDepth,\
            gameState.generateSuccessor(0, action), alpha, beta)
        if v < val:
            v = val
            bestAction = action
        if v > beta:
            return (v, action)
        alpha = max(v, alpha)
    return (v, bestAction)

def minTurn(self, ghostAgentIndex, actualDepth, gameState, alpha, beta):

    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState, alpha, beta)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        # the best options so far
        v = 10000
        bestAction = None
        for action in legalActions:
            val, action1 = self.minTurn(ghostAgentIndex + 1,\
                actualDepth,gameState.generateSuccessor\
                (ghostAgentIndex, action), alpha, beta)
            if val < v:
                v = val
                bestAction = action
            if v < alpha:
                return (v, action)
            if v < beta:
                beta = v
        return (v, bestAction)
def getAction(self, gameState):
    return self.maxTurn(-1, gameState)[1]

def maxTurn(self, actualDepth, gameState):

    actualDepth += 1
    if gameState.isLose() or gameState.isWin() or actualDepth >= self.depth:
        return (self.evaluationFunction(gameState), None)
    else:
        legalActions = gameState.getLegalActions(0)
        utilityValues = []

```

```

        for action in legalActions:
            val, action = self.minTurn(1, actualDepth, \
                gameState.generateSuccessor(0, action))\
                utilityValues.append(val)
        return (max(utilityValues),\
            legalActions[utilityValues.index(max(utilityValues))])

def minTurn(self, ghostAgentIndex, actualDepth, gameState):

    if gameState.isLose() or gameState.isWin():
        return (self.evaluationFunction(gameState), None)
    if ghostAgentIndex > gameState.getNumAgents() - 1:
        return self.maxTurn(actualDepth, gameState)
    else:
        legalActions = gameState.getLegalActions(ghostAgentIndex)
        utilityValues = []
        for action in legalActions:
            val, action = self.minTurn(ghostAgentIndex + 1,\
                actualDepth,gameState.generateSuccessor\
                (ghostAgentIndex, action))
            utilityValues.append(val)
        return (sum(utilityValues) * 1.0 / len(legalActions), None)

def betterEvaluationFunction(currentGameState):
    if currentGameState.isWin():
        return 10000
    elif currentGameState.isLose():
        return -10000

    pacmanPosition = currentGameState.getPacmanPosition()
    foodPositionMatrix = currentGameState.getFood()
    wallPositionMatrix = currentGameState.getWalls()
    ghostStates = currentGameState.getGhostStates()

    score = -10 * len(foodPositionMatrix.asList())

    score -= breadthFirstSearch(pacmanPosition, \
        wallPositionMatrix, foodPositionMatrix.asList())

    for ghostState in ghostStates:
        if ghostState.scaredTimer > 0:
            score += 10
        else:
            score -= 0.2 * manhatanDistance(pacmanPosition,\
                ghostState.getPosition())

    return score

def breadthFirstSearch(pacmanPosition, wallPosition, goalPosition):

```

```

from util import Queue
queue = Queue()
queue.push(pacmanPosition)
visited = { pacmanPosition: (None, None) }
solution = []
while not queue.isEmpty() and not solution:
    node = queue.pop()
    if node in goalPosition:
        solution = computeSolution(node, visited)
    if not solution:
        for direction in [Directions.NORTH, Directions.SOUTH,\
        Directions.EAST, Directions.WEST]:
            x, y = node
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not wallPosition[nextx][nexty]:
                if (nextx, nexty) not in visited:
                    visited[(nextx, nexty)] = (node, direction)
                    queue.push((nextx, nexty))
return len(solution)

```

```

def computeSolution(node, visited):
    solution = []
    while True:
        if visited[node][1]:
            solution.append(visited[node][1])
            node = visited[node][0]
        else:
            break
    # solution has to be reversed because it is computed starting from the goal
    solution.reverse()
    return solution

```


Appendix B

Quick technical guide for running your project

The project just has to be opened in Pycharm. All the setups for the experiment were already configured by me.

Bibliography

- [1] Charles T Batts. A beamer tutorial in beamer. *The University of North Carolina at Greensboro, Department of Computer Science*, 2007.
- [2] Kai-Tai Fang, Runze Li, and Agus Sudjianto. *Design and modeling for computer experiments*. CRC Press, 2005.
- [3] Andrew Mertz and William Slough. Beamer by example. *The PracTEX Journal*, 4, 2005.
- [4] Russell, Stuart Jonathan and Norvig, Peter and Canny, John F and Malik, Jitendra M and Edwards, Douglas D. *Artificial Intelligence: a modern approach*, 3, 2003. Prentice hall Upper Saddle River.

Intelligent Systems Group

