Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Loga Darius
Group: 30432
Email: dariusloga09@yahoo.com

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

Contents

Table 1: Lab scheduling

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

Lab organisation.

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

Chapter 1

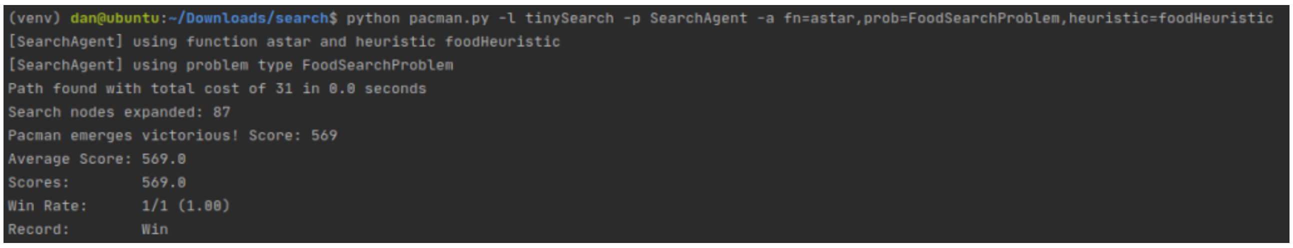
A1: Search

Weighted A^* is a bounded suboptimal search algorithm that is used to find good suboptimal solutions, because sometimes it is too expensive to search for an optimal solution.

It is similar to plain A^* with the difference that the cost from a node to the goal is computed differentely, given by the following formula: f(n) = g(n) + ε * h(n), where $\varepsilon \geq 1$ is the weight addded to the heuristic. A larger ε value results in a faster search with a greater cost.

The weight added to the heuristic makes that nodes which are further away from the goal will be affected significantly more by the added weight and will have very low priority when the algorithm performs its job, thus giving nodes that are closer to the goal a better chance of beign expanded.

During the tests performed on the algorithm I have found out that a weight of around 10 is the upper bound to how fast the algorithm gets (for our specific problem). Anything past that value returned similar values for the speed but an increase in cost. Thus, a value of 5 for the weight seems to strike a good balance between speed and cost.

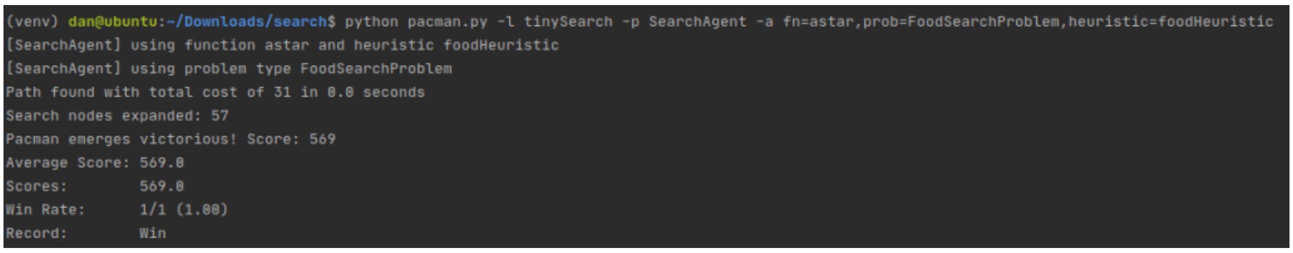


```
(venv) dan@ubuntu:~/Downloads/search$ python pacman.py -l tinySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 31 in 0.0 seconds
Search nodes expanded: 87
Pacman emerges victorious! Score: 569
Average Score: 569.0
Scores:        569.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Fringe search is a path finding algorithm that represent the middle ground between plain A^* and iterative deepening A^*. Fringe search aims to improve on IDA^* by improving the following, repeating steps when there are multiple path to a goal node which is solved by introducing a cache of the visited states.

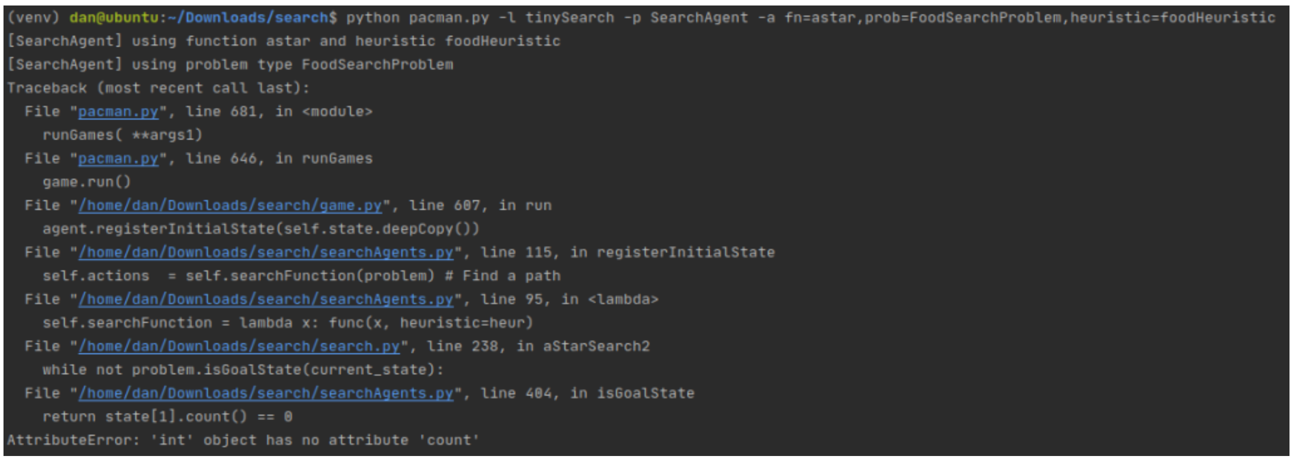
This code is a modification of the plain A^* by introducing some more lists that can hold nodes. This algorithm seems to perform better than A^* by needing to expand less nodes.

It is tested on the food problem, tinyMaze layout and with an heuristic that returns the number of remaining food on the map:



