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 Position-SearchProblem.

   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 Pac-Man
- **heuristic** = which heuristic function should be used by the A* search algorithm

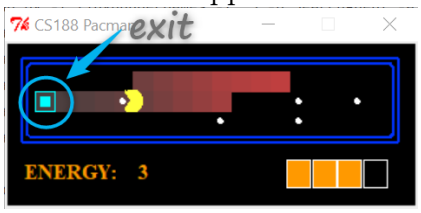
Example:
Run the game on layout tinyHungerGames.lay with initial energy level for PacMan of 7 and food energy level of 2, using the hungerGamesCombinedHeuristic:

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

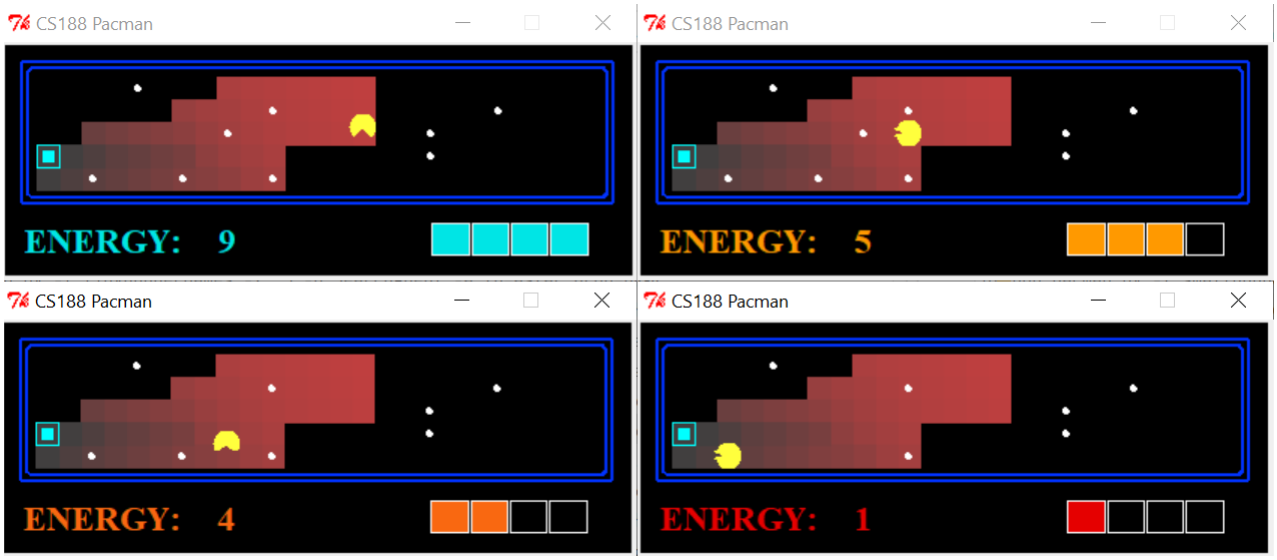
Some changes were made to the interface of the game in order to confirm with the new constraints.

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



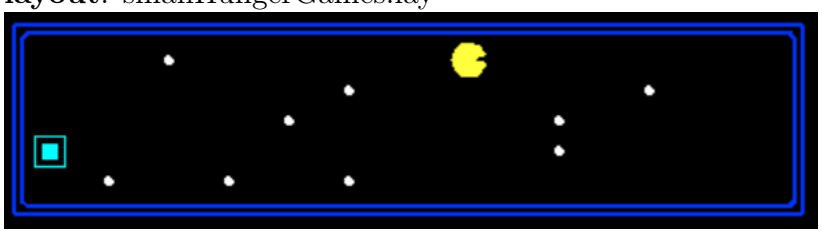
At each moment of the game, we display the current value of PacMan's energy level along with an indicator, which shows approximately how much energy he has left:

- 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



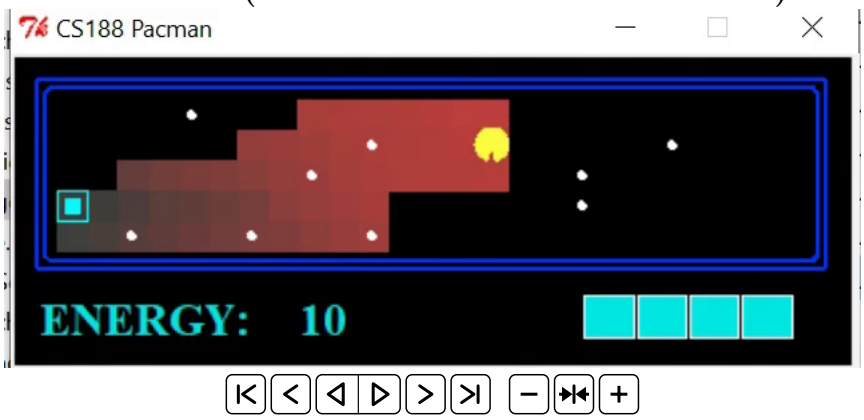
pacman_energy_level: 10

food_energy_level: 3

Compare heuristics:

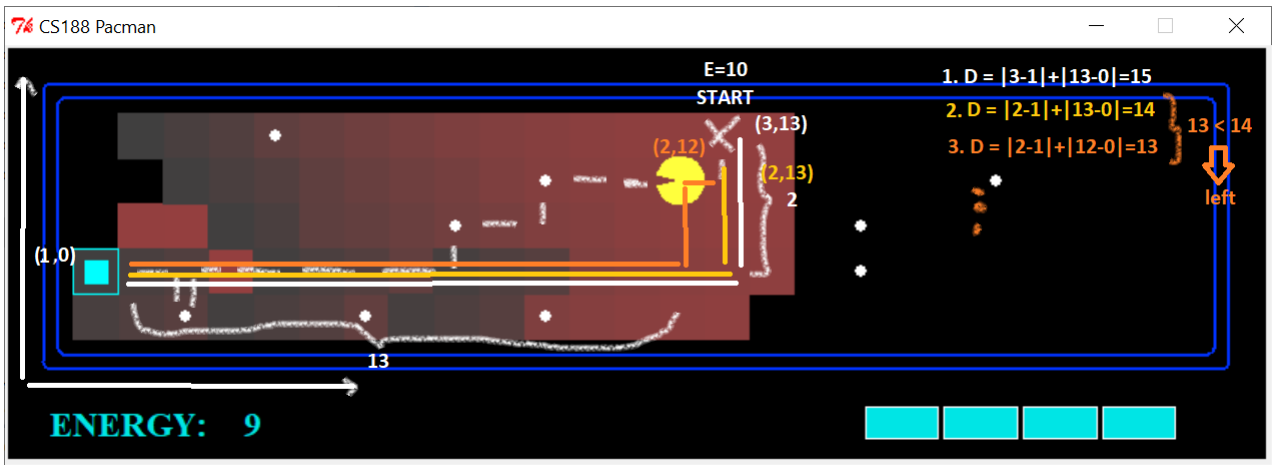
Manhattan distance:

Final heuristic (combination of 3 other heuristics):

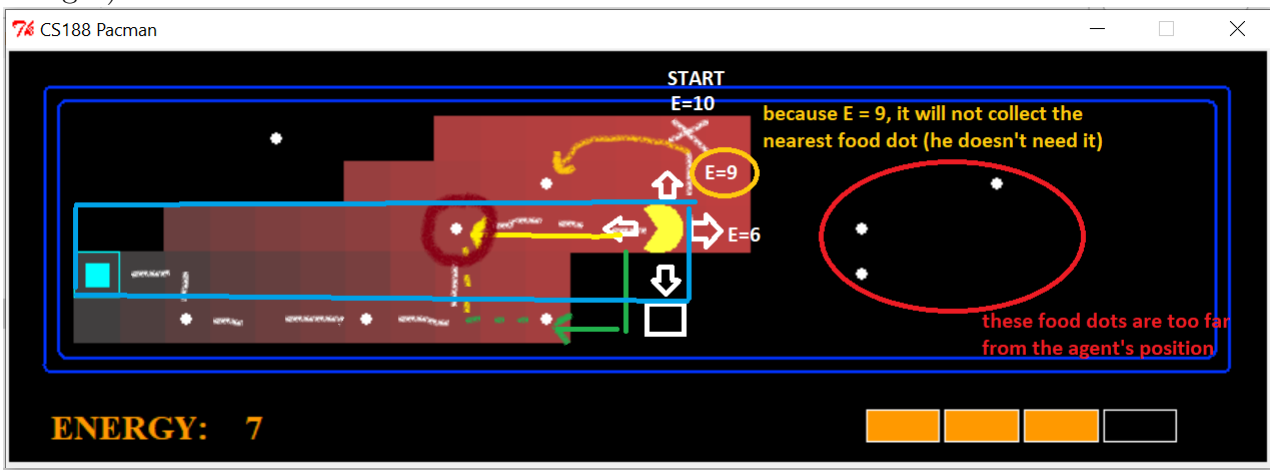


Explain what happened:

- 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

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

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"

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

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 →**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

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

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 nth**, then PacMan must take at least n incorrect-direction steps, because PacMan needs to reach at least one position contained in the nth but not contained in the (n-1)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

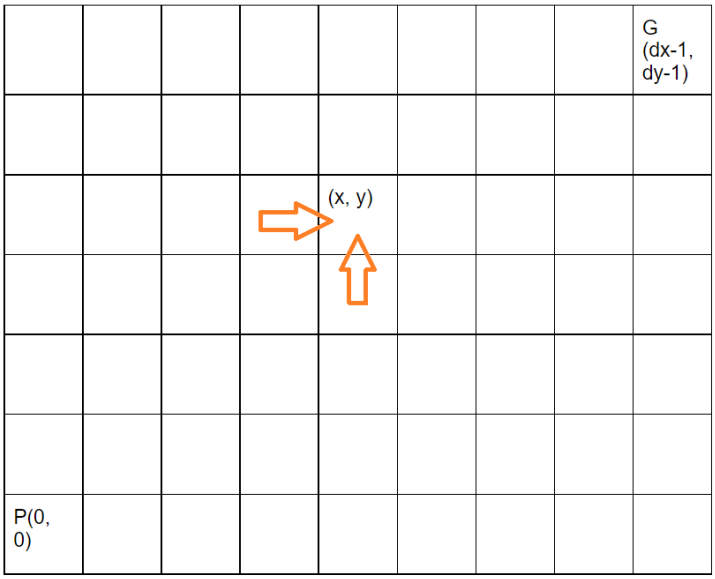


Figure 1.3: Matrix M. P denotes Pacman's position, G the goal's position. The arrows show the steps from whichh position (x, y) may be reached as part of an ideal path.

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

(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*

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

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:

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

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

$max\_energy\_level\_at[dx][dy] >= 0 \iff$

$\exists$ a valid path of cost ManhattanDistace from PacMan's current position to the goal position, considering all the constraints of the HungerGamesSearchProblem

(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

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

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

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

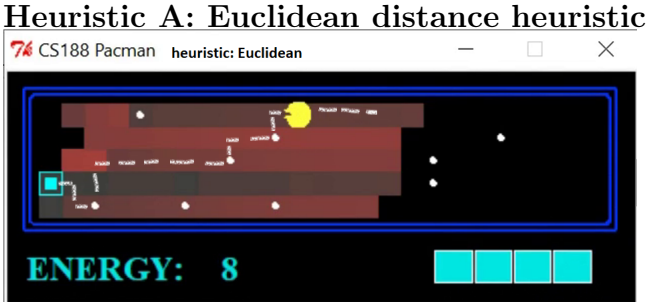
(a) return the value of heuristic C

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

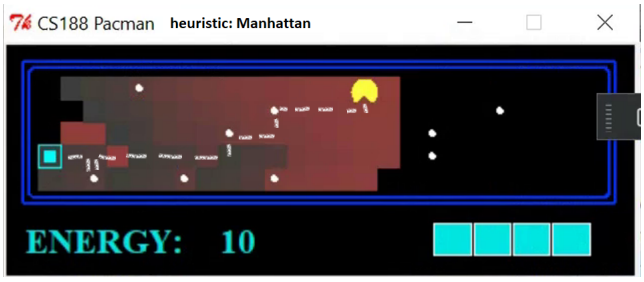
1. **Heuristic A: Euclidean distance heuristic**



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.

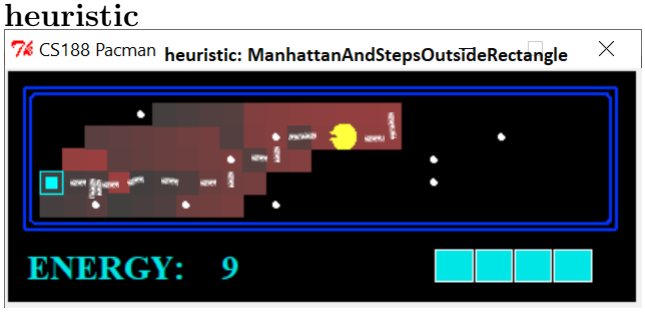
2. **Heuristic B: Manhattan distance heuristic**



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**

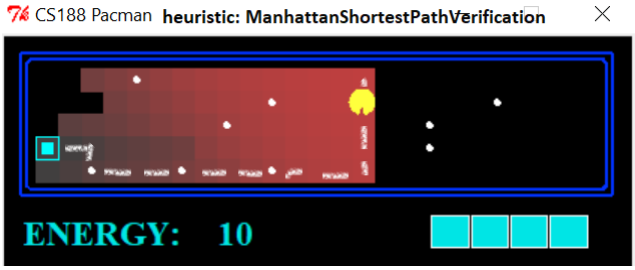


In this case, the heuristic considers how many steps would PacMan need to make outside the rectangle determined by its current position and the Exit's position, which obviously would add to the total traversed distance.
Therefore it will select a path that guarantees the min number of necessary food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

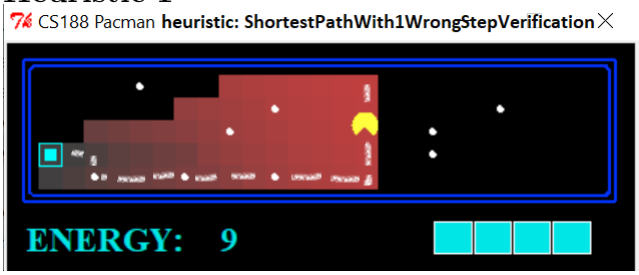
If there not enough food dots along the shortest path, then PacMan needs to leave the **goal rectangle**, and search outside it for food. That means he makes 1 move in the direction opposite to the goal, but as in the end he needs to arrive to the goal, it means when he comes back to the correct path, 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**

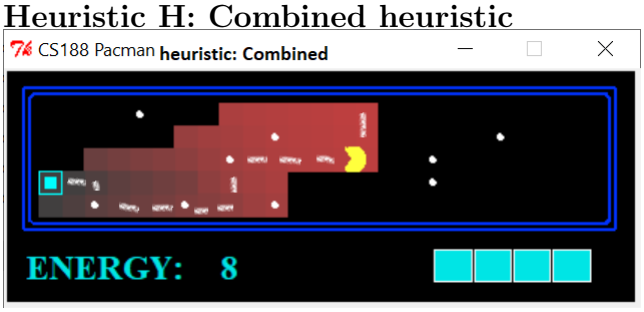


Similar to the previous case, but with fewer nodes expanded, which is due to the fact that it also considers whether PacMan makes more than 2 additional steps when he steps off of the shortest path and the rectangle to get to the food dots.

