Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name:Beáta Keresztes and Borbála Fazakas
Group:30432
Email:keresztesbeata00@yahoo.com, fazakasbori@gmail.com

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

Contents

Table 1: Lab scheduling

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

0.0.0.0.1 Lab organisation.

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

Chapter 1

A1: Search

1.1 The Hunger Games Search Probem

1.1.1 Problem Statement

1.1.1.1 Overview

This project is an extension of the PositionSearchProblem in the PacMan framework known from the laboratory activities. In this updated version of the game, PacMan doesn't only need to reach a specific goal position in the grid, but it also has to take into account additional constraints: PacMan may die of hunger if it doesn't eat enough food on its way to the goal! Thus, PacMan's objective is to reach the goal position on the shortest possible path while making sure that it always has enough energy for the next step.

1.1.1.2 Background

The framework used in the project was developed at the University of California, Berkeley, and the entire code is available at this link.

In the following sections, we will assume that the reader is familiar with the framework and objective and the rules imposed by the PositionSearchProblem.

1.1.1.3 Game rules

This updated version of the well-known PacMan game comes with a few constraints and alternations that make things even more interesting.

1. *What are we fighting for?*
   The goal of the game is to find the exit from the maze on the shortest possible path while keeping PacMan alive. The exit gate can be situated on any of the sides of the maze, and the game ends when PacMan finally reaches it.

2. *To live or not to live?*
   The new the constraint imposed on the player by this version of the game is that PacMan has become *"mortal"*. It goes without saying that one needs food in order to survive and the same applies to our PacMan as well. It needs energy to continue searching in the maze, and this energy can be obtained only by eating a food dot. Similarly, every step that PacMan takes drains from his energy. If at any point in the game, PacMan has an energy level of 0, and it hasn't reached the goal yet, then PacMan cannot take any further steps and it will die.

   PacMan's energy level is set to a predefined initial value at the beginning of the game, and then it is updated after every move:

   - step on any position in the grid: the energy level is decremented by 1.
   - step on a position containing a food-dot: the energy level is incremented by the amount of energy that it can gain from eating a food-dot. This is specified at the beginning of the game and it is denoted by *food_ energy_ level.*

   When considering the next move, if PacMan runs low on energy then he would choose to go after a food-dot, even if it means going off the shortest route in order to obtain it.

   Note that the cost of a path is equal to the number of steps in it, as in the PositionSearchProblem.

   The new game rules are implemented in the class *HungerGamesSearchProblem.*

Compared to the original layout of the game, in this case, the maze contains no walls, rather it should be considered a simple field, dotted with a certain amount of food pellets.

1.1.1.4 The State Space

With this new set of rules, it's not enough anymore to memorize PacMan's position only in the state variable. Instead, we need to keep track of

1. PacMan's current position in the maze
2. PacMan's current energy
3. the grid of the remaining food dots

1.1.1.5 Solutions

Ultimately, the logic of the game can be simplified to a shortest-path finding problem, with some additional constraints. The search algorithm we considered for this problem was the A* algorithm, for which we have defined different heuristic functions in order to optimize it.

1.1.1.6 How to get started?

The program expects a couple of arguments in order to configure the layout and the search algorithms/ heuristics used. The format of the command is the following:
python pacman.py
-l <chosen layout>
-z .5
-p SearchAgent
-a fn=astar,
prob=HungerGamesSearchProblem,
pacman_energy_level= <initial energy level of pacman>,
food_energy_level=<energy gained by eating one food dot>,
heuristic=<chosen heuristic>

The following parameters must be set when running the program:

- **prob** = HungerGamesSearchProblem
- **fn** = astar
- **-p** = SearchAgent
- **-l** = which layout to be used for the game; ex.: smallHungerGames.lay, mediumHungerGames.lay, etc.
- **pacman_energy_level** = how much energy should PacMan have at the beginning of the game
- **food_energy_level** = how much should one food dot contribute to the energy of PacMan
- **heuristic** = which heuristic function should be used by the A* search algorithm

Example:
Run the game on layout tinyHungerGames.lay with initial energy level for PacMan of 7 and food energy level of 2, using the hungerGamesCombinedHeuristic:

python pacman.py -l tinyHungerGames -z .5 -p SearchAgent -a fn=astar,
prob=HungerGamesSearchProblem,
pacman_energy_level=7,
food_energy_level=2,
heuristic=hungerGamesCombinedHeuristic

1.1.1.7 New graphical interface for the game

Some changes were made to the interface of the game in order to confirm with the new constraints.

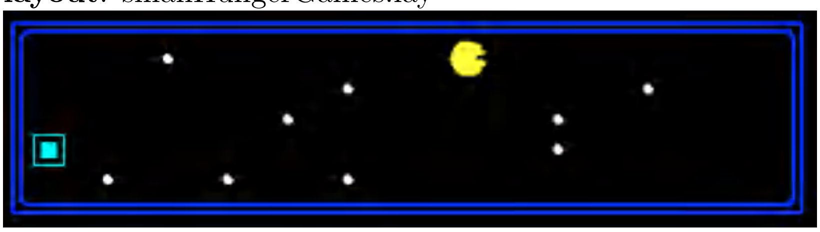
A small icon appears in the maze, representing the exit, this is were PacMan has to reach:

At each moment of the game, we display the current value of PacMan's energy level along with an indicator, which shows approximately how much energy he has left:

- blue: high energy (100%)
- yellow: medium energy (75%)
- orange: low energy (50(%)
- red: critical energy (25(%)

1.1.1.8 Sample execution: PacMan in action

In order to illustrate the rules presented above and how are they applied in the game, we will consider a concrete example.
Using the same intial configuration we run the game with different heuristics and compare the results.
Configuration:
layout: smallHungerGames.lay



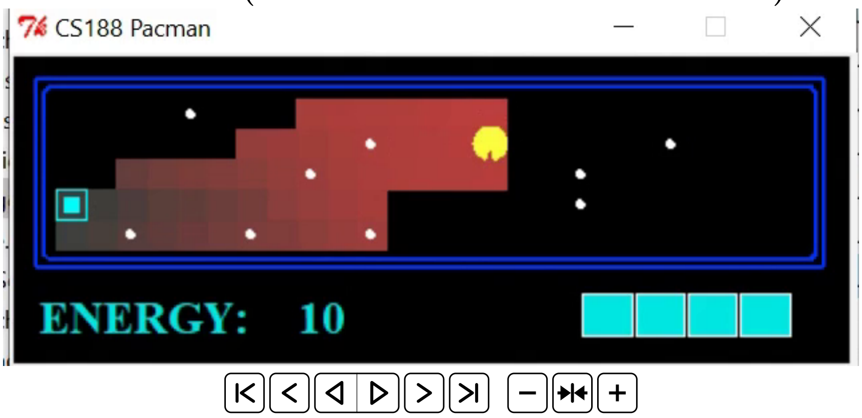
pacman_energy_level: 10

food_energy_level: 3

Compare heuristics:

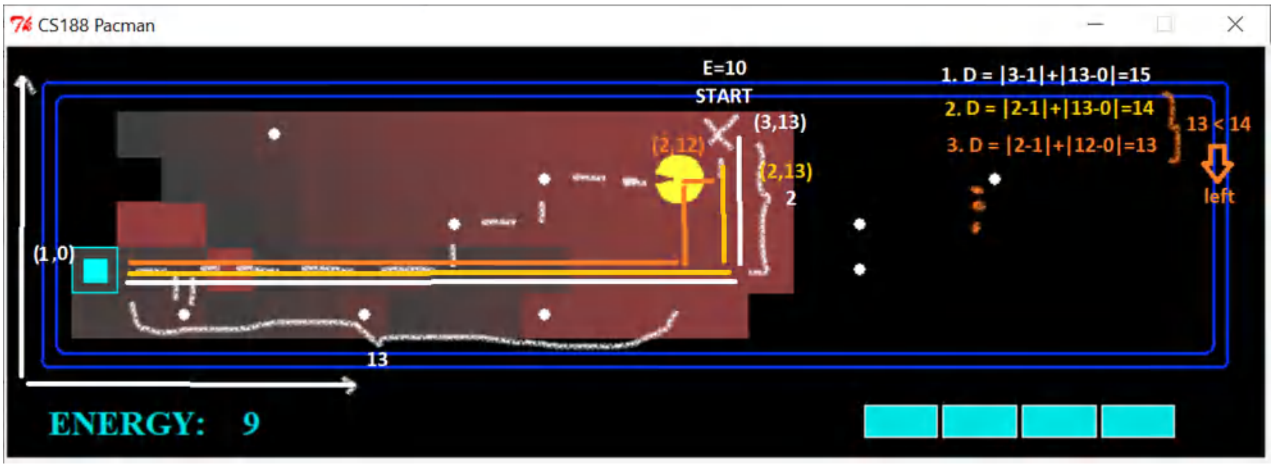
Manhattan distance:

Final heuristic (combination of 3 other heuristics):

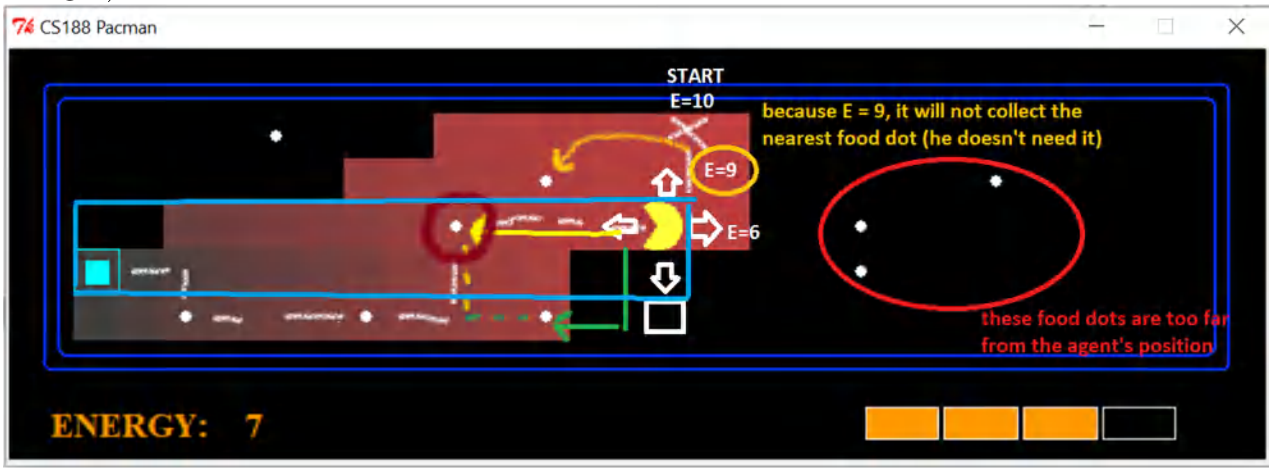


Explain what happened:

- Manhattan distance heuristic:
  At each step the heuristic function evaluates the absolute difference between the corresponding coordinates of the 2 endpoints, and moves in the direction which brings him closer to the goal.
  If he runs low on energy, PacMan needs to do a little detour from the shortest path, in order to collect a food dot, for example he needs to get off the path right before reaching the exit, to collect the food dot below.
  However, as long as the energy level is maintained, there is no need to leave the shortest path in order to gather more food. This explains why he hasn't collected the previous food dots, which were at the same distance (1 cell below).



- Final better heuristic as a combination of other heuristics to guarantee the optimality of the search algorithm:
As seen from the short animation, the second search finishes faster than the first, although the path length is the same. That is because fewer nodes were expanded during the search, thanks to the heuristics which gave a more accurate estimation of the remaining distance until the goal, than in the previous case.
Similarly as before, PacMan doesn't step off the route, going further from the goal (up or right) unless he would need to collect a food dot there.



Consider the current position of PacMan illustrated on the image above. The current energy level of PacMan is 7, but the Manhattan distance to the goal gives 13, which means he has to eat at least 3 food dots($E = E + 3*3 - 2*1$ for the detour), so that he would not starve and successfully reach the exit.
There are 3 food dots in the neighbourhood of Pacman, out of which he chooses to eat the one on the left. Why that one?
If we consider the rectangle formed by the current position of PacMan and the position of the Exit, then the only food dot contained inside the rectangle is the one on the left. The other 2 fall outside this rectangle.
The main idea is to find (if possible and if it won't cause PacMan to get further away from the goal) the closest food dot **inside** this rectangle, that could be gathered along the shortest path. Otherwise extend the search perimeter and try to find all the necessary food dots as close to the shortest path as possible.
The food dot below this rectangle, altough it is at the same distance from its current position, it is on the perimeter of distance 1 from the blue rectangle.
The food dot on the row above PacMan not only is it outside the rectangle but also one

row above, which means PacMan has to step back and move upwards, which means he would get further from the goal.