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.

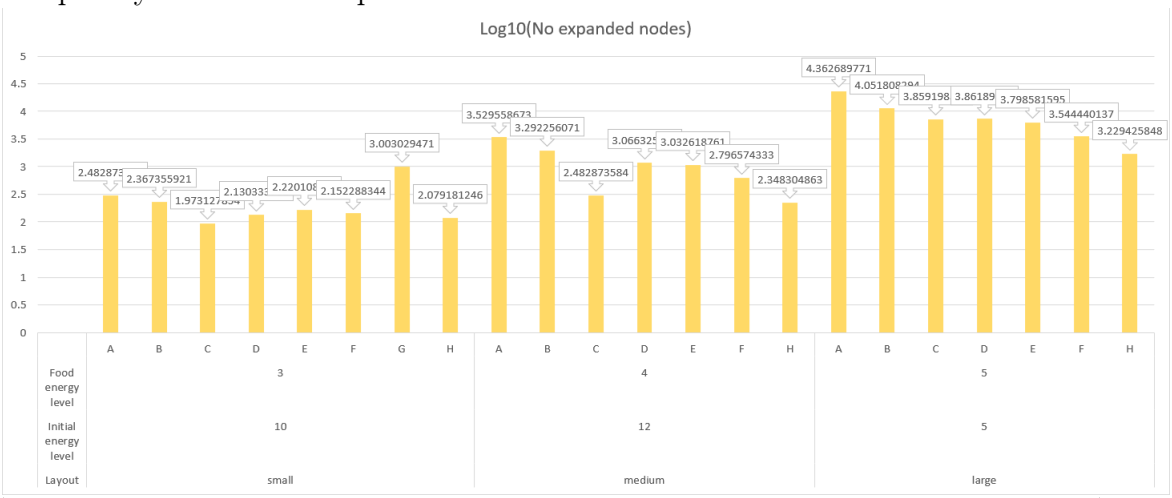
Notations:

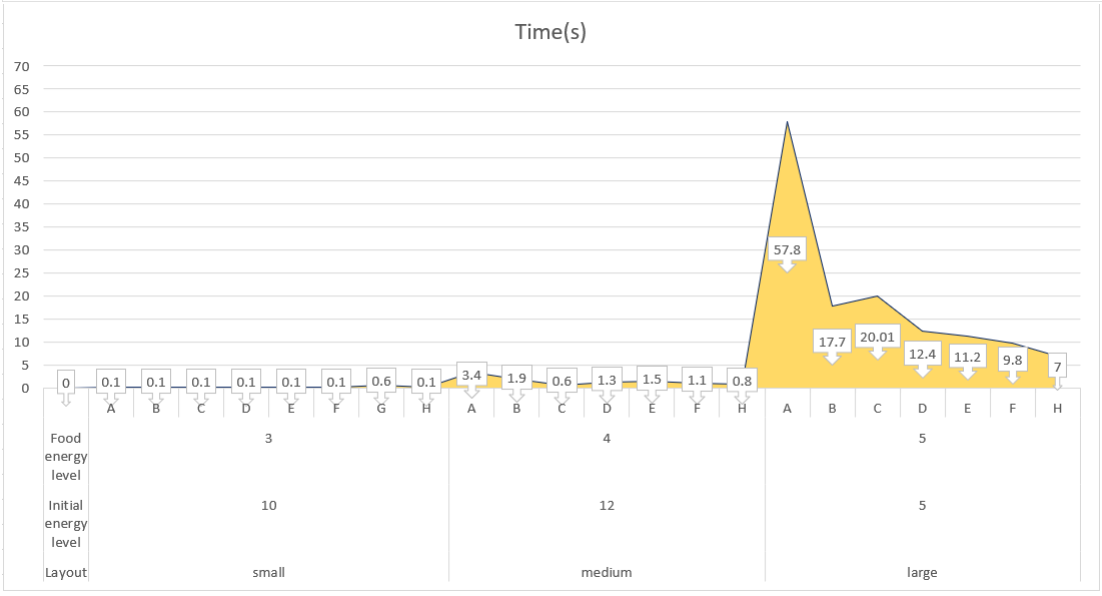
- A = EuclideanHeuristic
- B = ManhattanHeuristic
- C = hungerGamesManhattanAndStepsOutsideRectangleHeuristic
- D = hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic
- E = hungerGamesManhattanShortestPathVerificationHeuristic
- F = hungerGamesManhattanShortestPathWith1WrongStepVerificationHeuristic
- 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.

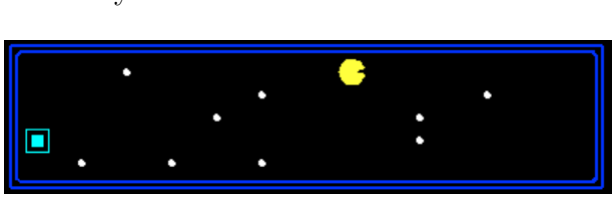
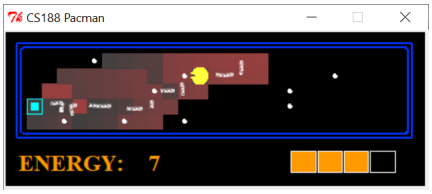




1.1.3.1 Conclusions based on the measurements:

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:

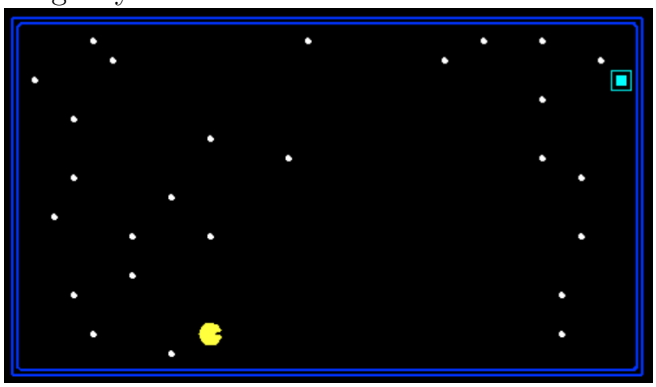



For the small maze the best results were obtained when running the algorithm with heuristic C.

- Medium layout:

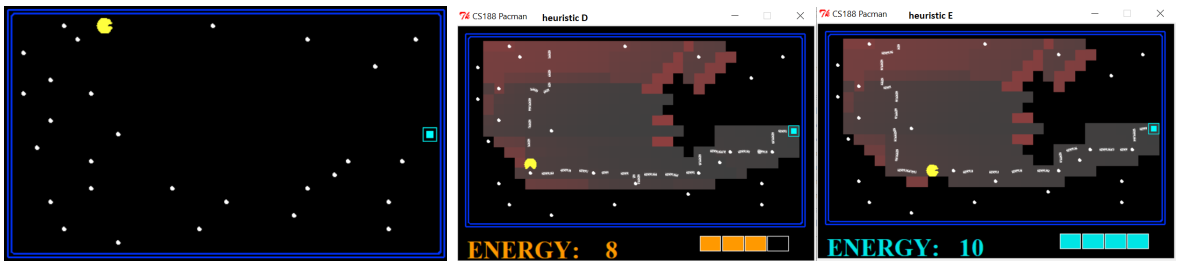

The C heuristic produced very good results for the medium layout as well, in which the food dots were placed close to the perimeter of the "safe" rectangle (determined by the Manhattan distance) and PacMan didn't have to wonder off from the shortest path very far to gather the foods, almost as good in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

- Large layout:



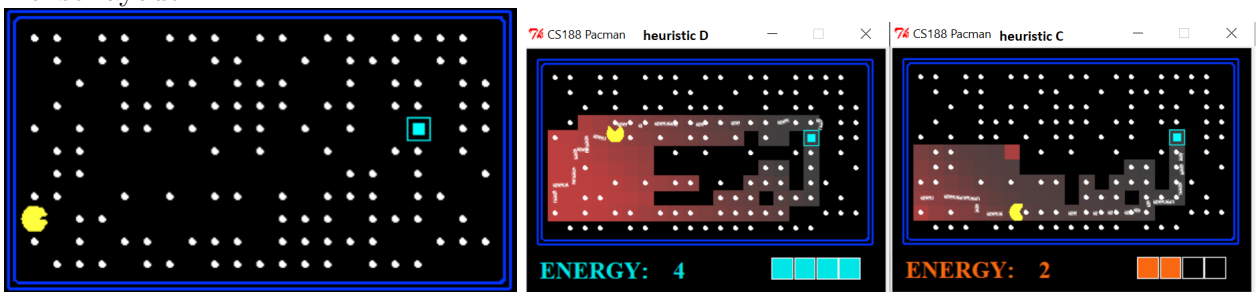

In case of the large layout, the H heuristic(*hungerGamesCombinedHeuristic*) produced the best results. It's interesting to note though, that for a relatively random, large layout, heuristic F produced twice better results than heuristic C.

- 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

2.1 The Lights Out Game

2.1.1 Problem Statement

Lights Out is a puzzle in which you are given a grid of tiles (or lights), some of them lit up others turned off. The goal is to turn off all the lights in the grid by clicking on the tiles, in as few moves as possible. Each click toggles that light and its neighbours in a cross-like pattern.

2.1.1.1 Background. Game Rules.

2.1.1.1.1 Short history

The classic *Lights Out* electronic puzzle, released by Tiger Electronics in 1995, consisted of a 5x5 grid of buttons, each containing a led, which is toggled on each press, along with its non-diagonal neighbours (left, right, top, bottom). Other similar games from that period were the XL-25, produced by Vulcan Electronics in 1983, or Merlin, by Parker Brothers in 1970s, which was played on a 3x3 grid. Other variations of the *Lights out* puzzle were also available such as the *Lights Out 2000*, a 5x5 grid but with multiple colours, than the *Lights Out Cube*, where each face of the cube consisted of a 3x3 grid of lights, and the *Lights Out Deluxe*, which was played on a 6x6 grid.

Figure 2.1: Lights out electronic puzzle

2.1.1.1.2 Gameplay

- **Game state:**
  The lights are arranged in a 5x5 grid. When a new game starts, a random pattern of lights is switched on and displayed. Each of the lights has only 2 states: On or Off.

- **Goal state:**
  The game ends when all the lights are successively turned off, preferably with the smallest possible number of moves.

- **Allowed steps:**
  The player can click on any button, and as an effect, it will toggle not only the state of the selected light but also that of its direct (non-diagonal) neighbours.

2.1.1.1.3 An inefficient but sure way to win: Light chasing

The method known as "Light chasing", similar to Gaussian elimination, guarantees to solve the problem, if there is a solution, but it might require many redundant steps to achieve this. It successively scans each row, starting with the second one, and clicks on every cell which has a light above it, on the previous row. The last row is handled separately, using a lookup table with some predefined light patterns in order to identify which lights should be toggled in the first row. After that the initial algorithm is applied again starting from the second row, until a solution is found.

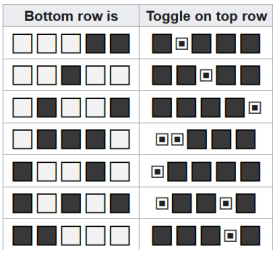
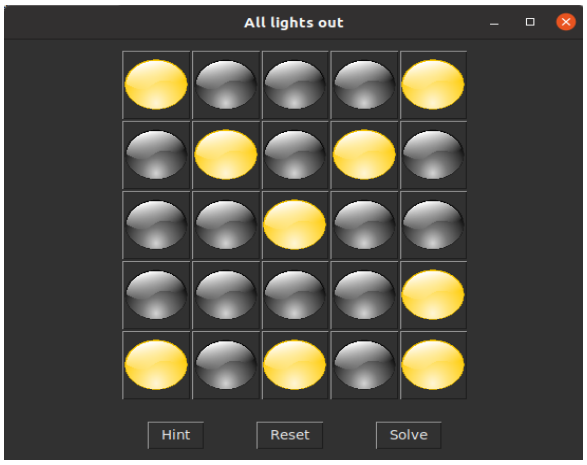


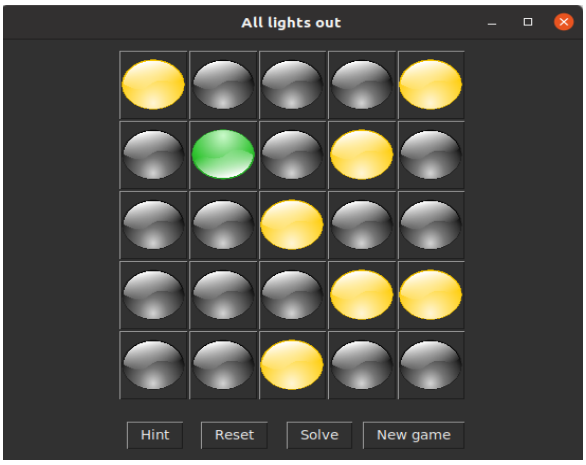
Figure 2.2: Lookup table for last row

2.1.1.2 The GUI. Main functionalities.

The player is provided a board consisting of the 5x5 grid of light bulbs. When the game starts an initial pattern of lights is loaded, and mapped on the grid, so that some of the lights would appear as turned on (bright) or turned off(dark).
The player can click on any of the lights and as a result it will flip their state and that of its neighbours on a cross-pattern. The lights which were on will be turned off and the lights which were off will be turned on.



(a) Game board representation



(b) Show hint for next step

Figure 2.3: The user interface for the Lights Out game

The player can also use the set of buttons, having the following functionalities:

- **Hint:** indicates a valid step, which would bring the player closer to the solution. An animation was added to show which light could be toggled next. The order of the moves is not relevant, the given hint represents only one possible move out of the set of moves for solving the whole grid. It will have no effect once the game is already solved.
- **Reset:** resets the grid to its initial state, which was loaded at the start of the game.
- **Solve:** presents a visualization of the solution for the given grid configuration, by tracing it step-by-step, indicating which tiles should be clicked and updating the grid after each move until the final solution.
- **New game:** starts a new game session by loading a new board configuration from a set of pre-generated layouts.

When the player wins the game (finds a valid solution) or they click on the *Solve* button and the animation reaches the last step, a new window pops up displaying the results of the game (score), namely the number of moves it took to reach a solution.

2.1.1.3 How to get started?

The application is built on top of an mvc pattern, which means it can be run by first loading the view, GameDisplay class, which is in turn linked to the controller, represented by the GameService class and it displays the current state of the model, as defined in the GameState class.
The command for running the application is:

```
$ python GameDisplay.py
```

The application was implemented using Python2, therefore it is needed to have the earlier version of Python installed in order to run it without problems.

For generating new game configurations and finding solutions for a given light pattern, we used the Mace4 program. Mace4 can be used to search for finite models of first-order formulas.

For installing prover9 and mace4 download the tar package from the following url: `https://www.cs.unm.edu/~mccune/mace4/download/`
Then unpack it usng the command:

```
$ tar -xvzf LADR-2009-11A.tar.gz
```

And then run the following code from the LADR directory:

```
$ make all
```

The GUI of the application was implemented using modules such as tkinter and pillow, which needs to be installed in order to display the interface of the game. Another requirement is to have a later version of tk, preferably tk 8.6, in order to guarantee support for all the features used in the application, such as loading images (.png).
The command for installing PIL is the following:

```
$ pip install Pillow
```

The following command installs the latest version of tk:

```
$ sudo apt-get install python-tk
```

2.1.2 Methodology and Implementation

The methodology and the implementation shall be discussed in two separate sections, based on the natural structure of the project.

- *Logics Part*: in the background, the logical challenges of the game are solved via automated theorem provers and model finders, based on first order logic. The problems solved with logics are
  - Generating solvable puzzles.
  - Finding the solutions for the current state of the game.
- *Interface Part*: on the surface, the game has an interactive graphical user interface implemented in python, which controls the reactions to the user actions and runs the Mace4 and Prover9 codes when necessary. Main functionalities:
  - Allow the user to start a new game session, with a new puzzle. →Relying on the logics section for puzzle generation.
  - Allow the user to switch a light bulb. →Fully implemented in python.
  - Verifying whether the user won the game. Display a message if so, with the number of steps taken to reach the winning state. →Fully implemented in python.
  - Provide a hint to the user. →Relying on the logics section to find the solutions for the puzzle with non-redundant steps.
  - Allow the user to restart the current game session, to the initial state of the puzzle. →Fully implemented in python.

2.1.2.1 Logics Part

As stated above, for implementing the game, there were two problems that were worth tackling with Mace4 or Prover9:

- Solving puzzles.
- Generating solvable puzzles.

2.1.2.1.1 Solving puzzles

Goal: given the current game state, represented by the state of each light bulb in the 5x5 grid, find a solution defined by the sequence of steps which lead to the winning state of the game.

Usage: the shortest solution is needed to provide hints for the user.

Note that it is not the goal to find the shortest solution based on logics, we aim only to find any solution. In the description of Approach C you'll find out why this is not a requirement.

Definitions:

- **Solution** = a sequence of (x_i, y_i) steps, meaning that the i^{th} step of the solution is switching the light bulb in position (x_i, y_i) of the grid, and by applying all steps from 0 to n-1, where n is the solution length, we get to the winning state.
- **Solution Length** = the length of the sequence of steps defined by the solution.
- **Shortest Solution** = the solution with the least number of steps.

2.1.2.1.2 Approach A: naive solution generation with Prover9 in Production Mode

A naive solution can be found directly from the game rules: knowing the current state of the game session, represented by state of each light bulb in the grid, use backtracking to find a sequence of allowed steps that lead to the winning state. Such a problem can be easily expressed in prover9, with production mode, we only need to define

- **The representation of a game state: Numeric Row Vector representation**: given that a puzzle consists of a 5x5 grid of the states of the light bulbs, we can expand this 5x5 boolean matrix into a 1x25 list V, with
  - V[i] = 0 →the light bulb in row i/5 and column i%5 is off
  - V[i] = 1 →the light bulb in row i/5 and column i%5 is on

For example,

$$V = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{2.1}$$

(where V is a row vector formatted on multiple rows) corresponds to the Game state

$$\begin{bmatrix} \text{Off Off Off Off Off} \\ \text{Off Off Off Off Off} \\ \text{Off Off Off Off Off} \\ \text{Off Off On On On} \\ \text{Off On Off On Off} \end{bmatrix} \tag{2.2}$$

This means, that knowing the state of the light bulbs in the initial state, we can tell prover the assumption about the initial state. For example, if the initial state is the one marked above, then we can tell Prover

```
formulas(assumptions).
state([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0]).  % initial state
end_of_list.
```

and the end goal for Prover9 is always

```
formulas(goals).
state([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]).
           % goal state: all light bulbs are off
end_of_list.
```

- **The allowed steps and their effect**: we need to tell Prover9 what step is allowed in each state and what the next state is after a steps is applied in a given state. This is a direct translation of the game rules to Prover9's language.
  - *Step representation*: given that we converted the state matrix to a state vector, we should represent the light bulb on position (x_i, y_i) by the number A = x_i*5 + y_i + 1, so the step of switching the light bulb (x_i, y_i) can be identified by A too.
  - *Allowed steps*: any light bulb can be switched any time.
  - *The effect of a step*: step A is taken →the state of light bulb A and the light bulbs adjacent to it is toggled →the state of the following light bulbs is toggled.
    - (x_i, y_i) = A
    - (x_i - 1, y_i) = A - 5, if and only if x_i - 1 >= 0 ←→A - 5 >= 1
    - (x_i + 1, y_i) = A + 5, if and only if x_i + 1 < 5 ←→A + 5 <= 25
    - (x_i , y_i - 1) = A - 1, if and only if y_i - 1 >= 0 ←→A - 1 >= 1 and A mod 5 != 1 (Note: A mod 5 != 1 means that the cell is not in the first column of a row)

* $(x_i, y_i + 1) = A + 1$, if and only if $y_i + 1 < 5 \leftrightarrow A + 1 <= 25$ and A mod 5 != 0 (Note: A mod 5 !+ 0 means that the cell is not in the last column of a row)

Thus, we can describe a *neighbor(x, y)* predicate with value true if any only if light bulb X is the neighbor of light bulb Y to Prover9 as:

```
neighbor(X, Y) <->
       (-(X == 5 | X == 10 | X == 15 |
            X == 20 | X == 25) & X+1 == Y) |
       X == Y |
       X+5 == Y |
       (-(Y == 5 | Y == 10 | Y == 15 |
            Y == 20 | Y == 25) & Y+1 == X) |
       Y+5 == X.
```

We can tell prover the effect of toggling a light bulb with a given state as:

```
toggle(0) = 1.
toggle(1) = 0.
```

And using these two helper functions and predicates, we can define the function toggleVector([F:R], S, C), which, given

* [F:R] = a vector containing the current state of the light bulbs
* S = the identifier step to be taken (1 <= C <= 25, according to the above description)
* C = the index of F in the original vector describing the current state of the game (indexed by 1) (note that the function is implemented recursively, and at each recursive step, 1 element is cut from the beginning of the list. This is why, for example, after 10 recursive steps, the F will actually correspond to the 10th element of the 25-element list describing the current state of the game).

returns the vector representing the new state of the game after applying step C, in the following way:

```
toggleVector([], S, C) = [].
toggleVector([F:R], S, C) =
    if(neighbor(S, C),
        [toggle(F):toggleVector(R, S, C+1)],
        [F:toggleVector(R, S, C+1)]).
```

Note: what toggleVector does is that it toggles the state of those elements, which are the neighbors of the light bulb C to be switched. Finally, we can describe the possible steps of the game in Prover9 in 25 statements, as:

```
state([F:R]) -> state(toggleVector([F:R],  1, 1)) #answer (1).
state([F:R]) -> state(toggleVector([F:R],  2, 1)) #answer (2).
state([F:R]) -> state(toggleVector([F:R],  3, 1)) #answer (3).
...
state([F:R]) -> state(toggleVector([F:R], 25, 1)) #answer (25).
```

The final code can be found seen in B.1. Run it as

See the following command :

```
$ prover9 -f allout_solver_A.in > allout_solver_A.out
```

Disadvantage: Inefficiency

While this code seems simple enough at the first sight, if we take into account that all Prover9 can do based on it is to apply a backtracking directly, with 25 options at each step and no clues for detecting earlier that from a given state S the goal state cannot be reached in the desired number of steps, it is clear, that this solution is highly inefficient.

For example, even if we try to solve the following puzzle with it:

$$InitialGameState1 = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.3}$$

whose shortest solution consists only of 5 steps:

- step 1: cell (0, 0)
- step 5: cell (0, 4)
- step 13: cell (2, 2)
- step 24: cell (4, 3)
- step 25: cell (4, 4)

Mace4 will take around 8 seconds to find a solution, and uses 56 MBs of memory. This is not so surprising, if we think about the fact that it needs to try at least all the $25^4 > 2_{16}$ potential solutions of length 4 before finding the shortest solution of length 5.

Moreover, for solving

$$InitialGameState2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.4}$$

whose shortest solution consists of 6 steps:

- step 1: cell (0, 0)
- step 2: cell (0, 1)
- step 4: cell (0, 3)
- step 5: cell (0, 4)
- step 21: cell (4, 1)
- step 25: cell (4, 4)

Mace4 will take around 59.6 seconds to find a solution, and uses 315 MBs of memory.

In general, we can observe that this solution cannot be applied for any puzzle with 7 or more steps in the shortest solution: the amount of memory required will be too large for a general PC, so the program will crash. This is a major disadvantage of this program: it's not only inefficient, but it cannot even guarantee that if it didn't find a solution, then there is no solution (obviously, there are plenty of puzzles with the shortest solution having length > 8).

2.1.2.1.3 Approach B: a more efficient solution with Mace4

With the goal of optimizing the program for solving a puzzle, we should analyze the problem a little bit more instead of just translating the game rules to Prover9/Mace4 code.

At a first sight, we may come up with the following observations:

1. *Redundant steps*: it is redundant to switch a light bulb twice, because that would cause its state to be set back to the original one. →*If a solution for a puzzle exists, then there must be a solution in which Any light bulb should be switched either once or twice.*
2. *The final state of a light bulb is given by the parity of the total number of switches applied on it or on its adjacent light bulbs*: the state of a light bulb is toggled exactly when it or an adjacent light bulb is switched. Overall, if the light bulb is toggled an even number of times, then its final state is the same as its original state. If it is toggled an odd number of times, then its final state is different from its original state.
3. *The order of the steps is irrelevant*: based on observation 2, it follows naturally that for the final state of the light bulb only the number of the switches applied on it and on its neighbors matters, but not the order in which these switches were applied.

Let's see how these observations help us to find a solution in a shorter time: Observations 1 and Observation 3 →Instead Instead of searching for a solution represented by a sequence of steps, we may search for just a set of steps. Whereas searching for a sequence of length n may require evaluating up to

$$25 * 24 * ...(26 - n) = 25^{\underline{n}}$$

options to be evaluated, for set of steps out of 25 possible steps there are 2^{25} options overall (not considering the size of the set).

Based on this simplification, we can avoid using Prover9 in production mode to generate a sequence of steps leading to the winning state, and we can instead use Mace4 to generate a solution model Switch(X, Y):

$$Switch(X, Y) \iff \text{switching the light bulb on position (X, Y) is a step of the solution} \tag{2.5}$$

based on the initial game state:

$$On(X, Y) \iff \text{the light bulb on position (X, Y) is initially turned on} \tag{2.6}$$

(Note that in the actual code, instead of a binary predicate, a function with a numeric value, 0 or 1 was used, because this made some function compositions easier.)

Let's use the matrix OddToggles(X, Y) to mark that the number of toggles applied to a light bulb(= the number of switches applied to that light bulb and its neighbors) is odd.

$$OddToggles(X, Y) \iff \text{the number of switches applied to light bulb (X, Y) and its adjacent light bulbs is odd} \tag{2.7}$$

$$OddToggles(X, Y) \iff \text{the number of switches applied to light bulbs (X, Y), (X-1, Y), (X+1, Y), (X, Y-1), (X, Y+1) is odd} \tag{2.8}$$

We can define OddToggles by computing the Xor-Sum of Switch(X, Y), Switch(X-1, Y), Switch(X+1, Y), Switch(X, Y-1) and Switch(X, Y+1), because the Xor-Sum of some binary numbers gives the modulo 2 value of their sum.

$$OddToggles(X, Y) \iff \text{Xor(Xor(Xor(Switch(X+1, Y), Switch(X-1, Y)), Xor(Switch(X, Y-1), Switch(X, Y+1))), Switch(X, Y))} \tag{2.9}$$

Now, based on Observation 2, we can say that we want a light bulb to be toggled an odd number of times if initially it's turned on, and an even number of times if initially it's turned off, so that, in the end, all light bulbs will be turned off. Thus,

$$OddToggles(X, Y) \iff On(X, Y) \tag{2.10}$$

The above described relations are enough to find a solution to an initial game state, provided as an assumption to Mace4 using the On(X, Y) relation.

One point that was missed in the above train of though is that some light bulbs do not have an adjacent light bulb in all 4 directions, so (2.10) cannot be applied directly. To overcome this

issue, in the actual code, all the matrices were extended with 1-1 additional row and column in all sides, so that all cells of the original matrices had neighbors in all directions. It was fixed though, that the "light bulbs" in the extended rows/columns cannot be turned on or switched, so they didn't actually have an effect on the other light bulbs.

The final code can be found seen in B.2. Run it as

See the following command :

```
$ mace4 -m -1 -f allout_solver_B.in > allout_solver_B.out
```

to find **all solutions** for a given initial state.

2.1.2.1.4 Comparing approach A and B

I performed an experiment to compare the efficiency of approach A and B, based on the execution time.

Note that the comparison is not entirely accurate: while approach A finds just one solution, but the shortest one, approach B was set to find all solutions, but cannot tell, which one is the shortest. However, we'll later show that any solvable puzzle has exactly 4 solutions, so finding the shortest one based on the output of approach B is in fact a simple task.

For testing, I used the following matrices:

$$M1 = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, Steps1 = \begin{bmatrix} (0,0) & (0,4) \end{bmatrix} \tag{2.11}$$

$$M2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} Steps2 = \begin{bmatrix} (0,0) & (0,4) & (2,2) & (4,3) & (4,4) \end{bmatrix} \tag{2.12}$$

$$M3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} Steps3 = \begin{bmatrix} (0,0) & (0,1) & (0,3) & (0,4) & (4,0) & (4,4) \end{bmatrix} \tag{2.13}$$

$$M4 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \; Steps4 = \begin{bmatrix} (0,0) & (0,4) & (1,1) & (1,2) \\ (1,3) & (2,0) & (2,4) & (4,2) \end{bmatrix} \tag{2.14}$$

$$M5 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \; Steps5 = \begin{bmatrix} (0,3) & (0,4) & (1,0) & (1,1) & (1,3) \\ (1,4) & (2,1) & (2,2) & (2,4) & (3,1) \\ (3,2) & (3,3) & (4,0) & (4,2) & (4,3) \end{bmatrix} \tag{2.15}$$

Results

Negative values mark that the program failed to produce a solution with the memory limit of 1000MB.

See 2.4.

Conclusions

While for approach 2, the execution time remains constant with respect to the number of steps in the solution, and it takes a little less than 1s, approach B's execution time grows exponentially with the number of steps, takes almost 1 minutes for a 6-step solution and fails to provide a solution with the memory limit of 1000MB for a puzzle with more than 6 steps in its shortest solution.