1.1.2 Heuristics

The algorithm used for finding the shortest possible path from the initial state to the goal state was A* algorithm. Without any heuristic, A* would have been the equivalent of a breadth-first search, but our goal was to develop heuristics which can reduce the number of nodes expanded during the search, such that in the end the searching process takes less time.

We developed several heuristics based on different approaches, and we performed experiments to evaluate and compare their performance. The configurations for the experiments and the results can be seen in the Experiments section.

To easily identify the heuristics, we assigned a letter to each of them. Here's the mapping from the identifier to the function implementing it:

- A = EuclideanHeuristic
- B = ManhattanHeuristic
- C = hungerGamesManhattanAndStepsOutsideRectangleHeuristic
- D = hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic
- E = hungerGamesManhattanShortestPathVerificationHeuristic
- F = hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic
- G = hungerGamesClosestFoodDotReachableHeuristic
- H = hungerGamesCombinedHeuristic

1.1.2.1 Previous heuristics: the Euclidean (A) and the Manhattan (B) distance

When it comes to finding a shortest path in a grid, the two heuristics that naturally come to mind are the Euclidean and the Manhattan distance between the current state's position and the goal state's position.

As these heuristics were discussed at the laboratory and their admissibility and consistency was proved at the lecture, we will not discuss their general characteristics any further this time. However, we still wanted to include them in the experiments, to show how much we managed to improve the search performance compared to these trivial heuristics.

What needs to be noted, is that *none of these two heuristics take into account the energy constraint* imposed by the Hunger Games problem, so they are incapable of predicting the necessity for a by-pass road, with a higher cost than the shortest distance between the current position and the goal position, in case PacMan does not have enough energy.

1.1.2.2 Heuristic C: based on the maximum obtainable energy from the rectangle to the goal

Objective

This heuristic is a first approach to overcome the limitations of the previous heuristics, as it tries to detect some of the cases in which PacMan cannot reach the goal state with a "straight path" of cost ManhattanDistance (current position, goal position).

Train of thought. Proof of the admissibility

First, let's see what a path with cost ManhattanDistance (current position of PacMan, goal position) requires. If we denote Pacman's position by (Px, Py) and the goal position by (Gx, Gy), then PacMan should take exactly (Gx - Px) steps to the right (assuming that a "negative step" to the right is a step to the left) and (Gy - Py) steps upwards.

In the following, we'll use the notations

- **ideal path** to denote a path from the current position of PacMan to the goal position, ignoring the energy constraint. An ideal path always has the cost ManhattanDistance (current position of PacMan, goal position).
- **valid ideal path** to denote an ideal path fulfilling the energy constraint.
- **correct-direction step** to denote any step into a direction, into which the number of steps in an ideal path is positive.
- **incorrect-direction step** to denote any step into a direction, into which the number of steps in an ideal path is negative.

Let's define the **PacMan-Goal-Rectangle** as the rectangular subsection of the maze-grid, enclosed by the current position of PacMan and the goal position. It is trivial to prove that an ideal path needs to be located fully inside the PacMan-Goal Rectangle.

We can say for sure that PacMan *cannot reach* the goal position through an ideal path if

$$\begin{aligned}
\text{the energy required for the ideal path path} \; > \; & \text{PacMan's current energy} \\
& + \text{ the maximum energy that} \\
& \text{PacMan can gain} \\
& \text{through an ideal path} \iff \\
\text{ManhattanDistance(PacMan's position, goal position)} \; > \; & \text{PacMan's current energy} \\
& + \text{ the maximum energy that} \\
& \text{PacMan can gain} \\
& \text{through an ideal path}
\end{aligned} \tag{1.1}$$

Moreover,

Figure 1.1: Example for a PacMan-Goal-Rectangle, marked with orange. In this case, steps to the right and towards the bottom are considered "correct-direction steps", and the others are "incorrect-direction steps"

$$\begin{aligned}\text{the maximum number of food dots that PacMan can eat on an ideal path} <= \\ \text{the number of food dots in the PacMan-Goal-Rectangle} \iff \\ \text{the maximum energy that PacMan can gain through an ideal path} <= \\ \text{the number of food dots in the PacMan-Goal-Rectangle} * \text{food\_energy\_level}\end{aligned} \tag{1.2}$$

Then, based on (1.1) and (1.2), it is guaranteed that PacMan *cannot reach* the goal position through an ideal path if

$$\begin{aligned}\text{ManhattanDistance(PacMan's position, goal position)} > \;& \text{PacMan's current energy} \\ & + (\text{the number of food dots} \\ & \text{in the PacMan-Goal-Rectangle} * \\ & \text{food\_energy\_level})\end{aligned} \tag{1.3}$$

Thus, we can verify if the inequality (1.3) is true, and if it is then we know that PacMan must take at least one incorrect-direction step to gather more energy on a by-pass road.

However, it is obvious, that if PacMan takes an incorrect-direction step, then that step must be "recovered" or "annulled" with another step into the opposite direction, that is also outside the ManhattanDistance steps of an ideal path. Thus, we proved that if no ideal path exists, then the cost of any path is at least ManhattanDistance + 2 $\rightarrow$**if the inequality** (1.3) **is true, then the shortest path to the goal must have the cost** $>=$ **ManhattanDistance** $+$ **2**.

The above observation in itself would form a useful heuristic, as adding 2 to the heuristic value in certain cases would stop certain nodes from being expanded. However, this idea can be further extended, if we try tro find a lower bound for the number of incorrect-direction steps.

To find such a lower bound, the PacMan-Goal-Rectangle will be iteratively extended by 1 cell on each side (see figure 1.2 for a better understanding), until the resulting rectangle will have enough food dots in it, i.e. until

$$\begin{aligned}\text{ManhattanDistance(PacMan's position, goal position)} \;<=\; & \text{PacMan's current energy} \\ & + \text{(the number of food dots in the} \\ & \text{n}^{\text{th}} \text{ extended PacMan-Goal-Rectangle} * \\ & \text{food\_energy\_level)}\end{aligned} \tag{1.4}$$

Figure 1.2: Example for the first 4 extensions of PacMan-Goal-Rectangle. In this particular execution, the red rectangle didn't contain enough food dots to satisfy (1.4) but the green rectangle did. Thus, since the green rectangle is the 2nd extension, we know that PacMan must take at least 2 incorrect-direction steps →ManhattanDIstance + 2 * 2 = ManhattanDistance + 4 is a lower bound for the cost of any valid path to the goal.

Final Conclusion

Then, it is guaranteed that **if the first extended PacmanGoalRectangle that satisfies** (1.4) **is the n$^{\text{th}}$**, then PacMan must take at least n incorrect-direction steps, because PacMan needs to reach at least one position contained in the n$^{\text{th}}$ but not contained in the (n-1)$^{\text{th}}$ extended rectangle. As a consequence, **ManhattanDistance + 2*n is a lower bound for the cost of any path to the goal.**

Main steps for computing the heuristic

1. n = 0

2. while the number of food dots in nth extended rectangle * food_energy_level + PacMan's current energy < ideal path length:
   (a) n++
   (b) extend again the rectangle
3. return ManhattanDistance (PacMan's current position, goal state) + 2 * n

Advantages

- takes into account the energy constraint

Disadvantages

- the verification for an existing valid ideal path is very weak, since it assumes that if PacMan goes on an ideal path, then it can eat all food dots in the PacMan-Goal-Rectangle, but in reality, many food dots may be off-road

1.1.2.3 Heuristic D: based on the maximum obtainable energy on any ideal path

Objective

As stated above, heuristic C's verification for an existing valid ideal path is too optimistic. We should try to find a stricter method, that is less optimistic than heuristic A, but still not pessimistic (i.e. it never says that no valid ideal (or more generally, a (ManhattanDistance + 2n) * cost path exists, if in fact it does exist).

Train of thought. Proof of admissibility

To verify whether an ideal path exists, let's try to compute the maximum number of food dots that PacMan may eat through any ideal path, and check whether that is enough.

Assuming that in an ideal path, PacMan may only take steps to the right and upwards (without loss of generality), let's extract the PacMan-Goal-Rectangle part of the food grid in a matrix, which I'll denote with M. PacMan's current position inside M is then (0, 0), the size of M is (dx+1, dy+1) and the goal's position in M is (dx, dy).

Then, through an ideal path, PacMan may reach position (x, y) inside M only from positions (x-1, y) or (x, y-1) (or a subset of them, in case any of these two positions fall outside M. These edge cases will not be treated in this description, but you may check out the code for more information).

Let's denote the maximum number of food dots that can be eaten by PacMan through a path from (0, 0) to (x, y) with cost (x - 1) + (y - 1) (i.e. an ideal path), by max_food_dots_until[x][y].

Thus, we can compute all values of the max_food_dots_until matrix using the recursive formula

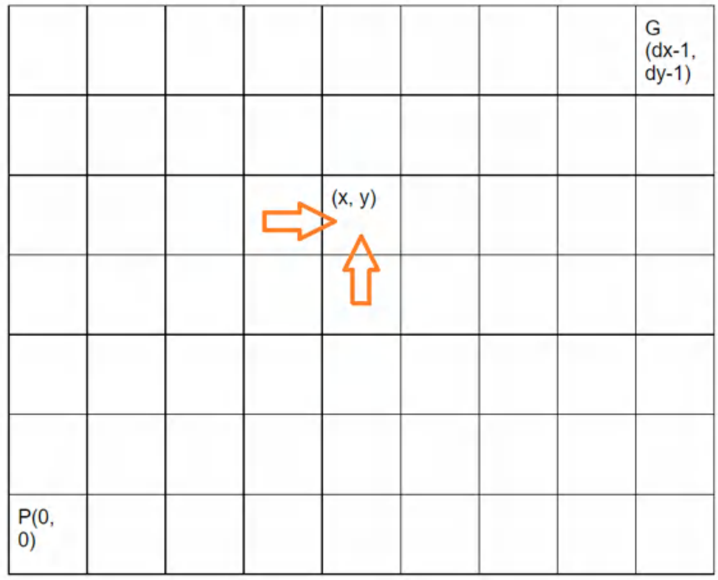


Figure 1.3: Matrix M. P denotes Pacman's position, G the goal's position. The arrows show the steps from whichh position (x, y) may be reached as part of an ideal path.

$$max\_food\_dots\_until[x][y] = \begin{cases} 1+ & max(max\_food\_dots\_until[x-1][y], \\ & max\_food\_dots\_until[x][y-1]), \\ & \text{if there is a food dot on position (x, y).} \\ 0+ & max(max\_food\_dots\_until[x-1][y], \\ & max\_food\_dots\_until[x][y-1]), \text{otherwise.} \end{cases} \tag{1.5}$$

(Note: in the formula, if max_food_dots_until[x-1][y] or max_food_dots_until[x][y-1] falls outside the boundaries of matrix M, then we can consider their value to be 0.)

Finally, max_food_dots_until[dx][dy] will give us the maximum number of food dots that PacMan can eat on an idea path to the goal.

Thus, using the inequality (1.1) from heuristic A, it is guaranteed that PacMan *cannot reach the goal on an ideal path if*

$$\text{ManhattanDistance(PacMan's position, goal position)} > \text{PacMan's current energy} + (\text{max\_food\_dots\_until[dx][dy]} * \text{food\_energy\_level}) \tag{1.6}$$

Final Conclusion

It is guaranteed that **if** (1.6) **is true, then PacMan must take at least 1 incorrect-direction step**, so ManhattanDistance (Pacman's current position, goal position) + 2 is a lower bound for the cost of any path to the goal.

Main steps for computing the heuristic

1. compute the max_food_dots_until matrix
2. if (1.6) is true
   (a) return ManhattanDistance (PacMan's current position, goal state) + 2
3. if (1.6) else
   (a) return ManhattanDistance (PacMan's current position, goal state)

Advantages

- less optimistic for verifying the existence of a valid ideal path, than heuristic A

Disadvantages

- if no valid ideal path exists, cannot give an approximation for how many incorrect-direction steps need to be taken

1.1.2.4 Heuristic E: based on the maximum possible energy level at any step of an ideal path

Objective

All previous heuristics were static, in the sense that they were trying to verify whether PacMan can eat enough food dots, such that together with its existing energy, it can cover the costs of the entire path from the current position to the goal. However, none of these heuristics considered that according to the HungerGamesSearchProblem, PacMan doesn't only need to have enough energy "overall", but needs to have enough energy all the time for the next step. For example, a path which contains lots of food dots in its final section, but PacMan has no energy in the middle is not feasible.