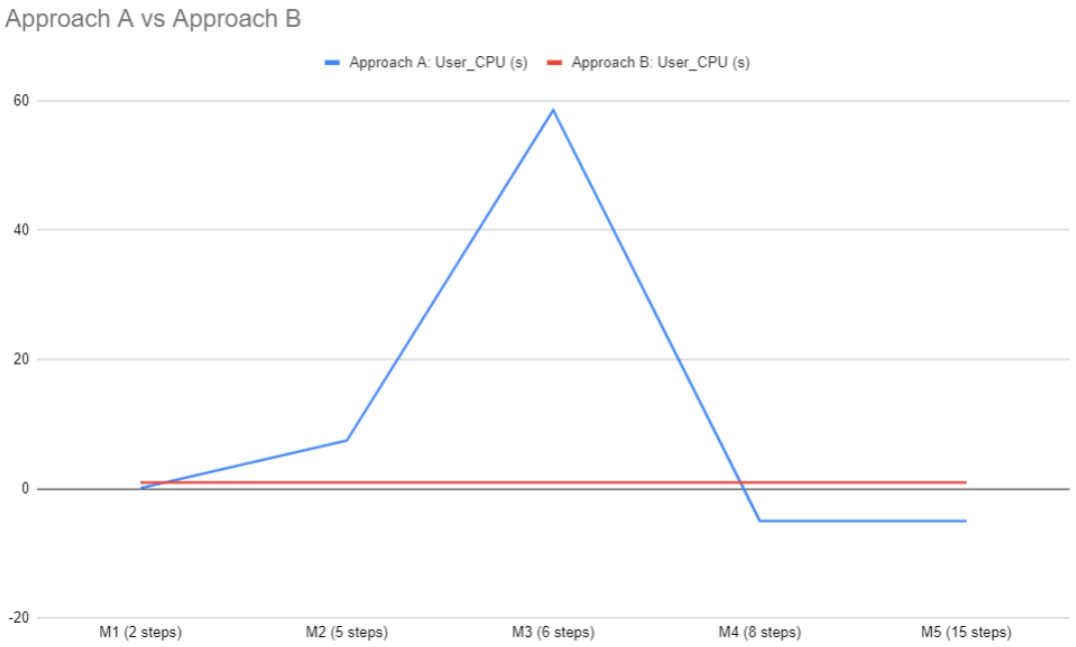


Figure 2.4: Comparing the User_CPU time of the two approaches

2.1.2.1.5 Generating puzzles

Goal: find multiple puzzles that have a solution.

Usage: in the desktop application based on this game, initially, 1000 solvable puzzles are generated, and then each time the user wins/asks for a new puzzle, one, previously unseen puzzle is selected randomly from the 1000 solvable puzzles.

2.1.2.1.6 Approach A: a naive generator based on the puzzle solving algorithm

Based on what was presented in the previous sections, an obvious idea for generating puzzles is to use a puzzle solver (the one from approach B), but with the initial game state being unspecified. To do so, what we need to do is only to remove the propositions about the On(X, Y) relation, and specify only that there must be at least one light bulb which is on in the initial state.

```
exists X exists Y On(X, Y)=1.
```

The final code can be found seen in B.3. Run it as

See the following command :

```
$ mace4 -m 100 -f allout_generator_A.in > allout_generator_A.out
```

to find 100 models.

Disadvantages

- *This program does unnecessary things*: note that our goal was just to generate a solvable puzzle, not that of solving it.
- *The same puzzle is generated multiple times*, because the models generated by mace4 include the solution to the puzzle also, and each solvable puzzle has multiple (4) solutions. This means, that based on the search strategy that mace4 applies, if we generated 100 models, we would get only 25 different solvable puzzles, and it would be an additional task to identify, which models represent the same puzzles.

2.1.2.1.7 Approach B: a seemingly better approach based on linear algebra and Mace4

Source of inspiration: "Turning Lights Out with Linear Algebra", by Marlow Anderson and Todd Feil

This new approach relies on relational algebra to model the game, and is based on an article entitled "Turning Lights Out with Linear Algebra", written by Marlow Anderson and Todd

Feil, published in Mathematics Magazine, Vol. 71, No. 4 (Oct., 1998), pp. 300-303. The article can be found on jstor or here.

The key learnings from this article are that:

1. **Equivalent criterion for the solvability of a game state**: if we represent the initial state of the game by the column vector B (as in the puzzle solver A), as

   - B[i] = 0 →the light bulb in row i/5 and column i%5 is off
   - B[i] = 1 →the light bulb in row i/5 and column i%5 is on

   in the format
   $$\vec{B} = \begin{pmatrix} On(0) & On(1) & On(2) & \dots & On(24) \end{pmatrix}^{\mathsf{T}} \tag{2.16}$$
   then
   $$\text{B is a solvable game state} \iff B \perp N1 \ \ \& \ \ B \perp N2 \tag{2.17}$$
   where
   $$\vec{N1} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & & & & & \end{pmatrix}^{\mathsf{T}} \tag{2.18}$$
   and
   $$\vec{N2} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & & & & & \end{pmatrix}^{\mathsf{T}} \tag{2.19}$$
   (Both N1 and N2 are column vectors, whose transpose is rendered on multiple rows.)

2. **The number of solutions of a game state**: Each solvable game state has exactly 4 solutions.

3. **The format of the solutions** If
   $$\vec{X} = \begin{pmatrix} Switch(0) & Switch(1) & Switch(2) & \dots & Switch(24) \end{pmatrix}^{\mathsf{T}} \tag{2.20}$$
   is a solution for game state B, then the 4 solutions are
   $$\begin{aligned} \vec{X1} &= \vec{X} \\ \vec{X2} &= \vec{X} + \vec{N1} \\ \vec{X3} &= \vec{X} + \vec{N2} \\ \vec{X4} &= \vec{X} + \vec{N1} + \vec{N2} \end{aligned} \tag{2.21}$$

The above theorems provide a great help towards finding a better solution for generating puzzles: we need to ask Mace4 to fill the vector On, representing the initial state of a solvable game, with 0's and 1's (at least 1 value of 1), such that On is perpendicular on N1 and N2. We can easily verify the perpendicularity by computing the dot products:

$$\vec{On} \perp \vec{N1} \iff \vec{On} \cdot \vec{N1} = 0 \tag{2.22}$$

$$\vec{On} \perp \vec{N2} \iff \vec{On} \cdot \vec{N2} = 0 \tag{2.23}$$

In Mace4, using the arithmetic mode with a domain size of 26, we can define N1 and N2 as

```
N1(0) =0.
N1( 1)=0. N1( 2)=1. N1( 3)=1. N1( 4)=1. N1( 5)=0.
N1( 6)=1. N1( 7)=0. N1( 8)=1. N1( 9)=0. N1(10)=1.
N1(11)=1. N1(12)=1. N1(13)=0. N1(14)=1. N1(15)=1.
N1(16)=1. N1(17)=0. N1(18)=1. N1(19)=0. N1(20)=1.
N1(21)=0. N1(22)=1. N1(23)=1. N1(24)=1. N1(25)=0.

N2(0)=0.
N2( 1)=1. N2( 2)=0. N2( 3)=1. N2( 4)=0. N2( 5)=1.
N2( 6)=1. N2( 7)=0. N2( 8)=1. N2( 9)=0. N2(10)=1.
N2(11)=0. N2(12)=0. N2(13)=0. N2(14)=0. N2(15)=0.
N2(16)=1. N2(17)=0. N2(18)=1. N2(19)=0. N2(20)=1.
N2(21)=1. N2(22)=0. N2(23)=1. N2(24)=0. N2(25)=1.
```

(Note that N1, N2 and On are indexed from 0, to allow an easier description of the following relations.)

We can implement a function for computing the dot products in a recursive way:

```
(X>0 & Y+1=X & Z=DotProduct1(Y) & O=On(X) & N=N1(X)) ->
    DotProduct1(X) = Xor(Z, And(O, N)).
DotProduct1(0)=0.

(X>0 & Y+1=X & Z=DotProduct2(Y) & O=On(X) & N=N2(X)) ->
    DotProduct2(X) = Xor(Z, And(O, N)).
DotProduct2(0)=0.
```

so the conditions for perpendicularity can be expressed as

```
DotProduct1(25)=0.
DotProduct2(25)=0.
```

Finally, Mace4 has to search for the unknown values of On, for which

```
On(0)=0.
exists X On(X)=1.
```

The final code can be found seen in B.4. Run it as

See the following command :

```
$ mace4 -m 1 -f allout_generator_B.in > allout_generator_B.out
```

to find 1 solvable puzzle.

Advantages

This approach seems much better than approach A, because it is based on an equivalent condition for a puzzle being solvable, and searches only for the values representing the states of the light bulbs in the initial state of the puzzle, and not for the solution of the puzzle, as approach A does.

Unexpected disadvantages

However, if you try to run this code, you'll see that it's extremely slow: finding one model takes seconds, but obviously, one model is not helpful at all, since Mace4 will always generate the same one if we ask for just one.

Conclusion

Even though the algorithm behind this approach is simpler, in practice, it's not usable.

2.1.2.1.8 Approach C: an improved version of approach B

Observations on approach B.

If one tries to run approach B with a smaller domain size and smaller vectors, then the program runs fairly quickly and produces unique results. Also, if one takes a look at the model generated by Mace4 in approach B, it can be noticed that because of using domain size 5, all the helper functions are unnecessarily large (the binary relations And and Xor have 25*25 = 625 elements). Defining all the unnecessary values, even if their value can be easily deduced and no backtracking is needed, takes time for Mace4.

This leads us to an idea for improving approach B by *reducing the domain size*: let's reorganize the vectors On(X), N1(X), N2(X), DotProduct(X) with 1 <= X <= 25 into 5x5 matrices On(X, Y), N1(X, Y), N2(X, Y) and DotProduct(X, Y) with 1 <= X, y <= 5. Thus, we can use a domain size of 6 instead of 26.

The relations can be redefined as:

```
N1(0, 0)=0. N1(0, 1)=1. N1(0, 2)=1. N1(0, 3)=1. N1(0, 4)=0.
N1(1, 0)=1. N1(1, 1)=0. N1(1, 2)=1. N1(1, 3)=0. N1(1, 4)=1.
N1(2, 0)=1. N1(2, 1)=1. N1(2, 2)=0. N1(2, 3)=1. N1(2, 4)=1.
N1(3, 0)=1. N1(3, 1)=0. N1(3, 2)=1. N1(3, 3)=0. N1(3, 4)=1.
N1(4, 0)=0. N1(4, 1)=1. N1(4, 2)=1. N1(4, 3)=1. N1(4, 4)=0.

N2(0, 0)=1. N2(0, 1)=0. N2(0, 2)=1. N2(0, 3)=0. N2(0, 4)=1.
N2(1, 0)=1. N2(1, 1)=0. N2(1, 2)=1. N2(1, 3)=0. N2(1, 4)=1.
N2(2, 0)=0. N2(2, 1)=0. N2(2, 2)=0. N2(2, 3)=0. N2(2, 4)=0.
N2(3, 0)=1. N2(3, 1)=0. N2(3, 2)=1. N2(3, 3)=0. N2(3, 4)=1.
N2(4, 0)=1. N2(4, 1)=0. N2(4, 2)=1. N2(4, 3)=0. N2(4, 4)=1.


(Y>=1 & YN+1=Y & D=DotProduct1(X, YN) & O=On(X, Y) & N=N1(X, Y)) ->
    DotProduct1(X, Y) = Xor(D, And(O, N)).
(X>=1 & XN+1=X & D=DotProduct1(XN, 4) & O=On(X, 0) & N=N1(X, 0)) ->
    DotProduct1(X, 0) = Xor(D, And(O, N)).
```

```
O=On(0, 0) & N=N1(0, 0) -> DotProduct1(0, 0)=And(O, N).

(Y>=1 & YN+1=Y & D=DotProduct2(X, YN) & O=On(X, Y) & N=N2(X, Y)) ->
    DotProduct2(X, Y) = Xor(D, And(O, N)).
(X>=1 & XN+1=X & D=DotProduct2(XN, 4) & O=On(X, 0) & N=N2(X, 0)) ->
    DotProduct2(X, 0) = Xor(D, And(O, N)).
O=On(0, 0) & N=N2(0, 0) -> DotProduct2(0, 0)=And(O, N).
```

where the goal is to get the values of On(X, Y)

```
On(X, Y)=1 | On(X, Y)=0.
exists X exists Y On(X, Y)=1.
```

such that

```
DotProduct1(4, 4)=0.
DotProduct2(4, 4)=0.
```

The final code can be found seen in B.5. Run it as

See the following command :

```
$ mace4 -m 1000 -f allout_generator_C.in > allout_generator_C.out
```

to find 1000 solvable puzzles.

Conclusion

With this simple improvement, the decrease in execution time is more than significant; generating one model takes just a fraction of a second with approach C.

2.1.2.1.9 Comparing approaches A, B and C

2.5 and 2.6 summarizes the differences in the performance of the Mace4 programs based on the three approaches, based on the User_CPU time (Note that 2.6 used a logarithmic scale.)

Note that for finding n solvable puzzles, from Mace4 with approach B and approach C, n models were requested, but with approach A, 4*n models were requested, for the reason that was explained above.

It's clear, that approach C outperforms all the other ones: it can generate 1000 solvable puzzles in 0.19 seconds, which makes it well-suited for being integrated into a desktop application implemented in python. The other approaches would increase the waiting time for getting a puzzle, and thus would make the game difficult to enjoy, unless all puzzles are pregenerated.

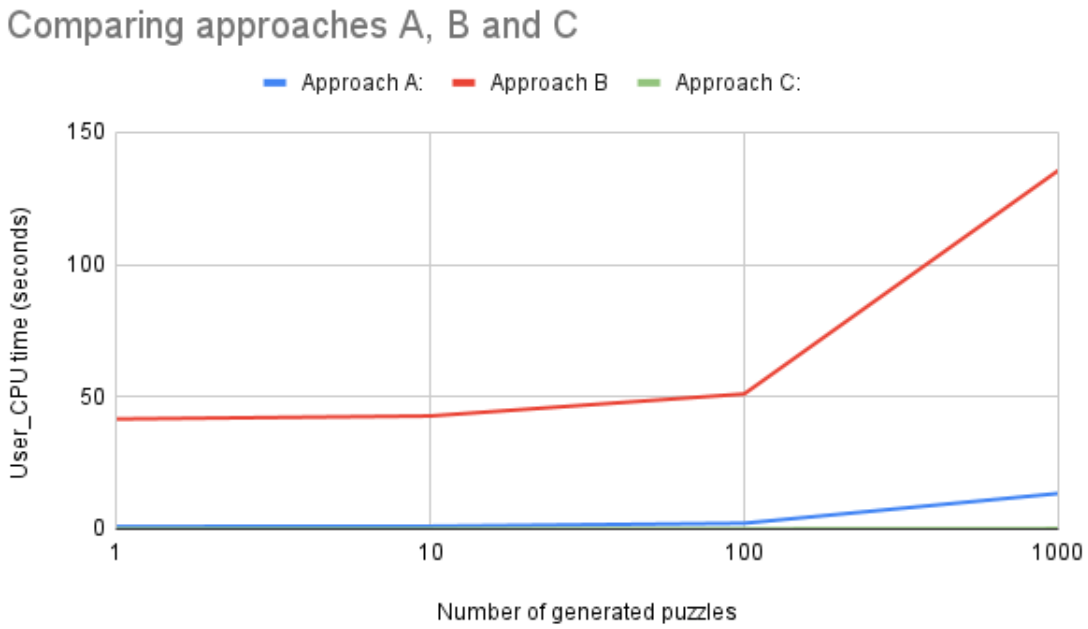


Figure 2.5: Comparing the User_CPU time of the three approaches

2.1.2.2 Interface Part

In order to demonstrate the flow of the game, we have created an interface for it in Python, which integrates the logic part implemented with Mace4.

For the implementation we have followed the Model-View-Controller pattern, to realize an interactive application, that reacts to each user action in real-time.

- *GameState.py*: represents the **model**.

  It contains the important attributes related to the state of the game, such as:

  - **lights_on**: 5x5 matrix, which represents the current state of the grid of lights. If the light is turned off in cell (x,y), then the matrix will contain a 0 at the corresponding position, otherwise if the light is on, then a 1 will be written in the lights_on matrix. It is updated after each move of the player.
  - **shortest_solution**: represents a list of moves/position's on which the lights should be toggled in order to reach a solution the fastest way possible (with the smallest number of moves).
  - **no_taken_steps**: represents the total number of steps taken by the player until the current state. It is incremented after each move of the player.

  The most important methods which can be used to update the state of the game are the following:

  - **switch_light**: toggles the light when a user clicks on a given cell.

```
1     def switch_light(self, x, y):
2         assert 0 <= x < GameState.no_rows and 0 <= y < GameState.
    no_cols
3         self.__no_taken_steps = self.__no_taken_steps + 1
```

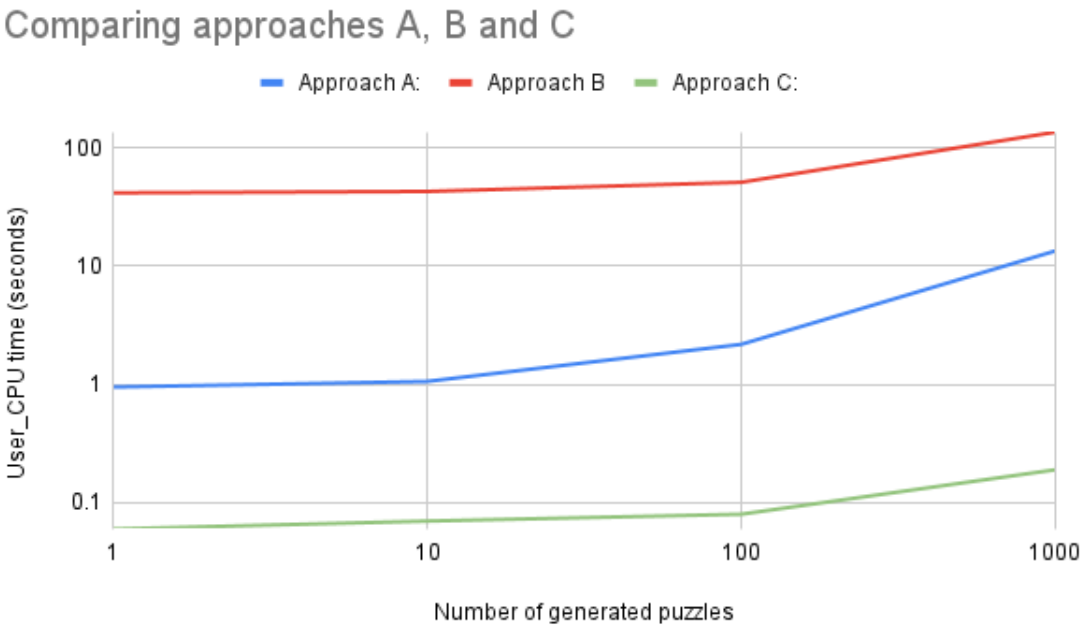


Figure 2.6: Comparing the User_CPU time of the three approaches, with a logarithmic scale

```
        self.__toggle_light(x, y)
        if x > 0:
            self.__toggle_light(x - 1, y)
        if x + 1 < GameState.no_rows:
            self.__toggle_light(x + 1, y)
        if y > 0:
            self.__toggle_light(x, y - 1)
        if y + 1 < GameState.no_cols:
            self.__toggle_light(x, y + 1)
        if self.__shortest_solution is not None:
            if self.__shortest_solution[x][y] == 1:
                self.__shortest_solution[x][y] = 0.
            else:
                self.__shortest_solution = None
```

– **set_solutions**: it takes a list of valid solutions and selects the shortest one to be saved.

```
    def set_solutions(self, solutions):
        shortest_solution_length = GameState.no_cols * GameState.
    no_rows + 1
        for solution in solutions:
            assert len(solution) == GameState.no_rows
            assert len(solution[0]) == GameState.no_cols
            solution_length = sum([toggle for sublist in solution
    for toggle in sublist])
            if solution_length < shortest_solution_length:
                shortest_solution_length = solution_length
                self.__shortest_solution = solution
                if self.__lights_on == self.__initial_lights_on:
                    self.__initial_shortest_solution = deepcopy(
    solution)
                    self.__initial_shortest_solution_no_steps =
    solution_length
```

- **get_hint_for_next_step**: returns a random valid move from the selected shortest solution. The order of the moves doesn't influence the outcome of the game.

```
    def get_hint_for_next_step(self):
        if self.__shortest_solution is not None:
            solution_vector = [toggle for sublist in self.
__shortest_solution for toggle in sublist]
            required_steps = [i for i, toggle in enumerate(
solution_vector) if toggle == 1]
            hinted_step_vector_index = random.choice(required_steps)
            hinted_step_row = hinted_step_vector_index / GameState.
no_cols
            hinted_step_col = hinted_step_vector_index % GameState.
no_cols
            hinted_step_matrix_index = (hinted_step_row,
hinted_step_col)
            return hinted_step_matrix_index
        else:
            return None
```

- **reset_to_initial_state**: reverts the lights_on matrix to the initial configuration at the start of the game, before the player made any changes to it.

```
    def reset_to_initial_state(self):
        self.__lights_on = deepcopy(self.__initial_lights_on)
        self.__shortest_solution = deepcopy(self.
__initial_shortest_solution)
        self.__no_taken_steps = 0
```

- **is_winning**: returns true if all the values in the lights_on matrix are 0, meaning that all the lights in the grid have been turned off.

```
    def is_winning(self):
        return all(not any(map(bool, lights_row)) for lights_row in
self.__lights_on)
```

The full code can be viewed at B.7.

- *GameService.py*: represents the **business logic of the game**. It is responsible for integrating the puzzle configurations and the solutions generated by mace4 and prover9 and for storing it in a GameState instance. It's main attributes include:

  - **game_state**: an instance of the GameState (model) which is updated whenever the board configuration changes (lights are toggled).
  - **game_models**: holds a list of generated solvable puzzle layouts (configurations).

  The main functionalities provided by the GameService class are the following:

  - **init_new_game_session**: initializes the game_state with a random model which was not played before from the initially generated game_models.

```
    def init_new_game_session(self):
        if len(self.__played_games) == GameService.NO_GAME_MODELS:
            self.__played_games = []
        game_model_index = random.randint(0, GameService.
NO_GAME_MODELS - 1)
        while game_model_index in self.__played_games:
            game_model_index = random.randint(0, GameService.
NO_GAME_MODELS - 1)
```

```
        self.__played_games.append(game_model_index)
        self.__game_state = GameState(deepcopy(self.__game_models[
    game_model_index]))
        self.__set_game_session_solutions()
```

– **generate_game_solutions**: runs mace4 in the background to find all solutions from the current state of the game. First it extends a Mace4 code containing the game rules with the propositions representing the current game state. Then in the command provided for mace4, the output (the solutions) is redirected to a separate file, which will be read, the solutions will be extracted and stored in the provided data structures, such as in the shortest_solution attribute of the game_state.

```
    def __generate_game_solutions(self):
        try:
            with open("GameSolutions/allout_solver_template.in") as
    template_file:
                solver_template = template_file.read()
                solver_mace4_code = solver_template.format(On=self.
    __generate_game_state_propositions_string())
                with open("GameSolutions/allout_solver.in", "w") as
    solver_file:
                    solver_file.write(solver_mace4_code)
                    solver_file.flush()
                    os.system("mace4 -m -1 -f GameSolutions/
    allout_solver.in | "
                              "interpformat portable > GameSolutions
    /allout_solutions.out")
                    with open("GameSolutions/allout_solutions.out")
    as solutions_file:
                        solutions_data = eval(solutions_file.read())
                        solutions = map(self.
    __transform_solution_data, solutions_data)
                        return solutions
        except IOError as e:
            print "I/O error({0}): {1}".format(e.errno, e.strerror)
        except:  # handle other exceptions such as attribute errors
            print "Unexpected error:", sys.exc_info()[0]
        sys.exit(1)
```

– **generate_random_games**: generates a given number of random, but solvable game layouts using Mace4, the results being written in the *allout_math_efficient_generated.out* to be reused later.

```
    @staticmethod
    def __generate_random_games(no_game_models):
        os.system("mace4 -m {nr_game_models} -f GameModels/
    allout_math_efficient_generator.in  | "
                  "interpformat portable > GameModels/
    allout_math_efficient_generated.out".format(
            nr_game_models=no_game_models))
```

– **get_generated_game_models**: reads and interprets the content of the *allout_math_efficient_generated.out* file, mapping the list of models from the file to the data structure used to store these random models internally in the GameService class.

```
    def __get_generated_game_models(self):
        f = open("GameModels/allout_math_efficient_generated.out")
        game_models_data = eval(f.read())
```

```
        game_models = map(self.__transform_game_model_data,
    game_models_data)
        return game_models
```

- **get_lights_state, get_hint, won_game_session get_no_steps_in_game_session, switch_light, reset_game_session**: the execution of these commands are delegated to the game_state attribute, as these functionalities are already implemented in the GameState class.
- **get_hint**: whenever this method is invoked, it runs the mace4 code in a new process, to find the solutions of the current game state, end waits for the Mace4 program to finish execution, which results in a small delay (1s) in the GUI until the animation for the hint would appear.
  One solution to eliminate this delay was to compute the solution for the current board configuration after every new move of the player, without waiting for the user to explicitly ask for a hint. This way, the hint would be prepared by the time the user asked for it. However, this solution would work only if the Mace4 program for finding the solutions would run in a separate thread, without the main thread having to wait for its execution, thus allowing the GUI to be responsive even while the Mace4 program is running. However, we decided against this implementation because of the escalating complexity of synchronizing multiple threads, which was out of scope in this case.

```
    def get_hint(self):
        if self.__game_state.has_shortest_solution():
            return self.__game_state.get_hint_for_next_step()
        else:
            self.__set_game_session_solutions()
            return self.__game_state.get_hint_for_next_step()
```

The full code can be viewed at B.8.

- *GameDisplay.py*: represents the **view** and **controller** layer of the game.
  It is responsible for displaying the game board and handle each user input, such as button clicks, by delegating the request to the GameService class. It uses the functionalities provided by the **tkinter** module to the create a window and adds widgets, such as buttons and images to it.

  As main attributes it contains:

  - **game_service**: represents an instance of the GameService class, which supplies the needed functionalities to react to the different events in the ui.
  - **window**: represents the main window of the application on which the board is displayed.
  - **tiles**: represents the 5x5 matrix of lights. Each light on the board is represented by a button with the image of a yellow (=On) or grey (=Off) led.

  The methods which handle the ui events are the following:

  - **refresh_board**: rerenders the grid of lights, changing the image of the corresponding led to the one which represents its current state.

```
    def __refresh_board(self):
        lights_on = self.game_service.get_lights_state()
        for i in range(self.no_tiles):
```

```
            for j in range(self.no_tiles):
                self.tiles[i][j].config(
                    image=self.tile_state[lights_on[i][j]])
```

– **on_flip**: when the user clicks on a button, it is called to configure the corresponding image for the button on position (x,y), depending on the new state of the given cell(light bulb).

```
    def __handle_finish_game(self, game_solved_by_player):
        self.__display_results_popup(
            game_solved_by_player=game_solved_by_player)
        self.__set_button_editability(enable_hint_button=False,
                                      enable_solve_button=False)

    def __flip_tile(self, x, y, enable_hint_button,
                    enable_solve_button,
                    game_solved_by_player=True):
        # stop any running hint animations
        self.__stop_running_hint_animations()
        self.__set_button_editability(enable_hint_button,
                                      enable_solve_button)
        self.game_service.switch_light(x, y)
        self.__refresh_board()
        if self.game_service.won_game_session():
            # delay showing the results to display the board after
            # the last step
            self.window.after(500, lambda: self.__handle_finish_game
(
                game_solved_by_player))

    def __on_flip(self, x, y):
        # tiles cannot be flipped while the solution simulation is
        # running
        if self.solution_animation_id is not None:
            return
        self.__flip_tile(x, y, enable_hint_button=True,
                         enable_solve_button=True)
```

– **on_hint**: it triggers an animation of a rotating green led, indicating a hint for the player (the next cell that shoul be clicked). Once a player clicked on the "Hint" button, it will stay disabled, until they click on a tile or they reset/start a new game.

```
    def __animate_hint(self, x, y, index=0):
        self.tiles[x][y].config(image=self.frames[index])
        index = (index + 1) % self.no_frames
        self.hint_animation_id = \
            self.window.after(100, lambda: self.__animate_hint(
                x, y, index))

    def __on_hint(self):
        # show hint only if the game has not yet been won
        if not self.game_service.won_game_session():
            # disable the button until the hint animation is stopped
:
            # only 1 hint should be played at a time
            self.__set_button_editability(enable_hint_button=False)
            (x, y) = self.game_service.get_hint()
            self.__animate_hint(x, y)
```

– **on_solve** and **animate_solution**: animate the steps required to reach a solution starting from the current configuration. Once the "Solve" button was clicked, it will

be disabled along with the "Hint" button, and the player cannot click on the tiles either. This is in order to not disrupt the animation. However, the solution tracing animation stops when the player resets/starts a new game.

```
    def __animate_solution(self):
        # run simulation only if the game has not yet been won
        if self.game_service.won_game_session():
            self.__stop_running_solution_animations()
            return
        (x, y) = self.game_service.get_hint()
        self.__animate_hint(x, y)
        self.solution_animation_id = \
            self.window.after(1500, lambda: self.__next_hint(x, y))

    def __on_solve(self):
        # stop previous hint animations and disable Hint button
        self.__stop_running_hint_animations()
        # clear the animation from the board
        self.__refresh_board()
        # disable the Hint and Solve buttons while the current
    animation is running
        self.__set_button_editability(enable_hint_button=False,
                                      enable_solve_button=False)
        self.__animate_solution()
```

– **display_results_popup**: opens a popup dialog in which the results of the game are displayed. If the player was the one to finish the game, then it shows the number of moves made by the player and a congratulatory message. If the solution was simulated using the "Solve" button, then it only displays the number of moves in the shortest solution.

```
    def __display_results_popup(self, game_solved_by_player):
        popup_dialog = Toplevel(self.window)
        popup_dialog.geometry(
            str(POPUP_WIDTH) + "x" + str(POPUP_HEIGHT))
        popup_dialog.title("Game results")
        popup_dialog.resizable(False, False)
        popup_dialog.columnconfigure(0, weight=1)
        popup_dialog.rowconfigure(0, weight=3)
        popup_dialog.rowconfigure(1, weight=1)
        popup_dialog.rowconfigure(2, weight=2)

        # display the congratulatory message only if the puzzle was
        # solved by the player (and not by clicking on the Solve
    button)
        if game_solved_by_player:
            Label(popup_dialog,
                  image=self.image_congratulations).grid(
                row=0, column=0)
            Label(popup_dialog, text="You solved the game in " + str
    (
                self.game_service.get_no_steps_in_game_session()) +
    " steps.",
                  font=('Mistral 17 bold')).grid(row=1, column=0)

        Label(popup_dialog,
              text="The shortest solution consisted of " + str(
                  self.game_service.
    length_of_shortest_solution_from_initial_state()) + " steps.",
              font=('Mistral 15 bold')).grid(
            row=2 if game_solved_by_player else 0, column=0)
```

– **on_new_game**: requests a new game configuration from the GameService class and displays the new game layout on the board.

```
    def __on_new_game(self):
        # stop all running animations
        self.__stop_running_hint_animations()
        # re-enable the buttons
        self.__set_button_editability()
        self.__stop_running_solution_animations()
        self.game_service.init_new_game_session()
        self.__refresh_board()
```

– **on_reset**: resets the lights to the initial configuration, before any move of the player.

```
    def __on_reset(self):
        # stop all running animations
        self.__stop_running_hint_animations()
        self.__stop_running_solution_animations()
        # re-enable the buttons
        self.__set_button_editability()
        self.game_service.reset_game_session()
        self.__refresh_board()
```

The full code can be viewed at B.9.