The goal of this heuristic is to find a dynamic verification method for the existence of a valid ideal path, which excludes paths with high total energy but no energy at any of their steps.

Train of thought. Proof of admissibility

Based on the ideas from heuristic D, in fact we only need slight modifications to verify the existence of a valid ideal path considering the constraint that the energy level must be positive at each steps.

We can consider the same matrix M as in the previous case, but instead of the max_food_dots_until[x][y], matrix, let's build the max_energy_level_at matrix of M's size, similarly to how max_food_dots_until was built, in which max_energy_level_at[x][y] gives us the maximum energy level that PacMan could have when it leaves position (x, y) if it went on an ideal path (i.e. of cost (x - 1) + (y - 1) from its current position (0, 0) to position (x, y).

Note: you may wonder why the maximum energy of PacMan when it *leaves* position (x, y) is computed instead of the one when it *arrives* to position (x, y). The explanation is that this approach makes computations simpler, as the potential extra energy gained from eating a food dot on position (x, y) can be added directly to max_food_dots_until[x][y].

Similarly to the approach in heuristic D, we can compute the values of this matrix with a recursive formula.

Let's introduce the notations:

$$max\_parent\_energy\_level[x][y] = max(max\_energy\_level\_at[x-1][y], \\ max\_energy\_level\_at[x][y-1]) \tag{1.7}$$

Semantically, max_parent_energy_level[x][y] gives the maximum energy level that PacMan may have 1 step before reaching position (x, y). Note that to take that 1 step, PacMan's energy level will be decrease by 1.

$$food[x][y] = \begin{cases} 1, \text{ if there is a food dot on position (x, y)} \\ 0, otherwise \end{cases} \tag{1.8}$$

Then, the recursive formula is:

$$max\_energy\_level\_at[x][y] = \begin{cases} food\_energy\_level - 1 + & max\_parent\_energy\_level[x][y], \\ & \text{if food[x][y]} == 1 \text{ and} \\ & \text{max\_parent\_energy\_level[x][y]} > 0 \\ -1+ & max\_parent\_energy\_level[x][y], \\ & \text{if food[x][y]} == 0 \text{ and} \\ & \text{max\_parent\_energy\_level[x][y]} > 0 \\ -1, & \text{if max\_parent\_energy\_level[x][y]} <= 0 \end{cases} \tag{1.9}$$

It's important to understand the role of the 3rd case in the above formula. Logically, that case handles the situation when PacMan simply cannot gather enough energy on an ideal path to reach position (x, y): if PacMan cannot have a positive energy level when "leaving" neither the left nor the lower neighbor of position (x, y), then PacMan simply cannot leave neither the left nor the lower neighbor, so position (x, y) is unreachable according to the rules of the HungerGamesSearchProblem.

This is the key idea to this heuristic: max_parent_energy_level[dx][dy] tells us whether PacMan can reach the goal position from its current position through an ideal path of cost dx + dy, i.e. only taking steps to the right and upwards.

Final Conclusion

$$max\_energy\_level\_at[dx][dy] >= 0 \iff \exists \text{ a valid path of cost ManhattanDistace from PacMan's current position to the goal position, considering all the constraints of the HungerGamesSearchProblem} \quad (1.10)$$

Note that with this heuristic we didn't only give a less optimistic verification approach for the existence of a valid ideal path, but we found an equivalent statement that can be easily computed.

Similarly to the previous heuristics, if no valid ideal path exists, then PacMan must take at least one incorrect-direction step and its annulment step additionally to the dx-1 steps to the right and dy-1 steps to the left, so the ManhattanDistance + 2 is a lower bound for the cost of any valid path from the current position of PacMan to the goal position.

Main steps for computing the heuristic

1. compute the max_energy_level_at matrix
2. if max_energy_level_at >= 0
   (a) return ManhattanDistance (PacMan's current position, goal state)
3. if (1.6) else
   (a) return ManhattanDistance (PacMan's current position, goal state) + 2

Advantages

- takes ihe energy constraint into account at every step
- gives an equivalent condition for the existence of a valid ideal path

Disadvantages

- if no valid ideal path exists, cannot give an approximation for how many incorrect-direction steps need to be taken

1.1.2.5 Heuristic F: based on verifying the existence of a path with a most 1 incorrect-direction step

Objective

In heuristic E we find an easy-to-compute equivalent condition to verifying whether a valid ideal path exists. However, if we could demonstrate that in a certain case, such a path does not exist, we could only guarantee, that the minimum cost path to the goal has the cost >=

ManhattanDistance + 2, whereas in fact the minimum cost may be much higher. Heuristic F should provide a closer approximation to the number of incorrect-direction steps.

Train of thought. Proof of admissibility

Additionally to verifying whether a valid ideal path exists, let's verify whether a valid path of cost ManhattanDistance + 2 exists. Such a path would contain only one single incorrect-direction step.

Note that the above dynamic programming approach cannot be directly applied here, because there we exploited the fact that steps were taken only into two possible directions, so the values in any of the helper matrices could not be mutually interdependent, in the sense that either m[x][y]'s computation was dependent on the value of m[a][b] or vice versa, bot not both. This is what made the recursive formulas possible. Allowing steps into all directions would make the previous O(n*m) algorithm intractable.

What we can do instead is to compute 2 helper matrices. The first one is max_energy_level_at, as in heuristic E. The second one is min_energy_level_at, whose value at (x, y), min_energy_level_at[x][y gives the minimum amount of energy that PacMan must have when arriving to position (x, y) such that a valid path from (x, y) to the goal of cost ManhattanDistance((x, y), goal position) exists.

It's important to notice that if we allow paths of length ManhattanDistance + 2, then we need to extend the PacMan-Goal-Rectangle by 1 on each side, similarly to how we did in heuristic C. The min_energy_level_at matrix will thus have size (dx + 2) * (dy + 2), PacMan will be located on position (1, 1) and the goal position will be (dx + 1, dy + 1).

The reason for which the min_energy_level_at matrix is computed is that it helps us treat paths with 1 single incorrect-direction step: there is such a path from PacMan's current position (1, 1) to the goal (dx + 1, dy + 1), with 1 incorrect-direction step taken at position (ix, iy) to the neighboring position (ix2, iy2), if and only if the the maximum energy that PacMan can have when leaving (ix, iy) after arriving there on a minimum cost path from (1, 1) (= max_energy_level_at[ix][iy]), is strictly greater than the minimum energy that PacMan must have when arriving to (ix2, iy2) such that a minimum-cost path from here to the goal (dx + 1, dy + 1) exists (= min_energy_level_at[ix2][iy2]). Mathematically, there is a path from PacMan's current position (1, 1) to the goal (dx + 1, dy + 1) of cost dx + dy + 2 if and only if

$$\begin{aligned}
&\exists \text{ a valid path of cost at most ManhattanDistace + 2} \\
&\text{from PacMan's current position to the goal position,} \\
&\text{considering all the constraints of the HungerGamesSearchProblem} \iff \\
&\exists \text{(ix, iy) and its neighboring position (ix2, iy2), such that} \\
&\max_energy_level_at[ix][iy] - 1 >= min_energy_level_at[ix2][iy2]
\end{aligned} \tag{1.11}$$

For computing the min_energy_level_at matrix, we may use a similar dynamic programming approach as before, but we need to start the computations at the other side of the matrix, based on the base value min_energy_level_at[dx][dy] = 0, as PacMan can have energy 0 at the goal to have a valid path to the goal, since no further steps are needed.

Since the exact formulas need to treat lots of edge cases, we'll not present them here, but they can be found in the code. Instead, intuitively one may see that the formula is similar to

$$\begin{aligned} min\_child\_energy\_level[x][y] = min(&min\_energy\_level\_at[x+1][y], \\ &min\_energy\_level\_at[x][y+1]) \end{aligned} \tag{1.12}$$

$$food[x][y] = \begin{cases} 1, & \text{if there is a food dot on position (x, y)} \\ 0, otherwise \end{cases} \tag{1.13}$$

Then, the recursive formula is:

$$min\_energy\_level\_at[x][y] = \begin{cases} max(-food\_energy\_level + 1+ & min\_child\_energy\_level[x][y], 0) \\ & \text{if food[x][y] == 1 and} \\ +1+ & max\_parent\_energy\_level[x][y], \\ & \text{if food[x][y] == 0 and} \end{cases} \tag{1.14}$$

Final Conclusion

We can verify the existance of a valid path of cost ManhattanDistance as in heuristic E. If it doesn't exist, we can verify the existence of a valid path of cost ManhattanDistance + 2 with the method described above. If it still doesn't, then ManhattanDistance + 4 is a lower bound for the cost of any path from PacMan's current position to the goal.

Main steps for computing the heuristic

1. compute the max_energy_level_at matrix.
2. if valid ideal path exists (verified as in heuristic E)
   (a) Return ManhattanDistance (PacMan's current position, goal position)
3. compute the min_energy_level_at matrix
4. for all (ix, iy) positions inside the PacMan-Goal-Rectangle
   (a) if there is a neighbor (ix2, iy2) of (ix, iy), such that max_energy_level_at[ix][iy] - 1 >=
   min_energy_level_at[ix2][iy2]
      i. Return ManhattanDistance (PacMan's current position, goal position) + 2
5. return ManhattanDistance (PacMan's current position, goal state) + 4

Advantages

- takes the energy constraint into account at every step
- gives an equivalent condition for the existence of a valid ideal path and for the existence of a valid path of cost ManhattanDistance + 2

Disadvantages

- if no valid path of cost at most ManhattanDistance + 2 exists, cannot give an approximation for how many incorrect-direction steps need to be taken

1.1.2.6 Heuristic G: based on the distance to the closest dot

In contrast with many search problems analyzed at the laboratory, a particularity of this one is that there are states from which not only that there is no *fast* solution, but there is *no solution at all*.

Heuristic G simply assigns +infinity to all states in which PacMan doesn't have enough energy neither to reach the goal directly nor to reach the closest food dot, and 0 to the other states.

1.1.2.7 Heuristic H: the best of all - a combination of the previous heuristics

When we compare the heuristics, we may observe that:

- heuristic G treats the states with no solution
- heuristic C offers an over-optimistic test for verifying the existence of a (ManhattanDistance (current position, goal position) + 2*n)-cost path, but can return closer-to-real values in some cases. Useful only when the number of food dots inside the PacMan-Goal-Rectangle doesn't cover PacMan's energy requirements for an ideal path.
- heuristic F, which is an improved version of D and E, offers an equivalent condition for the existence of a (ManhattanDistance (current position, goal position))-cost path and for the existence of a (ManhattanDistance (current position, goal position) + 2)-cost path, but cannot treat states for which the minimum cost is much higher than that. Useful even when the number of food dots inside the PacMan-Goal-Rectangle does cover PacMan's energy requirements for an ideal path.

As a consequence, we may combine the earlier heuristics into a new, also admissible heuristic.

Main steps for computing the heuristic

1. heuristic G: if PacMan doesn't have enough energy neither to reach the goal directly nor to reach the closest food dot
   (a) Return +infinity
2. heuristic C - initial verification: if the number of food dots inside the rectangle can cover PacMan's energy requirements to the goal

(a) return the value of heuristic C

3. else: return the value of heuristic F

Note that this combination is not optimal in terms of the number of expanded search nodes, in some cases it may be worth to run both heuristic C and F and return their maximum value. However, in practice these cases are hard to detect and running both heuristics each time would double the actual execution time, even if the number of expanded search nodes would be lower.

1.1.3 Experiments

1. **Heuristic A: Euclidean distance heuristic**

In this case, the search algorithm expands almost all nodes inside the rectangle determined by the two endpoints, which is very inefficient in case of an a larger search space. Also, to be noted the direction of traversal, when possible, the agent tries to choose the path which brings him closer to the exit (to the left or down).

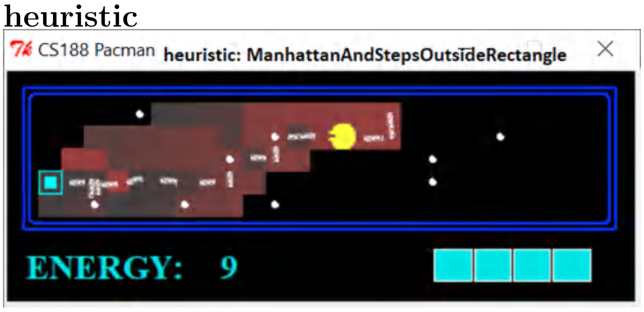
The heuristic didn't consider however the energy level of the agent when estimating the remaining distance, and so he still needed to do a small detour right before reaching the goal, as it remained without energy, thus adding to the cost.

2. **Heuristic B: Manhattan distance heuristic**

The Manhattan distance should be more suited for an agent which can only move in 4 directions, as it gives the shortest distance between any 2 points in the grid.

It can be observed, that PacMan gathered the same food dots as in case of the Euclidean heuristic, but the traversed path is slightly different, first it moves downwards instead of to the left.
The number of expanded nodes is still almost as much as in the previous case, width * height of the rectangle determined by the 2 endpoints.

3. **Heuristic C: Manhattan distance and the number of steps outside rectangle heuristic**

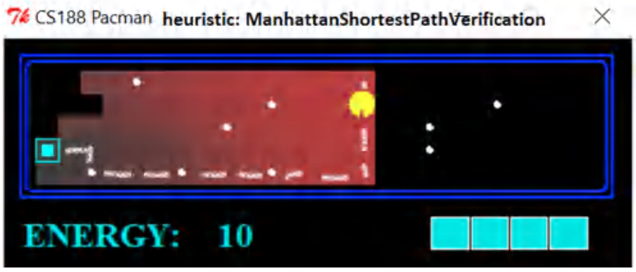


In this case, the heuristic considers how many steps would PacMan need to make outside the rectangle determined by its current position and the Exit's position, which obviously would add to the total traversed distance.
Therefore it will select a path that guarantees the min number of necessary food and it keeps PacMan inside the rectangle as much as possible.

4. **Heuristic D: Heuristic considering the Manhattan distance and the max number of food dots on the shortest path**

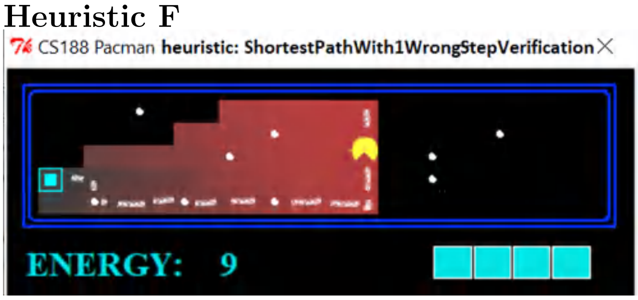
If there not enough food dots along the shortest path, then PacMan needs to leave the **goal rectangle**, and search outside it for food. That means he makes 1 move in the direction opposite to the goal, but as in the end he needs to arrive to the goal, it means when he comes back to the correct path, he needs to step back at least 1 position.
This idea is illustrated on the screen capture, when PacMan moves 1 position downwards, right before reaching the goal, to get the food dot, then he comes right back, moving up 1 position, and arrives to the goal.
This means the length of the shortest path to the goal has to be incremented by 2, when PacMan has to make such moves.
This heuristic resulted in almost the same solution as above, however, during the search, more nodes were expanded than in case of the previous heuristic.

5. **Heuristic E**



This heuristic considers the shortest path, or "min-cost" path to reach the goal, on which PacMan would surely not starve (it would be able to reach the exit). In contrast to the previous case, the path which was found leads PacMan outside the perimeter of the initial rectangle, where the most food dots are.
Interesting to note that PacMan does not have to make large detours to obtain all the food dots it needs. However, the number of expanded nodes is much larger than for the previous case.

6. **Heuristic F**

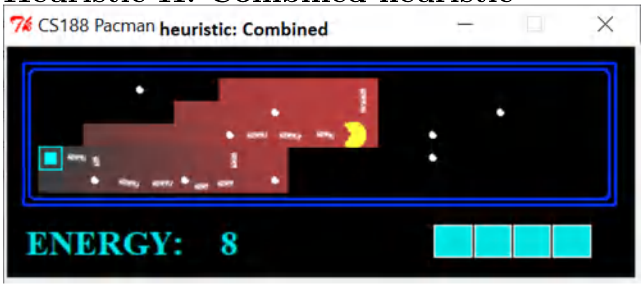


Similar to the previous case, but with fewer nodes expanded, which is due to the fact that it also considers whether PacMan makes more than 2 additional steps when he steps off of the shortest path and the rectangle to get to the food dots.

7. **Heuristic G: Closest food dot reachable heuristic**

The following heuristic verified whether the goal or the closest food dots is reachable with PacMan's current energy level. This resulted in expanding more nodes in most cases, as just very few states could be categorized as "impossible to solve", compared to the total number of the states.

8. **Heuristic H: Combined heuristic**



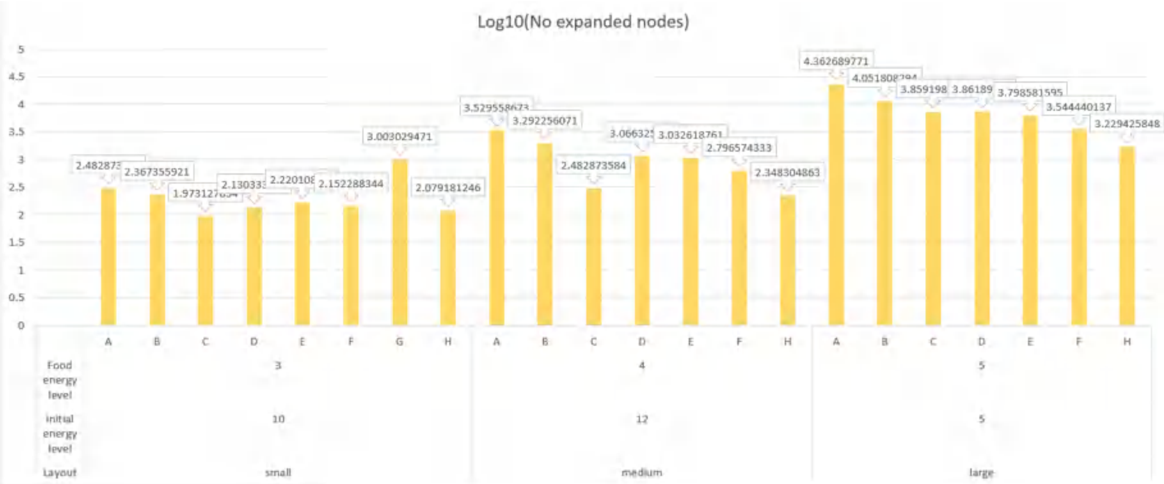
The most efficient heuristic is this one, which combines the previously defined heuristics to find the optimal solution. It expands the least amount of nodes and it is also the quickest.

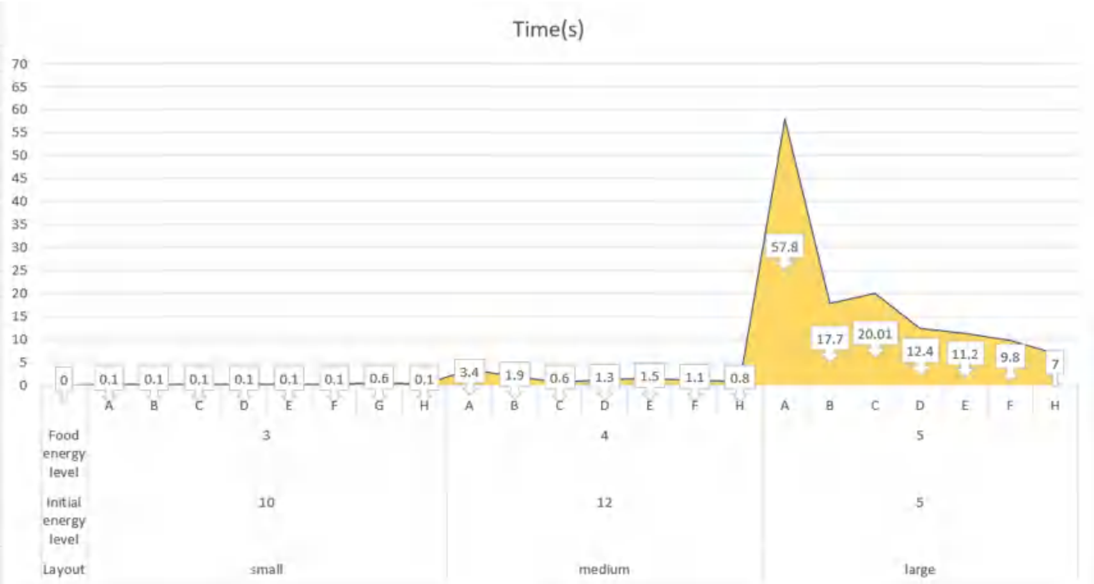
Notations:

- A = EuclideanHeuristic
- B = ManhattanHeuristic
- C = hungerGamesManhattanAndStepsOutsideRectangleHeuristic
- D = hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic
- E = hungerGamesManhattanShortestPathVerificationHeuristic
- F = hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic
- G = hungerGamesClosestFoodDotReachableHeuristic
- H = hungerGamesCombinedHeuristic

Configuration			Heuristic	No expanded nodes	Time
Layout	Initial energy	Food energy			
small	10	3	A	304	0.1
			B	233	0.1
			C	94	0.1
			D	135	0.1
			E	166	0.1
			F	142	0.1
			G	1007	0.6
			H	120	0.1
medium	12	4	A	3385	3.4
			B	1960	1.9
			C	304	0.6
			D	1165	1.3
			E	1078	1.5
			F	626	1.1
			H	223	0.8
large	5	5	A	23051	57.8
			B	11267	17.7
			C	7231	20.01
			D	7276	12.4
			E	6289	11.2
			F	3503	9.8
			H	1696	7.0
sparse	12	4	A	15909	46.3
			B	10946	21.8
			C	1740	4.4
			D	5563	10.3
			E	5306	10.3
			F	2924	8.1
			H	988	4.1
dense	5	1	A	8287	14.1
			B	3518	3.8
			C	802	1.1
			D	802	0.7
			E	452	0.4
			F	135	0.2
			H	73	0.1
diagonal	5	2	A	6429	6.7
			B	1147	1.0
			C	1104	1.7
			D	723	0.9
			E	113	0.2
			F	113	0.2
			H	113	0.3

Plots:
The number of expanded nodes and the execution time increases with the dimensions and the complexity of the search space.



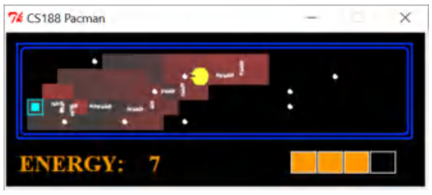


1.1.3.1 Conclusions based on the measurements:

One can immediately see, that the combined heuristic is the most efficient in almost all cases that we analyzed, both in terms of execution time and in terms of the number of expanded nodes. However, to understand how each elementary heuristic contributed to this result, let's analyze the cases separately.

- Small layout:

  For the small maze the best results were obtained when running the algorithm with heuristic C.

- Medium layout:

  The C heuristic produced very good results for the medium layout as well, in which the food dots were placed close to the perimeter of the "safe" rectangle (determined by the Manhattan distance) and PacMan didn't have to wonder off from the shortest path very far to gather the foods, almost as good in terms of the number of expanded nodes as the H heuristic, and it even exceeds it if we consider the execution time.

- Large layout:

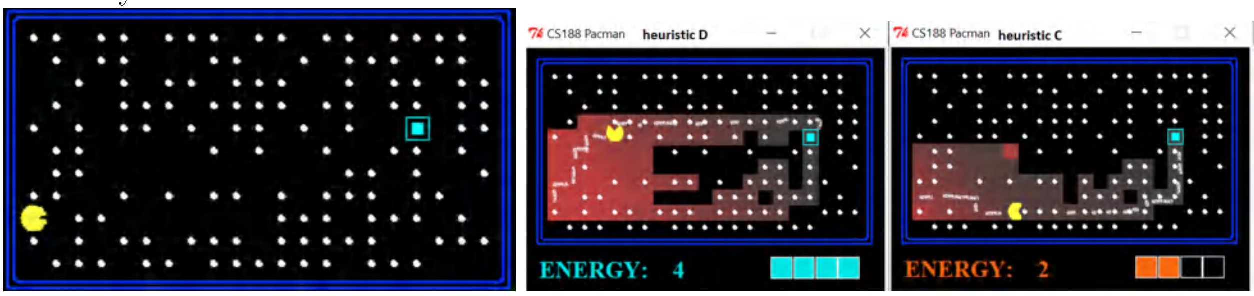
  

  In case of the large layout, the H heuristic(*hungerGamesCombinedHeuristic*) produced the best results. It's interesting to note though, that for a relatively random, large layout, heuristic F produced twice better results than heuristic C.