2.1.3 Variants of the Lights Out game

There are many variations of the classic game, for example changing the size of the board, the pattern of the moves, or the number of possible states for each cell.

- Instead of the classic 5x5 grid layout, the game can be played on a 9x9 grid.
- The type of the move can be modified as well, the buttons could give a pattern other than a vertical cross, for example the diagonal cross, torus or the Knight's move from chess. However, if a button changes lights that are not adjacent but further away then light chasing becomes more difficult.
- The number of states assigned to a cell could be extended as well, the Lights Out 2000 came with 3 states assigned to each led, which corresponded to 3 different colors. On each button press, the led transitioned from one state (colour) to the next.

The variant of the game, based on extending the number of possible states, can be implemented in mace4 in a similar way as in case of the classic 2 states.
The effect of changing the state of a light affects not only the clicked light button but also its direct neighbours, using the same cross pattern for the moves. The total number of switches of a given button together with the number of switches of its neighbours account to the final state of the button, which should be 0, corresponding to the final (off) state.
With respect to the implementation, instead of using the xor operation, we use the addition, for operands outside the range of modulo 2, in order to compute the effect of these switches on each button's state.

```
% Lights out problem generalized for the case when one cell can have more
% than 2 states.
% b(x,y) represent the initial state of the cell.
% c(x,y) represent the number of times a cell's state is changed.

% ds = represents the size of the extended grid
ds = domain_size + -1.
% the number of states in which a cell can be
noStates = 3.

% after applying a number of moves(switches) on the current and neighbour cells,
% the final state of the current cell should be the state 0.
% that means, if the cell b(x,y) should be changed either directly
% or through its neighbours if it is not in the correct state.
(x > 0 & x < ds & y > 0 & y < ds & xp = x + -1 & yp = y + -1 & xs = x + 1
& ys = y + 1) ->
(b(x,y) + c(x,y) + c(xp,y) + c(x,yp) + c(xs,y) + c(x,ys)) mod noStates = 0.
```

The complete code could be found on B.6.
Run it with:

```
$ mace4 -m -1 -f allout_many_states_solver.in > allout_many_states_solver.out
```

2.1.4 Bibliography

- "Turning Lights Out with Linear Algebra", by Marlow Anderson and Todd Feil, Mathematics Magazine, Vol. 71, No. 4 (Oct., 1998), pp. 300-303
- https://mathworld.wolfram.com/LightsOutPuzzle.html

Chapter 3

A3: Planning

Appendix A

Code for A1: Search

```
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


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.
```

```
224
225     The matrix may be mirrored horizontally and/or vertically,
226     such that start_corner_pos ends up being mapped to position (0,
227     0) of the
228     final matrix,
229     and goal_corner_pos ends up being mapped to position (n-1,
230     m-1) of the
231     final matrix,
232     where n and m are the number of rows and columns in the rectangle.
233     """
234     rows_step = 1
235     cols_step = 1
236     if goal_corner_pos[0] < start_corner_pos[0]:
237         rows_step = -1
238     if goal_corner_pos[1] < start_corner_pos[1]:
239         cols_step = -1
240     start_row = start_corner_pos[0]
241     start_col = start_corner_pos[1]
242
243     res = [[int(food_grid[start_row + i * rows_step][
244                     start_col + j * cols_step]) for j in
245             range(abs(goal_corner_pos[1] - start_col) + 1)] for i in
246            range(abs(goal_corner_pos[0] - start_row) + 1)]
247     return res
248
249
250 def hungerGamesManhattanAndMaxFoodOnShortestPathHeuristic(state,
251                                                           problem):
252     """
253     A heuristic for the HungerGamesSearchProblem, based on the
254     following idea:
255     - any path from the current state to the goal takes at least
256     manhattan
257     distance steps
258     - if there is a path from the current position to the maze exit
259     position,
260     such that the manhattan distance <= current energy level +
261     energy level
262     gained from food dots on the path,
263     then it's possible, that a path satisfying all problem
264     constraints with
265     cost manhattan distance exists
266     - if there is no such path, then PacMan must step at least once
267     in the
268     "wrong direction".
269     By wrong direction we mean that for obtaining a path with
270     manhattan
271     distance cost,
272     if the maze exit is to the South from PacMan's position,
273     then any step to
274     the North is wrong and vice-versa.
275     Similarly, if the exit is to the East from PacMan's position,
276     then any
277     step to the West is wrong, and vice-versa.
278     Moreover, if PacMan takes one step to any wrong direction,
279     than that step
280     must be "recovered" later,
281     i.e. annulled with a backwards step.
282     Thus, we can guarantee that if no path with manhattan distance
283     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,
```

```
344     # additionally to the cost of manhattan distance.
345     if energy_level + problem.foodEnergyLevel * \
346             max_food_on_shortest_path < shortest_path_length:
347         # Not enough food on any minimum cost path
348         return shortest_path_length + 2
349     else:
350         # Possibly enough food on the shortest path
351         return shortest_path_length
352
353
354 def buildMaxEnergyLevelGrid(init_energy_level, food_grid,
355                             food_energy_level):
356     """
357     Given
358     - the initial energy of PacMan, assumed to be located in
359     position (0,
360     0) of the food_grid,
361     - the energy level given by a food dot (food_energy_level)
362     - the food_grid with the maze exit situated in the top-right
363     corner of
364     the grid,
365     and the current position of PacMan being (0, 0).
366     Computes the maximum energy level which PacMan may have when
367     leaving
368     location (x, y) of the grid, assuming that
369     PacMan reached this location via a path from (0, 0) with x+y
370     steps (i.e.
371     a minimum-cost path with steps only into
372     the correct directions).
373     Returns a matrix with the values for all (x, y) locations (of
374     the same
375     size as food_grid).
376
377     If a location (x0, y0) is not reachable according to the rules of
378     HungerGames, with only steps into the correct
379     directions, then -1 is placed on the given location.
380
381     Note: the "leaving energy" is computed, not the "reaching energy",
382     meaning that the energy given by the potential
383     doos dot on psotion (x, y) is added to the value computed for
384     location (
385     x, y).
386
387     For the computations, a dynamic programming algorithm is used,
388     based on
389     the formula
390     max energy on (x, y) = max(max energy on (x-1, y), max energy
391     on (x, y-1)) +
392                             + the energy given by the food dots on
393                             location (
394                             x, y)
395                             - 1
396     Where -1 is due to the cost of stepping from (x-1, y) or from (
397     x, y-1) to
398     (x, y).
399     """
400     from copy import deepcopy
401     max_energy_level_grid = deepcopy(food_grid)
402     max_energy_level_grid[0][0] = init_energy_level
403
```

```
    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


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(
```

```
942             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)
    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
        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

```

"""==================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
```

```
34
35              """==================END OF MY OWN CODE================"""
36
37          self._foodEaten = None
38          self._foodAdded = None
39          self._capsuleEaten = None
40          self._agentMoved = None
41          self._lose = False
42          self._win = False
43          self.scoreChange = 0
44
45      def deepCopy(self):
46          state = GameStateData(self)
47          state.food = self.food.deepCopy()
48          state.layout = self.layout.deepCopy()
49          """==================START OF MY OWN CODE================="""
50
51          state.pacmanEnergyLevel = self.pacmanEnergyLevel
52          state.initialEnergyLevel = self.initialEnergyLevel
53          state.foodEnergyLevel = self.foodEnergyLevel
54
55          """==================END OF MY OWN CODE==================="""
56          state._agentMoved = self._agentMoved
57          state._foodEaten = self._foodEaten
58          state._foodAdded = self._foodAdded
59          state._capsuleEaten = self._capsuleEaten
60          return state
61
62      def copyAgentStates(self, agentStates):
63          copiedStates = []
64          for agentState in agentStates:
65              copiedStates.append(agentState.copy())
66          return copiedStates
67
68      def __eq__(self, other):
69          """
70          Allows two states to be compared.
71          """
72          if other == None: return False
73          # TODO Check for type of other
74          if not self.agentStates == other.agentStates: return False
75          if not self.food == other.food: return False
76          if not self.capsules == other.capsules: return False
77          if not self.score == other.score: return False
78          """==================START OF MY OWN CODE================="""
79
80          if not self.initialEnergyLevel == other.initialnergyLevel:
81              return False
82          if not self.pacmanEnergyLevel == other.pacmanEnergyLevel:
83              return False
84          if not self.foodEnergyLevel == other.foodEnergyLevel:
85              return False
86
87          """==================END OF MY OWN CODE==================="""
88          return True
89
90
91      """==================START OF CODE MODIFIED BY ME============="""
92      def initialize(self, layout, numGhostAgents, pacmanEnergyLevel,
93                     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

Appendix B

Code for A2: Logics

```
% Solution for the Light Out problem. Inefficient.
set(production).
set(prolog_style_variables).

formulas(usable).

state([F:R]) -> state(toggleVector([F:R],  1, 1)) #answer (1).
state([F:R]) -> state(toggleVector([F:R],  2, 1)) #answer (2).
state([F:R]) -> state(toggleVector([F:R],  3, 1)) #answer (3).
state([F:R]) -> state(toggleVector([F:R],  4, 1)) #answer (4).
state([F:R]) -> state(toggleVector([F:R],  5, 1)) #answer (5).
state([F:R]) -> state(toggleVector([F:R],  6, 1)) #answer (6).
state([F:R]) -> state(toggleVector([F:R],  7, 1)) #answer (7).
state([F:R]) -> state(toggleVector([F:R],  8, 1)) #answer (8).
state([F:R]) -> state(toggleVector([F:R],  9, 1)) #answer (9).
state([F:R]) -> state(toggleVector([F:R], 10, 1)) #answer (10).
state([F:R]) -> state(toggleVector([F:R], 11, 1)) #answer (11).
state([F:R]) -> state(toggleVector([F:R], 12, 1)) #answer (12).
state([F:R]) -> state(toggleVector([F:R], 13, 1)) #answer (13).
state([F:R]) -> state(toggleVector([F:R], 14, 1)) #answer (14).
state([F:R]) -> state(toggleVector([F:R], 15, 1)) #answer (15).
state([F:R]) -> state(toggleVector([F:R], 16, 1)) #answer (16).
state([F:R]) -> state(toggleVector([F:R], 17, 1)) #answer (17).
state([F:R]) -> state(toggleVector([F:R], 18, 1)) #answer (18).
state([F:R]) -> state(toggleVector([F:R], 19, 1)) #answer (19).
state([F:R]) -> state(toggleVector([F:R], 20, 1)) #answer (20).
state([F:R]) -> state(toggleVector([F:R], 21, 1)) #answer (21).
state([F:R]) -> state(toggleVector([F:R], 22, 1)) #answer (22).
state([F:R]) -> state(toggleVector([F:R], 23, 1)) #answer (23).
state([F:R]) -> state(toggleVector([F:R], 24, 1)) #answer (24).
state([F:R]) -> state(toggleVector([F:R], 25, 1)) #answer (25).

end_of_list.

formulas(assumptions).
state([1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1,
   0, 0]).  % initial state
end_of_list.

formulas(goals).
state([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0]).  % goal state: all light bulbs are off
end_of_list.
```

```
42
43 formulas(demodulators).
44
45 neighbor(X, Y) <-> (-(X == 5 | X == 10 | X == 15 | X == 20 | X == 25) & X+1
       == Y) |
46                     X == Y |
47                     X+5 == Y |
48                     (-(Y == 5 | Y == 10 | Y == 15 | Y == 20 | Y == 25) & Y+1
       == X) |
49                     Y+5 == X.
50
51 toggle(0) = 1.
52 toggle(1) = 0.
53
54 toggleVector([], S, C) = [].
55 toggleVector([F:R], S, C) = if(neighbor(S, C),[toggle(F):toggleVector(R, S,
       C+1)],[F:toggleVector(R, S, C+1)]).
56
57
58 end_of_list.
```

Listing B.1: Puzzle Solver A - Inefficient

```
1 %set(binary_resolution).
2 %set(print_gen).
3
4 set(prolog_style_variables).
5
6
7 set(arithmetic).
8 assign(domain_size, 7).
9
10 formulas(usable).
11
12 all X all Y (Switch(X, Y)=0 | Switch(X, Y)=1).
13
14 all X (Switch(X, 0)=0 & Switch(X, 6)=0).
15 all Y (Switch(0, Y)=0 & Switch(6, Y)=0).
16
17 Xor(X, Y)=1 <-> (X=1 & Y=0) | (X=0 & Y=1).
18 Xor(X, Y)!=1 <-> Xor(X, Y)=0.
19
20 ((X >= 1 & X <= 5 & Y >= 1 & Y <= 5 & Z=X+1 & X=U+1 & V=Y+1 & Y=W+1) -> (
       OddToggles(X, Y) <-> Xor(Xor(Xor(Switch(X, V), Switch(X, W)), Xor(Switch(
       U, Y), Switch(Z, Y))), Switch(X, Y))=1)).
21
22 On(X, Y)=1 <-> OddToggles(X, Y).
23 On(X, Y)=0 <-> -OddToggles(X, Y).
24
25 end_of_list.
26
27 formulas(assumptions).
28 On(0, 0) = 0. On(0, 1) = 0. On(0, 2) = 0. On(0, 3) = 0. On(0, 4) = 0. On(0,
       5) = 0. On(0, 6) = 0.
29 On(1, 0) = 0. On(1, 1) = 1. On(1, 2) = 0. On(1, 3) = 0. On(1, 4) = 0. On(1,
       5) = 1. On(1, 6) = 0.
30 On(2, 0) = 0. On(2, 1) = 0. On(2, 2) = 0. On(2, 3) = 0. On(2, 4) = 0. On(2,
       5) = 0. On(2, 6) = 0.
31 On(3, 0) = 0. On(3, 1) = 0. On(3, 2) = 0. On(3, 3) = 0. On(3, 4) = 0. On(3,
       5) = 0. On(3, 6) = 0.
32 On(4, 0) = 0. On(4, 1) = 0. On(4, 2) = 1. On(4, 3) = 0. On(4, 4) = 1. On(4,
```

```
   5) = 0. On(4, 6) = 0.
On(5, 0) = 0. On(5, 1) = 0. On(5, 2) = 0. On(5, 3) = 1. On(5, 4) = 1. On(5,
   5) = 1. On(5, 6) = 0.
On(6, 0) = 0. On(6, 1) = 0. On(6, 2) = 0. On(6, 3) = 0. On(6, 4) = 0. On(6,
   5) = 0. On(6, 6) = 0.
end_of_list.

formulas(goals).
end_of_list.
```