- Sparse layout:

Again, huristic H is the absolute winner, but this time heuristic C performs bettern than F.

- Dense layout:



In case of the dense layout, where many food dots were scattered around the maze, heuristic H produced outstanding results, and it is clear that heuristic F contributed the most to this succes (compared to heuristic C). Heuristic D (*hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic*) and heuristic C (*hungerGamesManhattanAndStepsOutsideRectangleHeuristic*) were just as good, with heuristic C slightly faster, but both of them explored the most nodes on the perimeter of the "safe" rectangle,a s close to the shortest path as possible.

- Diagonal layout:

Again, heuristic F was the one that made the short searching time possible, as it expanded 10 times less nodes than heuristic C. In case of the diagonal maze, the difference between the performance of the two heuristics: D (*hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic*) and E (*hungerGamesManhattanShortestPathVerificationHeuristic*) was much larger, heuristic D expanded almost 7 times more nodes then E when searching for the shortest path to the exit.

As a conclusion, we may say that heuristic H performs well because heuristic C and F complement each other: they behave well in different cases, but by combining them, we can achieve a fast searching algorithm for most of the cases, based on our experiments.

This is not a surprising conclusion, if we think about the approaches applied in heuristics C and F:

1. heuristic C treats well the sparser mazes, when the number of food dots in the Pacman-Goal-Rectangle is small. In these cases, heuristic C can provide closer approximations for the minimum cost of a path than heuristic F, as it can categorize the cases into multiple categories, not just 3.

2. heuristic F on the other hand can handle well the cases when there are lots of food dots in the PacMan-Goal-Rectangle, but PacMan cannot eat all of them unless it takes a short, curvy road, with lots of non-goal oriented steps. However, heuristic F categorises layouts into only 3 categories, and the maximum approximation that it can give for the minimum-cost of a path is ManhattanDistance + 4, whereas in reality, much more steps may be needed.

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Appendix A

Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. 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.

```
class HungerGamesSearchProblem(search.SearchProblem):
    """
    A search problem defines the state space, start state,
    goal test, successor
    function and cost function.  This search problem can be used to
    find paths
    to a particular point (maze exit point) in the maze,
    with the constraint that PacMan must always have a positive
    energy level.

    Initially, PacMan has a given energy level:
    pacman_energy_level. Then,
    - PacMan loses one energy level in each step
    - PacMan can get food_energy_level extra energy levels with
    every food
    dot that it eats.

    The state space consists of ((x, y), energy_level, food_grid)
    tuples, where
    - (x, y) denotes PacMan's current position
    - energy_level marks the current energy level of PacMan
    - food_grid is a grid showing whether a cell of the maze has
    any food on it.
    """

    IMPOSSIBLE_TO_SOLVE_STATE_HEURISTIC_VALUE = 1000000

    def __init__(self, game_state, pacman_energy_level=10,
                 food_energy_level=3, warn=True, visualize=True):
        """
        Stores the start and goal.

        gameState: A GameState object (pacman.py)
        pacman_energy_level: The initial energy of PacMan
        food_energy_level: The extra energy given by each food dot.
        warn: If set to true, the validity of the initial game
        state is
        initialized.
```

```
        visualize: If set to true, the expanded nodes are marked in
        the maze
        layout, in the graphical window.
        """
        self.walls = game_state.getWalls()
        self.startState = (
        game_state.getPacmanPosition(), pacman_energy_level,
        game_state.getFood())
        self.foodEnergyLevel = food_energy_level
        self.mazeExitPosition = game_state.getMazeExitPosition()
        self.visualize = visualize
        if warn and (self.mazeExitPosition == ()):
            print 'Warning: this does not look like a regular ' \
                  'hunger games ' \
                  'search maze'

        # For display purposes
        self._visited, self._visitedlist, self._expanded = {}, [], \
                                                               0  # DO
        # NOT CHANGE

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = state[0] == self.mazeExitPosition

        # For display purposes only
        if isGoal and self.visualize:
            self._visitedlist.append(state[0])
            import __main__
            if '_display' in dir(__main__):
                if 'drawExpandedCells' in dir(
                        __main__._display):  # @UndefinedVariable
                    __main__._display.drawExpandedCells(
                        self._visitedlist)  # @UndefinedVariable

        return isGoal

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a
        cost of 1.

         As noted in search.py:
             For a given state, this should return a list of triples,
         (successor, action, stepCost), where 'successor' is a
         successor to the current state, 'action' is the action
         required to get there, and 'stepCost' is the incremental
         cost of expanding to that successor
        """

        successors = []
        x, y = state[0]
        food_grid = state[2]
        energy_level = state[1]

        for action in [Directions.NORTH, Directions.SOUTH,
                       Directions.EAST, Directions.WEST]:
            dx, dy = Actions.directionToVector(action)
```

```
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                next_energy_level = energy_level - 1
                next_food_grid = food_grid.copy()
                if next_food_grid[nextx][nexty]:
                    next_energy_level += self.foodEnergyLevel
                    next_food_grid[nextx][nexty] = False
                if next_energy_level > 0:
                    # Else: invalid successor state, because PacMan
                    # would die
                    # because of hunger
                    next_state = ((nextx, nexty), next_energy_level,
                                  next_food_grid)
                    cost = 1
                    successors.append((next_state, action, cost))

        # Bookkeeping for display purposes
        self._expanded += 1  # DO NOT CHANGE
        if state not in self._visited:
            self._visited[state] = True
            self._visitedlist.append(state[0])

        return successors

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions. If
        those actions
        include an illegal move (stepping on a wall, or loosing all
        the
        energy), return 999999.
        """
        if actions == None: return 999999
        x, y = self.getStartState()[0]
        food_grid = self.getStartState()[2]
        cost = 0
        energy_level = self.getStartState()[1]
        for action in actions:
            # Check figure out the next state and see whether its'
            # legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            energy_level = energy_level - 1
            if food_grid[x][y]:
                energy_level += self.foodEnergyLevel
            if energy_level < 0 or self.walls[x][y]: return 999999
            cost += 1
        return cost


def hungerGamesEuclideanHeuristic(state, problem):
    """
    The Euclidean distance heuristic for a HungerGamesSearchProblem

    Heuristic identifier in the documentation: A
    """
    curr_pos = state[0]
    goal = problem.mazeExitPosition
    return ((curr_pos[0] - goal[0]) ** 2 + (
                curr_pos[1] - goal[1]) ** 2) ** 0.5
```

```python


def manhattanDistance(pointA, pointB):
    """Helper function for computing the ManhattanDistance between
    two points
    given via their (x, y) coordinates"""
    return abs(pointA[0] - pointB[0]) + abs(pointA[1] - pointB[1])


def hungerGamesManhattanHeuristic(state, problem):
    """
    The Manhattan distance heuristic for a HungerGamesSearchProblem

    Heuristic identifier in the documentation: B
    """
    curr_pos = state[0]
    goal = problem.mazeExitPosition
    return manhattanDistance(curr_pos, goal)


def isPosInRectangle(corner1, corner2, pos):
    """
    Returns True if (x, y) position is located inside the rectangle
    defined
    by the two corners,
    which are on one of the diagonals.
    """
    rectangle_low_bound = min(corner1[0], corner2[0])
    rectangle_high_bound = max(corner1[0], corner2[0])
    rectangle_left_bound = min(corner1[1], corner2[1])
    rectangle_right_bound = max(corner1[1], corner2[1])
    return rectangle_high_bound >= pos[
        0] >= rectangle_low_bound and rectangle_right_bound >= pos[
               1] >= rectangle_left_bound


def noFoodDotsInRectange(corner1, corner2, food_grid):
    """
    Computes the number of food dots located inside the rectangle
    defined by
    two corners,
    which are on one of the diagonals.

    food_grid: boolean matrix, with food_grid[x][y] True iff there
    is a food
    dot in the location (x, y).
    corner1, corner2: 2 tuples in the format (x coordinate,
    y coordinate),
                      marking the two diagonal corners of the
                      rectangle in
                      concern.
    """
    return len([food_dot for food_dot in food_grid.asList() if
                isPosInRectangle(corner1, corner2, food_dot)])


def buildGoalOrientedIntegerFoodGridRectangle(start_corner_pos,
                                              goal_corner_pos,
                                              food_grid):
    """
```

```
    Builds a matrix with the content of food_grid inside the rectangle
    defined by the 2 corners
    (start_corner_pos and goal_corner_pos), which represent the
    endpoints of
    one of the diagonals of the rectangle.

    The matrix may be mirrored horizontally and/or vertically,
    such that start_corner_pos ends up being mapped to position (0,
    0) of the
    final matrix,
    and goal_corner_pos ends up being mapped to position (n-1,
    m-1) of the
    final matrix,
    where n and m are the number of rows and columns in the rectangle.
    """
    rows_step = 1
    cols_step = 1
    if goal_corner_pos[0] < start_corner_pos[0]:
        rows_step = -1
    if goal_corner_pos[1] < start_corner_pos[1]:
        cols_step = -1
    start_row = start_corner_pos[0]
    start_col = start_corner_pos[1]

    res = [[int(food_grid[start_row + i * rows_step][
                    start_col + j * cols_step]) for j in
            range(abs(goal_corner_pos[1] - start_col) + 1)] for i in
           range(abs(goal_corner_pos[0] - start_row) + 1)]
    return res


def hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic(state,
                                                          problem):
    """
    A heuristic for the HungerGamesSearchProblem, based on the
    following idea:
    - any path from the current state to the goal takes at least
    manhattan
    distance steps
    - if there is a path from the current position to the maze exit
    position,
    such that the manhattan distance <= current energy level +
    energy level
    gained from food dots on the path,
    then it's possible, that a path satisfying all problem
    constraints with
    cost manhattan distance exists
    - if there is no such path, then PacMan must step at least once
    in the
    "wrong direction".
    By wrong direction we mean that for obtaining a path with
    manhattan
    distance cost,
    if the maze exit is to the South from PacMan's position,
    then any step to
    the North is wrong and vice-versa.
    Similarly, if the exit is to the East from PacMan's position,
    then any
    step to the West is wrong, and vice-versa.
    Moreover, if PacMan takes one step to any wrong direction,
```

```
    than that step
    must be "recovered" later,
    i.e. annulled with a backwards step.
    Thus, we can guarantee that if no path with manhattan distance
    cost exists,
    then any path has the cost >= manhattan distance + 2.

    Note that this heuristic takes into account only the energy level
    required for the entire path to the goal,
    but does not verify whether there is enough energy at all steps
    of the path.

    To verify whether a path with the above characteristics exists,
    a dynamic
    programming approach is used,
    based on the formula
    max no. food dots to (x, y) = max(max no. food dots to (x-1,
    y), max no.
    food dots to (x, y-1)) +
                                    + no. food dots on position (x, y)
    Please refer to the documentation for more information.

    Heuristic identifier in the documentation: D
    """
    curr_pos = state[0]
    energy_level = state[1]
    food_grid = state[2]
    goal = problem.mazeExitPosition

    # compute in max_food_dot_grid[x][y] the maximum number of food
    # dots that
    # can be eaten by PacMan along a path from
    # (0, 0) to (x, y), with only steps to the right (y++) or to
    # the left(x++).
    # O(n*m) dynamic programming algorithm, where n and m are the
    # sizes of
    # the grid
    max_food_dot_grid = buildGoalOrientedIntegerFoodGridRectangle(
        curr_pos, goal, food_grid)
    for j in range(1, len(max_food_dot_grid[0])):
        max_food_dot_grid[0][j] += max_food_dot_grid[0][j - 1]

    for i in range(1, len(max_food_dot_grid)):
        max_food_dot_grid[i][0] += max_food_dot_grid[i - 1][0]

    for i in range(1, len(max_food_dot_grid)):
        for j in range(1, len(max_food_dot_grid[i])):
            max_food_dot_grid[i][j] += max(
                max_food_dot_grid[i - 1][j],
                max_food_dot_grid[i][j - 1])

    # Find the maximum number of food dots that can be eaten by PacMan
    # through a minimum-cost path
    # from the starting position (0, 0) to the goal.
    max_food_on_shortest_path = \
    max_food_dot_grid[len(max_food_dot_grid) - 1][
        len(max_food_dot_grid[0]) - 1]
    shortest_path_length = manhattanDistance(curr_pos, goal)

    # If by eating the maximum amount of food dots that can be
```

```
    # found on a
    # minimum cost path PacMan still wouldn't have
    # enough energy to reach the goal, then at least 1 step in the
    # wrong
    # direction + 1 annulment step is needed,
    # additionally to the cost of manhattan distance.
    if energy_level + problem.foodEnergyLevel * \
            max_food_on_shortest_path < shortest_path_length:
        # Not enough food on any minimum cost path
        return shortest_path_length + 2
    else:
        # Possibly enough food on the shortest path
        return shortest_path_length


def buildMaxEnergyLevelGrid(init_energy_level, food_grid,
                            food_energy_level):
    """
    Given
    - the initial energy of PacMan, assumed to be located in
    position (0,
    0) of the food_grid,
    - the energy level given by a food dot (food_energy_level)
    - the food_grid with the maze exit situated in the top-right
    corner of
    the grid,
    and the current position of PacMan being (0, 0).
    Computes the maximum energy level which PacMan may have when
    leaving
    location (x, y) of the grid, assuming that
    PacMan reached this location via a path from (0, 0) with x+y
    steps (i.e.
    a minimum-cost path with steps only into
    the correct directions).
    Returns a matrix with the values for all (x, y) locations (of
    the same
    size as food_grid).

    If a location (x0, y0) is not reachable according to the rules of
    HungerGames, with only steps into the correct
    directions, then -1 is placed on the given location.

    Note: the "leaving energy" is computed, not the "reaching energy",
    meaning that the energy given by the potential
    doos dot on psotion (x, y) is added to the value computed for
    location (
    x, y).

    For the computations, a dynamic programming algorithm is used,
    based on
    the formula
    max energy on (x, y) = max(max energy on (x-1, y), max energy
    on (x, y-1)) +
                            + the energy given by the food dots on
                            location (
                            x, y)
                            - 1
    Where -1 is due to the cost of stepping from (x-1, y) or from (
    x, y-1) to
    (x, y).
```

```
    """
    from copy import deepcopy
    max_energy_level_grid = deepcopy(food_grid)
    max_energy_level_grid[0][0] = init_energy_level

    for j in range(1, len(max_energy_level_grid[0])):
        if max_energy_level_grid[0][j - 1] > 0:
            max_energy_level_grid[0][j] = max_energy_level_grid[0][
                                              j - 1] + \
                                          max_energy_level_grid[0][
                                              j] * \
                                          food_energy_level - 1
        else:
            max_energy_level_grid[0][j] = -1

    for i in range(1, len(max_energy_level_grid)):
        if max_energy_level_grid[i - 1][0] > 0:
            max_energy_level_grid[i][0] = \
            max_energy_level_grid[i - 1][0] + \
            max_energy_level_grid[i][0] * food_energy_level - 1
        else:
            max_energy_level_grid[i][0] = -1

    for i in range(1, len(max_energy_level_grid)):
        for j in range(1, len(max_energy_level_grid[i])):
            max_parent_energy_level = max(
                max_energy_level_grid[i - 1][j],
                max_energy_level_grid[i][j - 1])
            if max_parent_energy_level > 0:
                max_energy_level_grid[i][
                    j] = max_parent_energy_level + \
                         max_energy_level_grid[i][
                             j] * food_energy_level - 1
            else:
                max_energy_level_grid[i][j] = -1

    return max_energy_level_grid


def hungerGamesManhattanShortestPathVerificationHeuristic(state,
                                                          problem):
    """
    A heuristic for the HungerGamesSearchProblem, which adds an
    improvement to
    hungerGamesManhattan2MaxFoodOnShortestPathHeuristic, in that in
    doesn't
    only verify whether there exists any
    minimum-cost (=manhattan distance) path from the current state
    to the
    goal such that
    manhattan distance <= current energy level + energy level
    gained from
    food dots on the path,
    but it considers only the minimum-cost paths along which PacMan
    does not
    reach an energy level of 0 at any point.

    To verify whether a path with the above characteristics exists,
    a dynamic
    programming approach is used,
```

```
    as explained in buildMaxEnergyLevelGrid.
    Please refer to the documentation for more information.

    Heuristic identifier in the documentation: E
    """
    curr_pos = state[0]
    energy_level = state[1]
    food_grid = state[2]
    goal = problem.mazeExitPosition

    goal_oriented_food_grid = \
        buildGoalOrientedIntegerFoodGridRectangle(
        curr_pos, goal, food_grid)

    max_energy_level_grid = buildMaxEnergyLevelGrid(energy_level,
                                                    goal_oriented_food_grid,
                                                    problem.foodEnergyLevel)

    shortest_path_length = manhattanDistance(curr_pos, goal)

    if max_energy_level_grid[len(max_energy_level_grid) - 1][
        len(max_energy_level_grid[0]) - 1] < 0:
        # Not enough food
        return shortest_path_length + 2
    else:
        # Possibly enough food on the shortest path
        return shortest_path_length


def buildMinEnergyLevelGrid(food_grid, food_energy_level):
    """
    Given
    - the energy level given by a food dot (food_energy_level)
    - the food_grid with the maze exit situated in the (n-2,
    m-2) location,
    where n and m give the height and the width of the grid
    Computes the minimum energy level which PacMan must have when
    reaching
    location (x, y) of the grid,
    such that a valid, minimum cost (=manhattan distance((x, y),
    goal) path
    to the goal (n-2, m-2),
    according to the rules of HungerGames, exists.

    Returns a matrix with the values for all (x, y) locations (of
    the same
    size as food_grid).

    If the energy level when reaching (x, y) does not matter,
    because the
    energy gained from the food dot on (x, y)
    is enough anyway, then a 0 value is assigned to location (x, y).

    For the computations, a dynamic programming algorithm is used,
    based on
    the formula
    min energy when (x, y) = min(min energy when reaching (x+1, y),
    min energy when reaching (x, y-1)) +
                               - the energy given by the food dots on
                               location (x, y)
```

```
                            + 1
    Where +1 is due to the cost of stepping from (x, y) to (x,
    y+1) or to (
    x+1, y).
    """
    # assumes minimum 3 rows and 3 columns in the grid
    no_rows = len(food_grid)
    no_cols = len(food_grid[0])
    from copy import deepcopy
    min_energy_level_grid = deepcopy(food_grid)
    min_energy_level_grid[no_rows - 1][no_cols - 1] = 0
    min_energy_level_grid[no_rows - 2][no_cols - 2] = 0

    for j in range(no_cols - 2, -1, -1):
        min_energy_level_grid[no_rows - 1][j] = max(0,
                                                    min_energy_level_grid[
                                                        no_rows - 1][
                                                        j + 1] -
                                                    food_energy_level *
                                                    food_grid[
                                                        no_rows - 1][
                                                        j] + 1)

    for j in range(no_cols - 3, -1, -1):
        min_energy_level_grid[no_rows - 2][j] = max(0,
                                                    min_energy_level_grid[
                                                        no_rows - 2][
                                                        j + 1] -
                                                    food_energy_level *
                                                    food_grid[
                                                        no_rows - 2][
                                                        j] + 1)

    for i in range(no_rows - 2, -1, -1):
        min_energy_level_grid[i][no_cols - 1] = max(0,
                                                    min_energy_level_grid[
                                                        i + 1][
                                                        no_cols -
                                                        1] -
                                                    food_energy_level *
                                                    food_grid[i][
                                                        no_cols -
                                                        1] + 1)

    for i in range(no_rows - 3, -1, -1):
        min_energy_level_grid[i][no_cols - 2] = max(0,
                                                    min_energy_level_grid[
                                                        i + 1][
                                                        no_cols -
                                                        2] -
                                                    food_energy_level *
                                                    food_grid[i][
                                                        no_cols -
                                                        2] + 1)

    for i in range(no_rows - 3, -1, -1):
        for j in range(no_cols - 3, -1, -1):
            min_child_energy_level = min(
                min_energy_level_grid[i][j + 1],
                min_energy_level_grid[i + 1][j])
```

```
            min_energy_level_grid[i][j] = max(0,
                                                      min_child_energy_level -
    food_energy_level *
                                                      food_grid[i][j] + 1)

    return min_energy_level_grid


def extendMatrixWith0sOnAllSides(m):
    """
    Returns a matrix with 2 additional rows (the first and the last
    one) and
    2 additional columns (the first and the
    last one) compared to m, such that the middle values are taken
    from m and
    the additional rows and columns are filled
    with 0s.
    """
    m.insert(0, [0 for i in range(len(m[0]))])
    m.append([0 for i in range(len(m[0]))])
    for row in m:
        row.insert(0, 0)
        row.append(0)
    return m


def extendRectangeCornersInEachDirection(corner1, corner2):
    """
    Given twe two tuples corner1 and corner2, both in the format (
    x, y),
    representing 2 diagonal corners of a rectangle,
    computes the coordinates of a rectangle extended with 1 row and
    1 column
    on each side, and returns the coordinates
    of this extended rectangle.
    """
    x1, y1 = corner1
    x2, y2 = corner2

    if x2 < x1:
        x1 = x1 + 1
        x2 = x2 - 1
    else:
        x1 = x1 - 1
        x2 = x2 + 1
    if y2 < y1:
        y1 = y1 + 1
        y2 = y2 - 1
    else:
        y1 = y1 - 1
        y2 = y2 + 1

    return (x1, y1), (x2, y2)


def hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic(
        state, problem):
    """
    A heuristic for the HungerGamesSearchProblem, which adds an
    improvement to
```

```
    hungerGamesManhattan2ShortestPathVerificationHeuristic, in that it
    doesn't only verify whether a minimum-cost path
    to the goal exists, and returns minimum cost + 2 for all other
    cases,
    but also verifies whether a path with cost
    manhattan distance + 2 exists, and returns manhattan distance +
    4 for all
    other cases.

    To implement this verification, additionally to the methods in
    hungerGamesManhattan2ShortestPathVerificationHeuristic, it was
    verified
    whether 1 single step into a wrong direction
    + its annulment step is enough to reach the goal while
    fulfilling the
    constrains of HungerGamesSearchProblem.
    For this
    - the maximum possible energy level of PacMan was computed when
    leaving
    the position (x, y), assuming that PacMan
    reaches (x, y) through a path with steps only into a correct
    direction,
    from its current position.
    See buildMaxEnergyLevelGrid.
    - the minimum required energy level of PacMan when reaching
    position (a,
    b) was computed, such that the goal is
    reachable from (a, b) through a minimum-cost path (only steps
    in the
    correct direction)
    See buildMinEnergyLevelGrid.
    Thus,
    1. if the maximum possible energy level at the goal is >= 0,
    then and
    only then a path with only correct steps exists
    --> cost = manhattan distance
    2. if there is (x, y) and (a, b) such that they are neighboring
    positions, (a, b) is at one wrong step from
    (x, y), and the maximum energy at (x, y) - 1 >= the minimum
    required
    energy at (a, b), then (and only then)
    it is sure, that a path with just one wrong step and its
    annulment exists
    --> cost = manhattan distance + 2
    3. otherwise. cost >= manhattan distance + 4

    Heuristic identifier in the documentation: F
    """
    curr_pos = state[0]
    energy_level = state[1]
    food_grid = state[2]
    goal_pos = problem.mazeExitPosition

    goal_oriented_food_grid = \
        buildGoalOrientedIntegerFoodGridRectangle(
        curr_pos, goal_pos, food_grid)

    max_energy_level_grid = buildMaxEnergyLevelGrid(energy_level,
                                                    goal_oriented_food_grid,
                                                    problem.foodEnergyLevel)
```

```

    shortest_path_length = manhattanDistance(curr_pos, goal_pos)

    if max_energy_level_grid[len(max_energy_level_grid) - 1][
        len(max_energy_level_grid[0]) - 1] < 0:
        # Not enough food on the shortest path. Try with 1 step to
        # a wrong
        # direction
        extended_curr_pos, extended_goal_pos = \
            extendRectangeCornersInEachDirection(
            curr_pos, goal_pos)

        extended_goal_oriented_food_grid = \
            buildGoalOrientedIntegerFoodGridRectangle(
            extended_curr_pos, extended_goal_pos, food_grid)
        # extend max_energy_level matrix with 0s
        max_energy_level_grid = extendMatrixWith0sOnAllSides(
            max_energy_level_grid)

        min_energy_level_grid = buildMinEnergyLevelGrid(
            extended_goal_oriented_food_grid, problem.foodEnergyLevel)

        last_inside_col = len(max_energy_level_grid[0]) - 2
        last_inside_row = len(max_energy_level_grid) - 2

        for i in range(1, last_inside_row):
            for j in range(1, last_inside_col):
                if min_energy_level_grid[i][j - 1] + 1 <= \
                        max_energy_level_grid[i][j]:
                    return shortest_path_length + 2
                if min_energy_level_grid[i - 1][j] + 1 <= \
                        max_energy_level_grid[i][j]:
                    return shortest_path_length + 2

        for i in range(1, last_inside_row):
            if min_energy_level_grid[i][last_inside_col - 1] + 1 <= \
                    max_energy_level_grid[i][last_inside_col]:
                return shortest_path_length + 2
            if min_energy_level_grid[i - 1][last_inside_col] + 1 <= \
                    max_energy_level_grid[i][last_inside_col]:
                return shortest_path_length + 2
            if min_energy_level_grid[i][last_inside_col + 1] + 1 <= \
                    max_energy_level_grid[i][last_inside_col]:
                return shortest_path_length + 2

        for j in range(1, last_inside_col):
            if min_energy_level_grid[last_inside_row - 1][j] + 1 <= \
                    max_energy_level_grid[last_inside_row][j]:
                return shortest_path_length + 2
            if min_energy_level_grid[last_inside_row][j - 1] + 1 <= \
                    max_energy_level_grid[last_inside_row][j]:
                return shortest_path_length + 2
            if min_energy_level_grid[last_inside_row + 1][j] + 1 <= \
                    max_energy_level_grid[last_inside_row][j]:
                return shortest_path_length + 2

        return shortest_path_length + 4
    else:
        # Possibly enough food on the shortest path
        return shortest_path_length
```

```


def hungerGamesManhattanAndStepsOutsideRectangleHeuristic(state,
                                                          problem=None):
    """
    This heuristic takes into consideration how many "incorrect" steps
    (meaning in the wrong direction) does PacMan take to gather all
    the necessary food-dots which fall out of the PacMan-Goal
    rectangle.

    It gives an estimation of the min number of steps in the
    "incorrect"
    direction, by iteratively extending the search rectangle, to cover
    the necessary number of food dots.

    The search rectangle is extended at each step by 1 cell until it
    contains at least the remaining number of food dots required to
    reach the goal from PacMan's current position.

    Heuristic identifier in the documentation: C
    """
    (curr_position, curr_energy_level, food_grid) = state
    goal = problem.mazeExitPosition

    dist_to_exit = manhattanDistance(curr_position, goal)
    needed_energy = dist_to_exit - curr_energy_level

    # if the current energy level is not enough to reach the exit,
    # pacman
    # tries to accumulate food dots along the way;
    # estimate how far does pacman need to step out from the
    # initial shortest
    # path,
    # whose length is given by the manhattan distance;
    if needed_energy > 0 and len(food_grid.asList()) > 0:
        import math
        needed_food = int(
            math.ceil(float(needed_energy) / problem.foodEnergyLevel))
        no_food_dots_inside_rectangle = noFoodDotsInRectange(
            curr_position, goal, food_grid)

        if no_food_dots_inside_rectangle >= needed_food:
            return dist_to_exit
        else:
            remaining_needed_food = needed_food - \
                                    no_food_dots_inside_rectangle
            d = 1
            no_food_dots_outside_rectangle = 0
            rectangle_bottom_left_x = min(curr_position[0], goal[0])
            rectangle_bottom_left_y = min(curr_position[1], goal[1])
            rectangle_top_right_x = max(curr_position[0], goal[0])
            rectangle_top_right_y = max(curr_position[1], goal[1])

            while no_food_dots_outside_rectangle < \
                    remaining_needed_food:
                # extend the perimeter on which we search for food
                if rectangle_bottom_left_x - 1 >= 0:
                    rectangle_bottom_left_x -= 1
                if rectangle_bottom_left_y - 1 >= 0:
                    rectangle_bottom_left_y -= 1
```

```
                if rectangle_top_right_x + 1 < problem.walls.width:
                    rectangle_top_right_x += 1
                if rectangle_top_right_y + 1 < problem.walls.height:
                    rectangle_top_right_y += 1

                # no of food dots on the perimeter at distance d
                no_food_dots_on_perimeter = 0
                for x in range(rectangle_bottom_left_x,
                               rectangle_top_right_x):
                    no_food_dots_on_perimeter += food_grid[x][
                        rectangle_bottom_left_y]
                    no_food_dots_on_perimeter += food_grid[x][
                        rectangle_top_right_y]

                for y in range(rectangle_bottom_left_y + 1,
                               rectangle_top_right_y - 1):
                    no_food_dots_on_perimeter += \
                        food_grid[rectangle_bottom_left_x][y]
                    no_food_dots_on_perimeter += \
                        food_grid[rectangle_top_right_x][y]

                no_food_dots_outside_rectangle += \
                    no_food_dots_on_perimeter
                d += 1

            # pacman had to step out at least 2 times on a distance
            # d from
            # the original rectangle to gather enough food supply
            return dist_to_exit + 2 * d
    else:
        return dist_to_exit


def hungerGamesClosestFoodDotReachableHeuristic(state, problem):
    """
    A heuristic for the HungerGamesSearchProblem, which verifies
    whether any
    food dot is reachable from the current
    position of PacMan with the current energy level.

    If yes, the heuristic returns 0, otherwise infinity (or a very
    large
    value, specified in the problem definition).

    Heuristic identifier in the documentation: G
    """
    (curr_position, curr_energy_level, food_grid) = state
    goal = problem.mazeExitPosition

    if manhattanDistance(goal, curr_position) <= curr_energy_level:
        return 0

    for food_dot in food_grid.asList():
        if manhattanDistance(food_dot,
                             curr_position) <= curr_energy_level:
            return 0

    return HungerGamesSearchProblem\
        .IMPOSSIBLE_TO_SOLVE_STATE_HEURISTIC_VALUE
```

```python

def hungerGamesCombinedHeuristic(state, problem):
    """
    A heuristic for the HungerGamesSearchProblem which combines
    multiple
    previously defined heuristics:
    - if PacMan doesn't have enough energy to reach the goal and
    the closest
    food dot either, then PacMan cannot succeed
    --> based on hungerGamesClosestFoodDotReachableHeuristic,
    a very high
    value is returned
    - if the number of the food dots in the rectangle between the
    current
    position of PacMan and the goal position
    contains enough food dots to cover PacMan's energy requirements
    to the goal,
    assuming a path with manhattan distance steps, then
    hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic(
    state, problem) is returned
    - if the number of the food dots in the rectangle between the
    current
    position of PacMan and the goal position does
    not contain enough food dots to cover PacMan's energy
    requirements to the
    goal, even if
    a path with manhattan distance steps is assumed, then
    hungerGamesManhattanAndStepsOutsideRectangleHeuristic(state,
    problem) is
    returned.

    Heuristic identifier in the documentation: H
    """
    (curr_position, curr_energy_level, food_grid) = state
    goal = problem.mazeExitPosition

    dist_to_exit = hungerGamesManhattanHeuristic(state, problem)
    needed_energy = dist_to_exit - curr_energy_level

    if needed_energy > 0 and \
            hungerGamesClosestFoodDotReachableHeuristic(
            state,
            problem) == \
            HungerGamesSearchProblem\
                    .IMPOSSIBLE_TO_SOLVE_STATE_HEURISTIC_VALUE:
        return HungerGamesSearchProblem\
            .IMPOSSIBLE_TO_SOLVE_STATE_HEURISTIC_VALUE

    needed_food = needed_energy // problem.foodEnergyLevel
    no_food_dots_inside_rectangle = noFoodDotsInRectange(
        curr_position, goal, food_grid)

    if no_food_dots_inside_rectangle >= needed_food:
        return \

   hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic(
            state, problem)
    else:
        # if the current energy level is not enough to reach the exit,
        # pacman tries to accumulate food dots along the way;
```

```
        # estimate how far does pacman need to step out from the
        # initial
        # shortest path,
        # whose length is given by the manhattan distance;
        return hungerGamesManhattanAndStepsOutsideRectangleHeuristic(
            state, problem)
```