Listing B.2: Puzzle Solver B - Efficient

```
%set(binary_resolution).
%set(print_gen).

set(prolog_style_variables).
set(arithmetic).
assign(domain_size, 7).

formulas(usable).

all X all Y (Switch(X, Y)=0 | Switch(X, Y)=1).
all X all Y (On(X, Y)=0 | On(X, Y)=1).

exists X exists Y On(X, Y)=1.

all X (Switch(X, 0)=0 & Switch(X, 6)=0).
all Y (Switch(0, Y)=0 & Switch(6, Y)=0).
all X (On(X, 0)=0 & On(X, 6)=0).
all Y (On(0, Y)=0 & On(6, Y)=0).

Xor(X, Y)=1 <-> (X=1 & Y=0) | (X=0 & Y=1).
Xor(X, Y)!=1 <-> Xor(X, Y)=0.

((X >= 1 & X <= 5 & Y >= 1 & Y <= 5 & Z=X+1 & X=U+1 & V=Y+1 & Y=W+1) ->
    (OddToggles(X, Y) <-> Xor(Xor(Xor(Switch(X, V), Switch(X, W)), Xor(
   Switch(U, Y), Switch(Z, Y))), Switch(X, Y))=1)).

On(X, Y)=1 <-> OddToggles(X, Y).
On(X, Y)=0 <-> -OddToggles(X, Y).

end_of_list.

formulas(assumptions).
end_of_list.

formulas(goals).
end_of_list.
```

Listing B.3: Puzzle Generator A - Inefficient

```
%set(binary_resolution).
%set(print_gen).

set(prolog_style_variables).
set(arithmetic).
assign(max_megs, 1000).
assign(domain_size, 26).

formulas(assumptions).

```

```
Xor(X, Y)=1 <-> (X=0 & Y=1) | (X=1 & Y=0).
Xor(X, Y)!=1 -> Xor(X, Y)=0.

And(X, Y)=1 <-> (X=1 & Y=1).
And(X, Y)!=1 -> And(X, Y)=0.

On(X)=1 | On(X)=0.

(X>0 & Y+1=X & Z=DotProduct1(Y) & O=On(X) & N=N1(X)) -> DotProduct1(X) = Xor
   (Z, And(O, N)).
DotProduct1(0)=0.

(X>0 & Y+1=X & Z=DotProduct2(Y) & O=On(X) & N=N2(X)) -> DotProduct2(X) = Xor
   (Z, And(O, N)).
DotProduct2(0)=0.

DotProduct1(25)=0.
DotProduct2(25)=0.

On(0)=0.
exists X On(X)=1.

N1(0)=0.
N1(X) = 0 | N1(X)=1.
N2(X) = 0 | N2(X)=1.
N1( 1)=0. N1( 2)=1. N1( 3)=1. N1( 4)=1. N1( 5)=0.
N1( 6)=1. N1( 7)=0. N1( 8)=1. N1( 9)=0. N1(10)=1.
N1(11)=1. N1(12)=1. N1(13)=0. N1(14)=1. N1(15)=1.
N1(16)=1. N1(17)=0. N1(18)=1. N1(19)=0. N1(20)=1.
N1(21)=0. N1(22)=1. N1(23)=1. N1(24)=1. N1(25)=0.

N2(0)=0.
N2( 1)=1. N2( 2)=0. N2( 3)=1. N2( 4)=0. N2( 5)=1.
N2( 6)=1. N2( 7)=0. N2( 8)=1. N2( 9)=0. N2(10)=1.
N2(11)=0. N2(12)=0. N2(13)=0. N2(14)=0. N2(15)=0.
N2(16)=1. N2(17)=0. N2(18)=1. N2(19)=0. N2(20)=1.
N2(21)=1. N2(22)=0. N2(23)=1. N2(24)=0. N2(25)=1.

end_of_list.
```

Listing B.4: Puzzle Generator B - Based on Linear Algebra, Inefficient

```
%set(binary_resolution).
%set(print_gen).

set(prolog_style_variables).
set(arithmetic).
assign(domain_size, 5).

formulas(assumptions).

Xor(X, Y)=1 <-> (X=0 & Y=1) | (X=1 & Y=0).
Xor(X, Y)!=1 -> Xor(X, Y)=0.

And(X, Y)=1 <-> (X=1 & Y=1).
And(X, Y)!=1 -> And(X, Y)=0.

On(X, Y)=1 | On(X, Y)=0.

(Y>=1 & YN+1=Y & D=DotProduct1(X, YN) & O=On(X, Y) & N=N1(X, Y)) ->
   DotProduct1(X, Y) = Xor(D, And(O, N)).
```

```
(X>=1 & XN+1=X & D=DotProduct1(XN, 4) & O=On(X, 0) & N=N1(X, 0)) ->
   DotProduct1(X, 0) = Xor(D, And(O, N)).
O=On(0, 0) & N=N1(0, 0) -> DotProduct1(0, 0)=And(O, N).

(Y>=1 & YN+1=Y & D=DotProduct2(X, YN) & O=On(X, Y) & N=N2(X, Y)) ->
   DotProduct2(X, Y) = Xor(D, And(O, N)).
(X>=1 & XN+1=X & D=DotProduct2(XN, 4) & O=On(X, 0) & N=N2(X, 0)) ->
   DotProduct2(X, 0) = Xor(D, And(O, N)).
O=On(0, 0) & N=N2(0, 0) -> DotProduct2(0, 0)=And(O, N).

DotProduct1(4, 4)=0.
DotProduct2(4, 4)=0.

exists X exists Y On(X, Y)=1.

N1(0, 0)=0. N1(0, 1)=1. N1(0, 2)=1. N1(0, 3)=1. N1(0, 4)=0.
N1(1, 0)=1. N1(1, 1)=0. N1(1, 2)=1. N1(1, 3)=0. N1(1, 4)=1.
N1(2, 0)=1. N1(2, 1)=1. N1(2, 2)=0. N1(2, 3)=1. N1(2, 4)=1.
N1(3, 0)=1. N1(3, 1)=0. N1(3, 2)=1. N1(3, 3)=0. N1(3, 4)=1.
N1(4, 0)=0. N1(4, 1)=1. N1(4, 2)=1. N1(4, 3)=1. N1(4, 4)=0.

N2(0, 0)=1. N2(0, 1)=0. N2(0, 2)=1. N2(0, 3)=0. N2(0, 4)=1.
N2(1, 0)=1. N2(1, 1)=0. N2(1, 2)=1. N2(1, 3)=0. N2(1, 4)=1.
N2(2, 0)=0. N2(2, 1)=0. N2(2, 2)=0. N2(2, 3)=0. N2(2, 4)=0.
N2(3, 0)=1. N2(3, 1)=0. N2(3, 2)=1. N2(3, 3)=0. N2(3, 4)=1.
N2(4, 0)=1. N2(4, 1)=0. N2(4, 2)=1. N2(4, 3)=0. N2(4, 4)=1.

end_of_list.
```

Listing B.5: Puzzle Generator C - Based on Linear Algebra, Efficient

```
set(arithmetic).
assign(domain_size,7).
assign(max_models,-1).

formulas(usable).

% ds = represents the size of the extended grid
ds = domain_size + -1.

% the number of states in which a cell can be
noStates = 3.

end_of_list.

% Lights out problem generalized for the case when one cell can have more
   than 2 states.
% b(x,y) represent the initial state of the cell.
% c(x,y) represent the number of times a cell's state is changed.

formulas(assumptions).

% extend the grid with 1 more column and row which are not considered in the
    solution.
(x = 0 | x = ds | y = 0 | y = ds) -> b(x,y) = 0 & c(x,y) = 0.

% each light should be switched less than the nr of states in total,
   otherwise it would end up in its initial state => eliminate redundant
   moves.
c(x,y) < noStates.

```

```
% after applying a number of moves(switches) on the current and neighbour cells, the final state of the current cell should be the state 0.
% that means, if the cell b(x,y) should be changed either directly or through its neighbours if it is not in the correct state.
(x > 0 & x < ds & y > 0 & y < ds & xp = x + -1 & yp = y + -1 & xs = x + 1 & ys = y + 1) -> (b(x,y) + c(x,y) + c(xp,y) + c(x,yp) + c(xs,y) + c(x,ys)) mod noStates = 0.

end_of_list.

% grid: top-left:(0,0) and bottom-right:(Ds,Ds)

formulas(sample).

b(1,1) = 0.
b(1,2) = 1.
b(1,3) = 1.
b(1,4) = 0.
b(1,5) = 2.
b(2,1) = 1.
b(2,2) = 1.
b(2,3) = 1.
b(2,4) = 0.
b(2,5) = 2.
b(3,1) = 0.
b(3,2) = 0.
b(3,3) = 0.
b(3,4) = 1.
b(3,5) = 2.
b(4,1) = 2.
b(4,2) = 2.
b(4,3) = 1.
b(4,4) = 2.
b(4,5) = 0.
b(5,1) = 0.
b(5,2) = 0.
b(5,3) = 2.
b(5,4) = 1.
b(5,5) = 2.

end_of_list.

formulas(goals).
end_of_list.
```

Listing B.6: Puzzle Solver Multiple states

```
import random
from copy import deepcopy

"""
Class representing the current state of the current game session.

Note that a game may consists of multiple game sessions, i.e. of multiple puzzles.
"""


class GameState:
    no_rows = 5
    no_cols = 5
```

```
14
15      """
16      Constructor
17
18      lights_on: a list of list of integers, each with the value 0 or 1,
    representing the initial state of the light
19      bulbs (0=Off, 1=On).
20      """
21
22      def __init__(self, lights_on):
23          self.__initial_lights_on = deepcopy(lights_on)
24          self.__lights_on = lights_on
25          self.__shortest_solution = None
26          self.__no_taken_steps = 0
27          self.__initial_shortest_solution = None
28          self.__initial_shortest_solution_no_steps = None
29
30      """
31      Returns true if and only if the game is won.
32      """
33
34      def is_winning(self):
35          return all(not any(map(bool, lights_row)) for lights_row in self.
    __lights_on)
36
37      """
38      Switches the light bulb in position (x, y).
39
40      As a consequence, the state of the light bulbs adjacent to the switched
    bulb is toggled too.
41
42      If this step is not part of the currently known shortest path to a
    winning state, then the shortest_solution is
43      deleted. Else, it is updated, with the current step being removed from
    it.
44
45      Note that the positions are indexed from 0.
46      X: the row of the light bulb to switch.
47      y; the column of the light bulb to switch.
48      """
49
50      def switch_light(self, x, y):
51          assert 0 <= x < GameState.no_rows and 0 <= y < GameState.no_cols
52          self.__no_taken_steps = self.__no_taken_steps + 1
53          self.__toggle_light(x, y)
54          if x > 0:
55              self.__toggle_light(x - 1, y)
56          if x + 1 < GameState.no_rows:
57              self.__toggle_light(x + 1, y)
58          if y > 0:
59              self.__toggle_light(x, y - 1)
60          if y + 1 < GameState.no_cols:
61              self.__toggle_light(x, y + 1)
62          if self.__shortest_solution is not None:
63              if self.__shortest_solution[x][y] == 1:
64                  self.__shortest_solution[x][y] = 0.
65              else:
66                  self.__shortest_solution = None
67
68      """
```

```
    Informs the GameState about the solutions for the problem from the current state.

    Based on the solutions (it can be mathematically proved, that there are 4 of them), the shortest one is selected
    and cached.

    solutions: a list of solution matrices. Each solution matrix is a list of lists of integers, with a 1 on position
    (x, y) if the light bulb on position (x, y) needs to be toggled, and 0 otherwise.
    """

    def set_solutions(self, solutions):
        shortest_solution_length = GameState.no_cols * GameState.no_rows + 1
        for solution in solutions:
            assert len(solution) == GameState.no_rows
            assert len(solution[0]) == GameState.no_cols
            solution_length = sum([toggle for sublist in solution for toggle in sublist])
            if solution_length < shortest_solution_length:
                shortest_solution_length = solution_length
                self.__shortest_solution = solution
                if self.__lights_on == self.__initial_lights_on:
                    self.__initial_shortest_solution = deepcopy(solution)
                    self.__initial_shortest_solution_no_steps = solution_length

    """
    If previously the solutions for the game were specified, then returns a step (x, y) meaning that by switching
    the light bulb on position (x, y) the winning state will be 1 step closer. If no solutions are known,
    returns None.
    """

    def get_hint_for_next_step(self):
        if self.__shortest_solution is not None:
            solution_vector = [toggle for sublist in self.__shortest_solution for toggle in sublist]
            required_steps = [i for i, toggle in enumerate(solution_vector) if toggle == 1]
            hinted_step_vector_index = random.choice(required_steps)
            hinted_step_row = hinted_step_vector_index / GameState.no_cols
            hinted_step_col = hinted_step_vector_index % GameState.no_cols
            hinted_step_matrix_index = (hinted_step_row, hinted_step_col)
            return hinted_step_matrix_index
        else:
            return None

    """
    Returns true if and only if the shortest solution to the winning state from the current state is known.
    Otherwise false.
    """

    def has_shortest_solution(self):
        return self.__shortest_solution is not None

    """
```

```
    Resets the game to its initial state.
    """

    def reset_to_initial_state(self):
        self.__lights_on = deepcopy(self.__initial_lights_on)
        self.__shortest_solution = deepcopy(self.__initial_shortest_solution
    )
        self.__no_taken_steps = 0

    """
    Returns the total number of steps since the game was started/reset.
    """

    def get_total_no_steps(self):
        return self.__no_taken_steps

    """
    Returns the state of the light bulbs, organized in a list of lists.
    """
    def get_lights_state(self):
        return deepcopy(self.__lights_on)

    """
    Returns true if and only if the shortest solution to the winning state
    from the initial state is known.
    Otherwise false.
    """
    def has_known_shortest_solution_from_initial_state(self):
        return self.__initial_shortest_solution is not None

    """
    Returns the number of steps in the shortest solution from the initial
    state to the winning state.
    """
    def length_of_shortest_solution_from_initial_state(self):
        return self.__initial_shortest_solution_no_steps

    def __toggle_light(self, x, y):
        self.__lights_on[x][y] = 1 - self.__lights_on[x][y]
```