Listing A.1: searchAgents.py - Our Original Code

```

"""======================START OF MY OWN CODE======================"""
HIGH_ENERGY_COLOR = formatColor(.0, .9, .9)
MEDIUM_ENERGY_COLOR = formatColor(1.0, 0.6, 0.0)
LOW_ENERGY_COLOR = formatColor(.98, .41, .07)
CRITICAL_ENERGY_COLOR = formatColor(.9, 0, 0)
ENERGY_BAR_OUTLINE_COLOR = formatColor(.99, .99, .99)
# Exit-door graphics
MAZE_EXIT_COLOR = formatColor(0, 1, 1)
MAZE_EXIT_SIZE = 0.5
"""=====================END OF MY OWN CODE========================="""

class InfoPane:

"""==================START OF MY OWN CODE======================"""

    def drawPane(self, energyLevel):
        self.energyLevelText = text(self.toScreen(0, 0),
                                    HIGH_ENERGY_COLOR,
                                    "ENERGY:    " + str(energyLevel),
                                    "Times", self.fontSize, "bold")
        self.drawEnergyLevelIndicator()

    def drawEnergyLevelIndicator(self):
        width = self.width * 0.03
        height = self.height * 0.3
        space = self.width * 0.005
        startPosX, startPosY = (self.width * 0.7, height)
        self.energyLevelBars = []
        self.energyLevelBars.append(
            rectangle(self.toScreen(startPosX, startPosY), width,
                      height, ENERGY_BAR_OUTLINE_COLOR,
                      HIGH_ENERGY_COLOR))
        self.energyLevelBars.append(rectangle(
            self.toScreen(startPosX + 2 * width + space, startPosY),
            width, height, ENERGY_BAR_OUTLINE_COLOR,
            HIGH_ENERGY_COLOR))
        self.energyLevelBars.append(rectangle(
            self.toScreen(startPosX + 4 * width + 2 * space,
                          startPosY), width, height,
            ENERGY_BAR_OUTLINE_COLOR, HIGH_ENERGY_COLOR))
        self.energyLevelBars.append(rectangle(
            self.toScreen(startPosX + 6 * width + 3 * space,
                          startPosY), width, height,
            ENERGY_BAR_OUTLINE_COLOR, HIGH_ENERGY_COLOR))

    def updateEnergyLevel(self, energyLevel, maxEnergyLevel):
        changeText(self.energyLevelText, "ENERGY: % 4d" % energyLevel)

        if energyLevel >= maxEnergyLevel * 0.75:
            for bar in self.energyLevelBars:
                changeColor(bar, HIGH_ENERGY_COLOR)
```

```
            changeColor(self.energyLevelText, HIGH_ENERGY_COLOR)
        elif energyLevel >= maxEnergyLevel * 0.5:
            changeColor(self.energyLevelBars[0], MEDIUM_ENERGY_COLOR)
            changeColor(self.energyLevelBars[1], MEDIUM_ENERGY_COLOR)
            changeColor(self.energyLevelBars[2], MEDIUM_ENERGY_COLOR)
            changeColor(self.energyLevelBars[3], BACKGROUND_COLOR)
            changeColor(self.energyLevelText, MEDIUM_ENERGY_COLOR)
        elif energyLevel >= maxEnergyLevel * 0.25:
            changeColor(self.energyLevelBars[0], LOW_ENERGY_COLOR)
            changeColor(self.energyLevelBars[1], LOW_ENERGY_COLOR)
            changeColor(self.energyLevelBars[2], BACKGROUND_COLOR)
            changeColor(self.energyLevelBars[3], BACKGROUND_COLOR)
            changeColor(self.energyLevelText, LOW_ENERGY_COLOR)
        else:
            changeColor(self.energyLevelBars[0],
                        CRITICAL_ENERGY_COLOR)
            changeColor(self.energyLevelBars[1], BACKGROUND_COLOR)
            changeColor(self.energyLevelBars[2], BACKGROUND_COLOR)
            changeColor(self.energyLevelBars[3], BACKGROUND_COLOR)
            changeColor(self.energyLevelText, CRITICAL_ENERGY_COLOR)

    def drawMazeExit(self, exit_pos):
        (screen_x, screen_y) = self.to_screen(exit_pos)
        outerSquare = square((screen_x, screen_y),
                             MAZE_EXIT_SIZE * self.gridSize,
                             color=MAZE_EXIT_COLOR, filled=0)
        innerSquare = square((screen_x + 0.25, screen_y + 0.25),
                             MAZE_EXIT_SIZE * self.gridSize * 0.5,
                             color=MAZE_EXIT_COLOR, filled=1)
        imageParts = []
        imageParts.append(outerSquare)
        imageParts.append(innerSquare)
        return imageParts

    """====================END OF MY OWN CODE======================"""
```

Listing A.2: graphicsDisplay.py - Our Original Code

```
"""===================START OF CODE MODIFIED BY ME============="""

def rectangle(pos, rx, ry, outlineColor, fillColor, filled=1,
              behind=0, width=1):
    x, y = pos
    coords = [(x - rx, y - ry), (x + rx, y - ry), (x + rx, y + ry),
              (x - rx, y + ry)]
    return polygon(coords, outlineColor, fillColor, filled, 0,
                   behind=behind, width=width)

"""===================END OF CODE MODIFIED BY ME============="""
```