Listing B.7: Game state

```
import os
import random
import sys
from copy import deepcopy

from GameState import GameState

"""
Class responsible for the logic of the game:
- it stores and updates the game state
- generates puzzles and solved them using system calls to mace4.

All communication between the controller and the service layer should be
    done via the GameService, and the controller
should never access the game state directly.
"""


class GameService:
```

```
    NO_GAME_MODELS = 1000

    """
    Initializes the game and starts the first game session.

    To do this, it generates a list of random solvable puzzles and picks one
    of them as the starting game session.
    """

    def __init__(self):
        self.__generate_random_games(GameService.NO_GAME_MODELS)
        self.__game_models = self.__get_generated_game_models()
        self.__played_games = []
        self.__game_state = None
        self.init_new_game_session()

    """Starts a new game session, with a puzzle that was not used before in
    this game, unless all puzzles have
    already been used. """

    def init_new_game_session(self):
        if len(self.__played_games) == GameService.NO_GAME_MODELS:
            self.__played_games = []
        game_model_index = random.randint(0, GameService.NO_GAME_MODELS - 1)
        while game_model_index in self.__played_games:
            game_model_index = random.randint(0, GameService.NO_GAME_MODELS
    - 1)
        self.__played_games.append(game_model_index)
        self.__game_state = GameState(deepcopy(self.__game_models[
    game_model_index]))
        self.__set_game_session_solutions()

    """
    Returns true if and only if the current game session was won.
    """

    def won_game_session(self):
        return self.__game_state.is_winning()

    """
    Switches the light bulb in position (x, y) (indexed from 0) for the
    current game session.
    """

    def switch_light(self, x, y):
        self.__game_state.switch_light(x, y)

    """
    Returns the state of all light bulbs in the current game session, in the
    format of a list of lists of integers,
    where result[x][y] = 1 if the light bulb on position (x, y) is on, and 0
    otherwise.
    """

    def get_lights_state(self):
        return self.__game_state.get_lights_state()

    """
    Returns a step represented by a tuple with two integers, (x, y), meaning
    that by switching the light bulb on
```

```
    position (x, y) the winning state will be 1 step closer.
    """

    def get_hint(self):
        if self.__game_state.has_shortest_solution():
            return self.__game_state.get_hint_for_next_step()
        else:
            self.__set_game_session_solutions()
            return self.__game_state.get_hint_for_next_step()

    """
    Resets the game session to its original state.
    """

    def reset_game_session(self):
        self.__game_state.reset_to_initial_state()

    """
    Returns the number of steps taken since the last game session was
    started/reset to its initial state.
    """

    def get_no_steps_in_game_session(self):
        return self.__game_state.get_total_no_steps()

    """
    Returns the number of steps in the shortest solution from the initial
    state to the winning state.
    """
    def length_of_shortest_solution_from_initial_state(self):
        return self.__game_state.
    length_of_shortest_solution_from_initial_state()

    def __set_game_session_solutions(self):
        game_solutions = map(self.__cut_first_last_row_col, self.
    __generate_game_solutions())
        self.__game_state.set_solutions(game_solutions)

    def __generate_game_solutions(self):
        try:
            with open("GameSolutions/allout_solver_template.in") as
    template_file:
                solver_template = template_file.read()
                solver_mace4_code = solver_template.format(On=self.
    __generate_game_state_propositions_string())
                with open("GameSolutions/allout_solver.in", "w") as
    solver_file:
                    solver_file.write(solver_mace4_code)
                    solver_file.flush()
                    os.system("mace4 -m -1 -f GameSolutions/allout_solver.in
     | "
                              "interpformat portable > GameSolutions/
    allout_solutions.out")
                    with open("GameSolutions/allout_solutions.out") as
    solutions_file:
                        solutions_data = eval(solutions_file.read())
                        solutions = map(self.__transform_solution_data,
    solutions_data)
                        return solutions
        except IOError as e:
```

```
            print "I/O error({0}): {1}".format(e.errno, e.strerror)
        except:  # handle other exceptions such as attribute errors
            print "Unexpected error:", sys.exc_info()[0]
        sys.exit(1)


    def __get_generated_game_models(self):
        f = open("GameModels/allout_math_efficient_generated.out")
        game_models_data = eval(f.read())
        game_models = map(self.__transform_game_model_data, game_models_data
    )
        return game_models

    def __generate_game_state_propositions_string(self):
        lights_state = self.__game_state.get_lights_state()
        lights_state_propositions = []
        lights_state.append([0] * GameState.no_cols)
        lights_state.insert(0, [0] * GameState.no_cols)
        for row, lights_row in enumerate(lights_state):
            lights_row.insert(0, 0)
            lights_row.append(0)
            for col, light_on in enumerate(lights_row):
                lights_state_propositions.append(
                    "On({row}, {col})={on}.".format(row=row, col=col, on=
    light_on))
        return "\n".join(lights_state_propositions)

    def __cut_first_last_row_col(self, matrix):
        matrix.pop(0)
        matrix.pop(len(matrix) - 1)
        for row in matrix:
            row.pop(0)
            row.pop(len(row) - 1)
        return matrix

    @staticmethod
    def __generate_random_games(no_game_models):
        os.system("mace4 -m {nr_game_models} -f GameModels/
    allout_math_efficient_generator.in  | "
                  "interpformat portable > GameModels/
    allout_math_efficient_generated.out".format(
            nr_game_models=no_game_models))

    @staticmethod
    def __transform_game_model_data(game_model_data):
        game_extracted_model = game_model_data[2][7][3]
        return game_extracted_model

    @staticmethod
    def __transform_solution_data(solution_data):
        extracted_solution = solution_data[2][1][3]
        return extracted_solution
```

Listing B.8: Game service

```
from ttk import Style
from Tkinter import *
from PIL import Image, ImageTk
from Tkinter import E, S, N, W
from os import *

```

```
from GameService import GameService

WINDOW_HEIGHT = 450
WINDOW_WIDTH = 550
BOARD_WIDTH = WINDOW_WIDTH * 55 / 100
BOARD_HEIGHT = WINDOW_HEIGHT * 60 / 100
WINDOW_MARGIN_X = WINDOW_WIDTH * 20 / 100
WINDOW_MARGIN_Y = 10
POPUP_WIDTH = 600
POPUP_HEIGHT = 350

"""
This class is responsible for displaying the interface of the game
and handle user actions.
"""


class GameDisplay:

    def __init__(self, no_tiles):
        self.no_tiles = no_tiles
        self.game_service = GameService()
        self.__init_window()
        self.__init_tiles()
        self.__init_board_panel()
        self.__init_button_panel()

    def __init_window(self):
        self.window = Tk()
        self.window.title('All lights out')
        self.window.geometry(
            WINDOW_WIDTH.__str__() + 'x' + WINDOW_HEIGHT.__str__())
        self.window.minsize(250, 350)
        self.window.tk.call("source", "ttk-theme/forest-dark.tcl")
        style = Style()
        style.theme_use("forest-dark")

    def __init_board_panel(self):
        self.board_panel = PanedWindow(master=self.window,
                                       width=BOARD_WIDTH,
                                       height=BOARD_HEIGHT)
        self.board_panel.pack(fill=BOTH, expand=1, side=TOP,
                              padx=WINDOW_MARGIN_X,
                              pady=WINDOW_MARGIN_Y)
        lights_on = self.game_service.get_lights_state()
        for i in range(self.no_tiles):
            # set column and row dimensions to handle window resizing
            self.board_panel.rowconfigure(i, weight=1)
            self.board_panel.columnconfigure(i, weight=1)
            self.tiles.append([])

            for j in range(self.no_tiles):
                self.tiles[i].append(Button(master=self.board_panel,
                                            image=self.tile_state[
                                                lights_on[i][j]],
                                            command=lambda x=i,
                                                           y=j:
                                            self.__on_flip(
                                                x, y)))
                self.tiles[i][j].grid(row=i, column=j,
```

```
                                        sticky=N + S + E + W)

    def __init_tiles(self):
        self.tiles = []
        image_led_on = ImageTk.PhotoImage(
            Image.open("images/yellowButton.png").resize((
                BOARD_WIDTH / self.no_tiles,
                BOARD_HEIGHT / self.no_tiles)))
        image_led_off = ImageTk.PhotoImage(
            Image.open("images/greyButton.png").resize((
                BOARD_WIDTH / self.no_tiles,
                BOARD_HEIGHT / self.no_tiles)))
        self.tile_state = {0: image_led_off, 1: image_led_on}

        self.image_congratulations = ImageTk.PhotoImage(
            Image.open("images/congratulations.png").resize((
                POPUP_WIDTH,
                POPUP_HEIGHT / 2)))

        self.no_frames = self.__count_images_in_gif_dir(
            "images/animatedButton")
        self.frames = [ImageTk.PhotoImage(Image.open(
            "images/animatedButton/greenButton{}.png".format(
                i + 1)).resize((BOARD_WIDTH / self.no_tiles,
                                BOARD_HEIGHT / self.no_tiles))) for i
                       in range(self.no_frames)]
        self.hint_animation_id = None
        self.solution_animation_id = None

    @staticmethod
    def __count_images_in_gif_dir(dir_name):
        no_images = 0
        for base, dirs, files in walk(dir_name):
            for _ in files:
                no_images += 1
        return no_images

    def __refresh_board(self):
        lights_on = self.game_service.get_lights_state()
        for i in range(self.no_tiles):
            for j in range(self.no_tiles):
                self.tiles[i][j].config(
                    image=self.tile_state[lights_on[i][j]])

    def __init_button_panel(self):
        self.button_panel = PanedWindow(master=self.window)
        self.button_panel.pack(fill=BOTH, expand=0, side=TOP,
                               padx=WINDOW_MARGIN_X,
                               pady=WINDOW_MARGIN_Y)
        self.button_panel.rowconfigure(0, weight=1)
        self.button_panel.columnconfigure(0, weight=1)
        self.button_panel.columnconfigure(1, weight=1)
        self.button_panel.columnconfigure(2, weight=1)
        self.hint_button = Button(master=self.button_panel,
                                  text="Hint",
                                  command=self.__on_hint)
        self.hint_button.grid(row=0, column=0)
        self.reset_button = Button(master=self.button_panel,
                                   text="Reset",
                                   command=self.__on_reset)
```

```
        self.reset_button.grid(row=0, column=1)
        self.solve_button = Button(master=self.button_panel,
                                   text="Solve",
                                   command=self.__on_solve)
        self.solve_button.grid(row=0, column=2)
        self.new_game_button = Button(master=self.button_panel,
                                      text="New game",
                                      command=self.__on_new_game)
        self.new_game_button.grid(row=0, column=3)

    def __stop_running_hint_animations(self):
        if self.hint_animation_id is not None:
            self.window.after_cancel(self.hint_animation_id)
            self.hint_animation_id = None

    def __stop_running_solution_animations(self):
        if self.solution_animation_id is not None:
            self.window.after_cancel(self.solution_animation_id)
            self.solution_animation_id = None

    def __display_results_popup(self, game_solved_by_player):
        popup_dialog = Toplevel(self.window)
        popup_dialog.geometry(
            str(POPUP_WIDTH) + "x" + str(POPUP_HEIGHT))
        popup_dialog.title("Game results")
        popup_dialog.resizable(False, False)
        popup_dialog.columnconfigure(0, weight=1)
        popup_dialog.rowconfigure(0, weight=3)
        popup_dialog.rowconfigure(1, weight=1)
        popup_dialog.rowconfigure(2, weight=2)

        # display the congratulatory message only if the puzzle was
        # solved by the player (and not by clicking on the Solve button)
        if game_solved_by_player:
            Label(popup_dialog,
                  image=self.image_congratulations).grid(
                row=0, column=0)
            Label(popup_dialog, text="You solved the game in " + str(
                self.game_service.get_no_steps_in_game_session()) + " steps.
",
                  font=('Mistral 17 bold')).grid(row=1, column=0)

        Label(popup_dialog,
              text="The shortest solution consisted of " + str(
                  self.game_service.
length_of_shortest_solution_from_initial_state()) + " steps.",
              font=('Mistral 15 bold')).grid(
            row=2 if game_solved_by_player else 0, column=0)

    def __set_button_editability(self, enable_hint_button=True,
                                 enable_solve_button=True):
        self.hint_button.config(
            state=NORMAL if enable_hint_button else DISABLED)
        self.solve_button.config(
            state=NORMAL if enable_solve_button else DISABLED)

    def __handle_finish_game(self, game_solved_by_player):
        self.__display_results_popup(
            game_solved_by_player=game_solved_by_player)
        self.__set_button_editability(enable_hint_button=False,
```

```
                                    enable_solve_button=False)

    def __flip_tile(self, x, y, enable_hint_button,
                    enable_solve_button,
                    game_solved_by_player=True):
        # stop any running hint animations
        self.__stop_running_hint_animations()
        self.__set_button_editability(enable_hint_button,
                                      enable_solve_button)
        self.game_service.switch_light(x, y)
        self.__refresh_board()
        if self.game_service.won_game_session():
            # delay showing the results to display the board after
            # the last step
            self.window.after(500, lambda: self.__handle_finish_game(
                game_solved_by_player))

    def __on_flip(self, x, y):
        # tiles cannot be flipped while the solution simulation is
        # running
        if self.solution_animation_id is not None:
            return
        self.__flip_tile(x, y, enable_hint_button=True,
                         enable_solve_button=True)

    def __animate_hint(self, x, y, index=0):
        self.tiles[x][y].config(image=self.frames[index])
        index = (index + 1) % self.no_frames
        self.hint_animation_id = \
            self.window.after(100, lambda: self.__animate_hint(
                x, y, index))

    def __on_hint(self):
        # show hint only if the game has not yet been won
        if not self.game_service.won_game_session():
            # disable the button until the hint animation is stopped:
            # only 1 hint should be played at a time
            self.__set_button_editability(enable_hint_button=False)
            (x, y) = self.game_service.get_hint()
            self.__animate_hint(x, y)

    def __on_reset(self):
        # stop all running animations
        self.__stop_running_hint_animations()
        self.__stop_running_solution_animations()
        # re-enable the buttons
        self.__set_button_editability()
        self.game_service.reset_game_session()
        self.__refresh_board()

    def __next_hint(self, x, y):
        self.__flip_tile(x, y, enable_hint_button=False,
                         enable_solve_button=False,
                         game_solved_by_player=False)
        self.__animate_solution()

    def __animate_solution(self):
        # run simulation only if the game has not yet been won
        if self.game_service.won_game_session():
            self.__stop_running_solution_animations()
```

```
            return
        (x, y) = self.game_service.get_hint()
        self.__animate_hint(x, y)
        self.solution_animation_id = \
            self.window.after(1500, lambda: self.__next_hint(x, y))

    def __on_solve(self):
        # stop previous hint animations and disable Hint button
        self.__stop_running_hint_animations()
        # clear the animation from the board
        self.__refresh_board()
        # disable the Hint and Solve buttons while the current animation is
    running
        self.__set_button_editability(enable_hint_button=False,
                                      enable_solve_button=False)
        self.__animate_solution()

    def __on_new_game(self):
        # stop all running animations
        self.__stop_running_hint_animations()
        # re-enable the buttons
        self.__set_button_editability()
        self.__stop_running_solution_animations()
        self.game_service.init_new_game_session()
        self.__refresh_board()

    def run(self):
        self.window.mainloop()


if __name__ == '__main__':
    game = GameDisplay(no_tiles=5)
    game.run()
```

Listing B.9: Game display