Listing A.3: graphicsUtils.py - Modified Original Code

```

###################################################
# YOUR INTERFACE TO THE PACMAN WORLD: A GameState #
###################################################


"""=========================START OF MY OWN CODE==================="""
DEFAULT_INITIAL_PACMAN_ENERGY_LEVEL = 10
DEFAULT_FOOD_ENERGY_LEVEL = 3
"""=========================START OF MY OWN CODE==================="""
```

```
class GameState:
"""===================START OF CODE MODIFIED BY ME============="""
    def initialize(self, layout, pacmanEnergyLevel, foodEnergyLevel,
                   numGhostAgents=1000, ):
        """
        Creates an initial game state from a layout array (see
        layout.py).
        """
        self.data.initialize(layout, numGhostAgents,
                             pacmanEnergyLevel, foodEnergyLevel)
    """====================END OF CODE MODIFIED BY ME=============="""


class ClassicGameRules:
    """
    These game rules manage the control flow of a game, deciding when
    and how the game starts and ends.
    """

    def __init__(self, timeout=30):
        self.timeout = timeout

    """==================START OF CODE MODIFIED BY ME=============="""

    def newGame(self, layout, pacmanAgent, ghostAgents,
                pacmanEnergyLevel, foodEnergyLevel, display,
                quiet=False, catchExceptions=False):
        agents = [pacmanAgent] + ghostAgents[:layout.getNumGhosts()]
        initState = GameState()
        initState.initialize(layout,
                             pacmanEnergyLevel=pacmanEnergyLevel,
                             foodEnergyLevel=foodEnergyLevel,
                             numGhostAgents=len(ghostAgents), )
        game = Game(agents, display, self,
                    catchExceptions=catchExceptions)
        game.state = initState
        self.initialState = initState.deepCopy()
        self.quiet = quiet
        return game

    """=================END OF CODE MODIFIED BY ME================"""

############################
# FRAMEWORK TO START A GAME #
############################

def readCommand(argv):
    """
    Processes the command used to run pacman from the command line.
    """
    from optparse import OptionParser
    usageStr = """
    USAGE:      python pacman.py <options>
    EXAMPLES:   (1) python pacman.py
                    - starts an interactive game
                (2) python pacman.py --layout smallClassic --zoom 2
                OR  python pacman.py -l smallClassic -z 2
                    - starts an interactive game on a smaller
                    board, zoomed in
```

```
    """
    parser = OptionParser(usageStr)

    parser.add_option('-n', '--numGames', dest='numGames', type='int',
                      help=default('the number of GAMES to play'),
                      metavar='GAMES', default=1)
    parser.add_option('-l', '--layout', dest='layout', help=default(
        'the LAYOUT_FILE from which to load the map layout'),
                      metavar='LAYOUT_FILE', default='mediumClassic')
    parser.add_option('-p', '--pacman', dest='pacman', help=default(
        'the agent TYPE in the pacmanAgents module to use'),
                      metavar='TYPE', default='KeyboardAgent')
    parser.add_option('-t', '--textGraphics', action='store_true',
                      dest='textGraphics',
                      help='Display output as text only',
                      default=False)
    parser.add_option('-q', '--quietTextGraphics',
                      action='store_true', dest='quietGraphics',
                      help='Generate minimal output and no graphics',
                      default=False)
    parser.add_option('-g', '--ghosts', dest='ghost', help=default(
        'the ghost agent TYPE in the ghostAgents module to use'),
                      metavar='TYPE', default='RandomGhost')
    parser.add_option('-k', '--numghosts', type='int',
                      dest='numGhosts', help=default(
            'The maximum number of ghosts to use'), default=4)
    parser.add_option('-z', '--zoom', type='float', dest='zoom',
                      help=default(
                          'Zoom the size of the graphics window'),
                      default=1.0)
    parser.add_option('-f', '--fixRandomSeed', action='store_true',
                      dest='fixRandomSeed',
                      help='Fixes the random seed to always play '
                           'the same game',
                      default=False)
    parser.add_option('-r', '--recordActions', action='store_true',
                      dest='record',
                      help='Writes game histories to a file (named '
                           'by the time they were played)',
                      default=False)
    parser.add_option('--replay', dest='gameToReplay',
                      help='A recorded game file (pickle) to replay',
                      default=None)
    parser.add_option('-a', '--agentArgs', dest='agentArgs',
                      help='Comma separated values sent to agent. '
                           'e.g. "opt1=val1,opt2,opt3=val3"')
    parser.add_option('-x', '--numTraining', dest='numTraining',
                      type='int', help=default(
            'How many episodes are training (suppresses output)'),
                      default=0)
    parser.add_option('--frameTime', dest='frameTime', type='float',
                      help=default(
                          'Time to delay between frames; <0 means '
                          'keyboard'),
                      default=0.1)
    parser.add_option('-c', '--catchExceptions', action='store_true',
                      dest='catchExceptions',
                      help='Turns on exception handling and '
                           'timeouts during games',
                      default=False)
```

```python
    parser.add_option('--timeout', dest='timeout', type='int',
                      help=default(
                          'Maximum length of time an agent can spend '
                          'computing in a single game'),
                      default=30)

    options, otherjunk = parser.parse_args(argv)
    if len(otherjunk) != 0:
        raise Exception(
            'Command line input not understood: ' + str(otherjunk))
    args = dict()

    # Fix the random seed
    if options.fixRandomSeed: random.seed('cs188')

    # Choose a layout
    args['layout'] = layout.getLayout(options.layout)
    if args['layout'] == None: raise Exception(
        "The layout " + options.layout + " cannot be found")

    # Choose a Pacman agent
    noKeyboard = options.gameToReplay == None and (
            options.textGraphics or options.quietGraphics)
    pacmanType = loadAgent(options.pacman, noKeyboard)
    agentOpts = parseAgentArgs(options.agentArgs)
    if options.numTraining > 0:
        args['numTraining'] = options.numTraining
        if 'numTraining' not in agentOpts: agentOpts[
            'numTraining'] = options.numTraining

    """======================START OF MY OWN CODE================="""

    if 'prob' in agentOpts and agentOpts[
        'prob'] == 'HungerGamesSearchProblem':
        if 'pacman_energy_level' in agentOpts:
            args['pacmanEnergyLevel'] = int(
                agentOpts['pacman_energy_level'])
        if 'food_energy_level' in agentOpts:
            args['foodEnergyLevel'] = int(
                agentOpts['food_energy_level'])

    """========================END OF MY OWN CODE================="""

    pacman = pacmanType(
        **agentOpts)  # Instantiate Pacman with agentArgs
    args['pacman'] = pacman

    # Don't display training games
    if 'numTrain' in agentOpts:
        options.numQuiet = int(agentOpts['numTrain'])
        options.numIgnore = int(agentOpts['numTrain'])

    # Choose a ghost agent
    ghostType = loadAgent(options.ghost, noKeyboard)
    args['ghosts'] = [ghostType(i + 1) for i in
                      range(options.numGhosts)]

    # Choose a display format
    if options.quietGraphics:
        import textDisplay
```

```python
        args['display'] = textDisplay.NullGraphics()
    elif options.textGraphics:
        import textDisplay
        textDisplay.SLEEP_TIME = options.frameTime
        args['display'] = textDisplay.PacmanGraphics()
    else:
        import graphicsDisplay
        args['display'] = graphicsDisplay.PacmanGraphics(
            options.zoom, frameTime=options.frameTime)
    args['numGames'] = options.numGames
    args['record'] = options.record
    args['catchExceptions'] = options.catchExceptions
    args['timeout'] = options.timeout

    # Special case: recorded games don't use the runGames method or
    # args structure
    if options.gameToReplay != None:
        print 'Replaying recorded game %s.' % options.gameToReplay
        import cPickle
        f = open(options.gameToReplay)
        try:
            recorded = cPickle.load(f)
        finally:
            f.close()
        recorded['display'] = args['display']
        replayGame(**recorded)
        sys.exit(0)

    return args
```

Listing A.4: pacman.py - Modified Original Code

```python

"""===================START OF CODE MODIFIED BY ME============="""

class GameStateData:
    """
    Added additional attributes to the GameState, which help us
    monitor the energy level of PacMan.
    - pacmanEnergyLevel = current energy level of PacMan at a given
    step in the game
    - initialEnergyLevel = initial energy level of PacMan at the
    start of the game
    - foodEnergyLevel = the gain in energy obtained from eating a
    food-dot
    """

    def __init__(self, prevState=None):
        """
        Generates a new data packet by copying information from its
        predecessor.
        """
        if prevState != None:
            self.food = prevState.food.shallowCopy()
            self.capsules = prevState.capsules[:]
            self.agentStates = self.copyAgentStates(
                prevState.agentStates)
            self.layout = prevState.layout
            self._eaten = prevState._eaten
            self.score = prevState.score
            """=================START OF MY OWN CODE=============="""
```

```
            self.pacmanEnergyLevel = prevState.pacmanEnergyLevel
            self.initialEnergyLevel = prevState.initialEnergyLevel
            self.foodEnergyLevel = prevState.foodEnergyLevel

            """==================END OF MY OWN CODE================"""

        self._foodEaten = None
        self._foodAdded = None
        self._capsuleEaten = None
        self._agentMoved = None
        self._lose = False
        self._win = False
        self.scoreChange = 0

    def deepCopy(self):
        state = GameStateData(self)
        state.food = self.food.deepCopy()
        state.layout = self.layout.deepCopy()
        """==================START OF MY OWN CODE==================="""

        state.pacmanEnergyLevel = self.pacmanEnergyLevel
        state.initialEnergyLevel = self.initialEnergyLevel
        state.foodEnergyLevel = self.foodEnergyLevel

        """==================END OF MY OWN CODE====================="""
        state._agentMoved = self._agentMoved
        state._foodEaten = self._foodEaten
        state._foodAdded = self._foodAdded
        state._capsuleEaten = self._capsuleEaten
        return state

    def copyAgentStates(self, agentStates):
        copiedStates = []
        for agentState in agentStates:
            copiedStates.append(agentState.copy())
        return copiedStates

    def __eq__(self, other):
        """
        Allows two states to be compared.
        """
        if other == None: return False
        # TODO Check for type of other
        if not self.agentStates == other.agentStates: return False
        if not self.food == other.food: return False
        if not self.capsules == other.capsules: return False
        if not self.score == other.score: return False
        """=================START OF MY OWN CODE=================="""

        if not self.initialEnergyLevel == other.initialnergyLevel:
            return False
        if not self.pacmanEnergyLevel == other.pacmanEnergyLevel:
            return False
        if not self.foodEnergyLevel == other.foodEnergyLevel:
            return False

        """=================END OF MY OWN CODE===================="""
        return True
```

```python

    """===================START OF CODE MODIFIED BY ME============="""
    def initialize(self, layout, numGhostAgents, pacmanEnergyLevel,
                   foodEnergyLevel):
        """
        Creates an initial game state from a layout array (see
        layout.py).

        It also initializes the additional variables which monitor
        the relevant energy levels:

        pacmanEnergyLevel = the current energy level of PacMan
        which is updated after every move
        initialEnergyLevel = the initial energy level passed as a
        command line argument when running the program;
                            it is needed as reference to visualize
                            the remaining energy level with the
                            energy level
                            indicator bar on the right of the frame.
        foodEnergyLevel = the gain in energy when PacMan eats a
        food dot; also initialized by a program argument,
                            and its value remains the same
                            throughout the game
        """
        self.food = layout.food.copy()
        # self.capsules = []
        self.capsules = layout.capsules[:]
        self.layout = layout
        self.score = 0
        self.scoreChange = 0
        """==================START OF MY OWN CODE============="""

        self.initialEnergyLevel = pacmanEnergyLevel
        self.pacmanEnergyLevel = pacmanEnergyLevel
        self.foodEnergyLevel = foodEnergyLevel

        """===================END OF MY OWN CODE=============="""
        self.agentStates = []
        numGhosts = 0
        for isPacman, pos in layout.agentPositions:
            if not isPacman:
                if numGhosts == numGhostAgents:
                    continue  # Max ghosts reached already
                else:
                    numGhosts += 1
            self.agentStates.append(
                AgentState(Configuration(pos, Directions.STOP),
                           isPacman))
        self._eaten = [False for a in self.agentStates]
    """===================END OF CODE MODIFIED BY ME============="""
```
Listing A.5: game.py - Modified Original Code