```
(venv) dan@ubuntu:~/Downloads/search$ python pacman.py -l tinySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 31 in 0.0 seconds
Search nodes expanded: 57
Pacman emerges victorious! Score: 569
Average Score: 569.0
Scores:        569.0
Win Rate:      1/1 (1.00)
Record:        Win
```

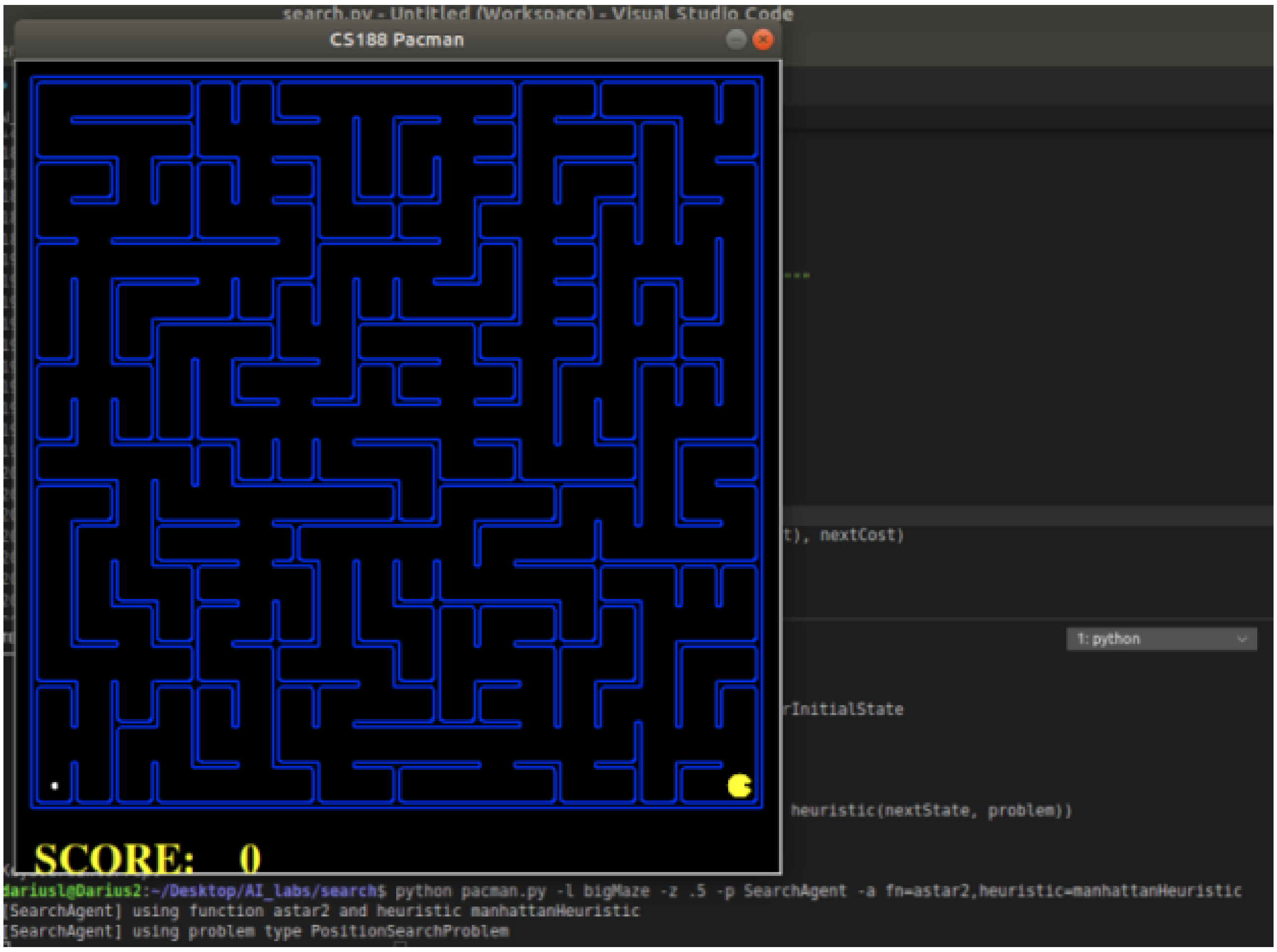
It would seem that it is necessary to enque and pop the starting node before entering the main loop of the program, otherwise the program crashes:



```
(venv) dan@ubuntu:~/Downloads/search$ python pacman.py -l tinySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Traceback (most recent call last):
  File "pacman.py", line 681, in <module>
    runGames( **args1)
  File "pacman.py", line 646, in runGames
    game.run()
  File "/home/dan/Downloads/search/game.py", line 607, in run
    agent.registerInitialState(self.state.deepCopy())
  File "/home/dan/Downloads/search/searchAgents.py", line 115, in registerInitialState
    self.actions  = self.searchFunction(problem) # Find a path
  File "/home/dan/Downloads/search/searchAgents.py", line 95, in <lambda>
    self.searchFunction = lambda x: func(x, heuristic=heur)
  File "/home/dan/Downloads/search/search.py", line 238, in aStarSearch2
    while not problem.isGoalState(current_state):
  File "/home/dan/Downloads/search/searchAgents.py", line 404, in isGoalState
    return state[1].count() == 0
AttributeError: 'int' object has no attribute 'count'
```

We saw online that one variation for A^* is similar to a Dijkstra based algorithm, closely related to Uniformed Cost Search algorithm where we have a directed weighted space that starts from a node and reaches the goal with the minimum cost. We figured out, from the comment's text, that the UCS and A^* are somehow related. We, first, searched for a pseudocode online resembling the UCS algorithm and try to modify it so we can adapt it to our PacMan problem.

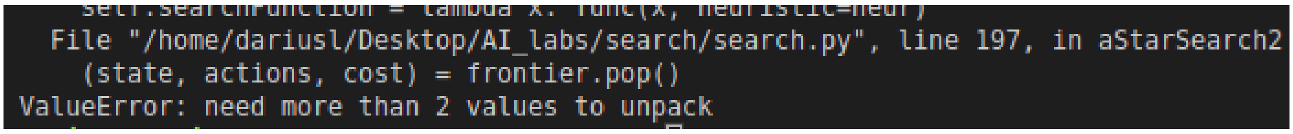
On the wikipedia page, referenced in the Bibliography chapter, we found the pseudocode for the UCS algorithm and had an idea that we can adapt the BFS algorithm, implemented in one lab session, to compute the UCS, implemented only on one of our search.py file, and then adapt it to obtain the desired variation of A^*. We saw that the priority queue has one parameter for setting the priority, so we could choose to have the cost as a parameter for that later in the push() function. We tried our best not to have problems compiling the code, and, luckily, we managed to have no obvious errors. We tested the algorithm on the bigMaze layout and found out that the PacMan is not doing anything just like in the image below:



```
t), nextCost)
rInitialState
heuristic(nextState, problem))
dariusl@Darius2:~/Desktop/AI_labs/search$ python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar2,heuristic=manhattanHeuristic
[SearchAgent] using function astar2 and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
```

Figure 1.1: The PacMan is unable to move

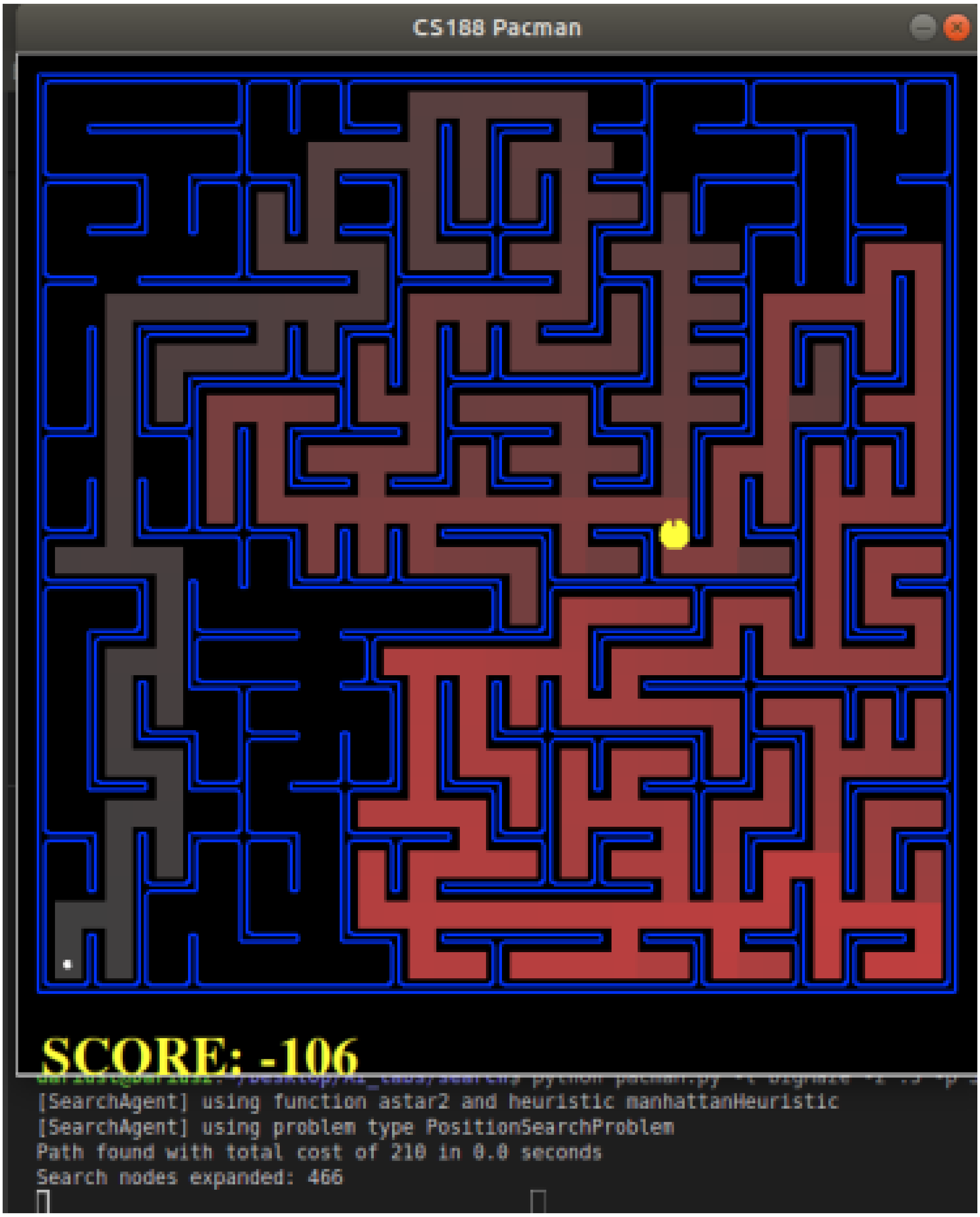
We tested the above variant without any heuristic to the algorithm because we did not know exactly where we should add the heuristic, but, again, it was in front of our eyes, just in the comment text ”combined cost and heuristic”. We added there the heuristic, but no difference. The logic was good, but we were unconscious on what to change next and tried to modify the items in the push() and pop() functions, maybe there is the problem. We took out the last parameter of the first declaration of push() and got the error shown:



```
    self.searchFunction = lambda x: func(x, heuristic=heur)
  File "/home/dariusl/Desktop/AI_labs/search/search.py", line 197, in aStarSearch2
    (state, actions, cost) = frontier.pop()
ValueError: need more than 2 values to unpack
```

Figure 1.2: Removing one parameter

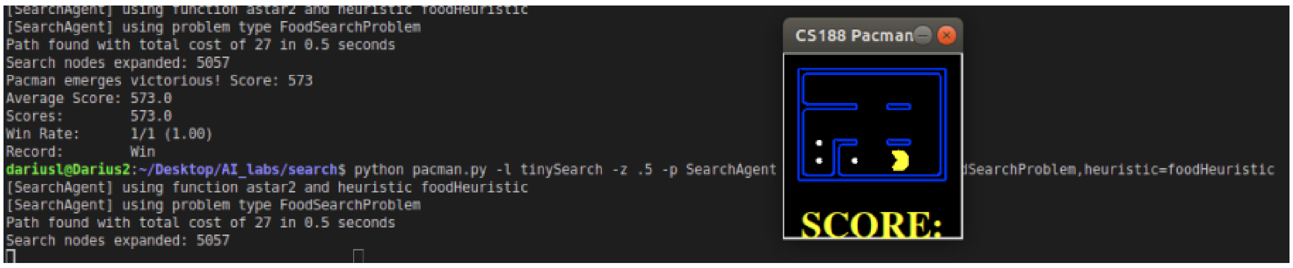
We quickly figured out that we also needed to modify what we wanted to pop(), obvious error, but not so on the spot and also from the push() in the for loop. Removing one parameter still has not resolved the problem. We thought that we might indent further away one tab and, indeed, it was because each statement below the first "if" should have been one tab away. After all of that errors, the algorithm worked as intended, and below is an image of the test on the bigMaze layout:



```
[illegible]
[SearchAgent] using function astar2 and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 466
```

Figure 1.3: The PacMan is able to move and performs the task

We also took a test, just in case something would not run in normal terms, for the food-Heuristic on the tinySearch layout and it was also a success, as you can see from the image underneath:



```
[SearchAgent] using function astar2 and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 27 in 0.5 seconds
Search nodes expanded: 5057
Pacman emerges victorious! Score: 573
Average Score: 573.0
Scores:        573.0
Win Rate:      1/1 (1.00)
Record:        Win
dariusl@Darius2:~/Desktop/AI_labs/search$ python pacman.py -l tinySearch -z .5 -p SearchAgent                 dSearchProblem,heuristic=foodHeuristic
[SearchAgent] using function astar2 and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 27 in 0.5 seconds
Search nodes expanded: 5057
```

Figure 1.4: The test for the foodHeuristic

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Chapter 4

Bibliography

https://inst.eecs.berkeley.edu/~cs188/fa18/assets/sections/final_review_solutions.pdf
https://webdocs.cs.ualberta.ca/~holte/Publications/fringe.pdf
https://en.wikipedia.org/wiki/Fringe_search
https://en.wikipedia.org/wiki/A*_search_algorithm#Relations_to_other_algorithms
Dijkstra Wikipedia
Own laboratory's work

Appendix A

Your original code

```
# Weighted A*
def aStarSearch1(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    priority_queue = util.PriorityQueue()
    visited = []
    start_state = (problem.getStartState(), [], 0)
    priority_queue.push(start_state, 0)
    while not priority_queue.isEmpty():
        state = priority_queue.pop()
        if problem.isGoalState(state[0]):
            return state[1]
        if not state[0] in visited:
            visited.append(state[0])
            successors = problem.getSuccessors(state[0])
            for successor in successors:
                if not successor[0] in visited:
                    cost = state[2] + successor[2]
                    total_cost = cost + 5 * heuristic(successor[0], problem)
                    new_state = (successor[0], state[1] + [successor[1]], cost)
                    priority_queue.push(new_state, total_cost)
    return nullHeuristic(problem)
# Fringe search
def aStarSearch2(problem, heuristic=nullHeuristic):
    fringe = util.PriorityQueue()
    visited = []
    temp = []
    later = []
    now = util.PriorityQueue()
    fringe.push(problem.getStartState(), 0)
    current_state = fringe.pop()
    while not problem.isGoalState(current_state):
        if current_state not in visited:
            visited.append(current_state)
            successors = problem.getSuccessors(current_state)
            for successor in successors:
                temp = later + [successor[1]]
                total_cost = problem.getCostOfActions(temp) + heuristic(successor[0], problem)
                if successor[0] not in visited:
                    fringe.push(successor[0], total_cost)
                    now.push(temp, total_cost)
        current_state = fringe.pop()
        later = now.pop()
    return later
#Variation of Dijkstra's algorithm using heuristic
def aStarSearch3(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    frontier = util.PriorityQueue()
    frontier.push((problem.getStartState(), []), 0)
    explored = set()
    while not frontier.isEmpty():
        (state, actions) = frontier.pop()
        if state not in explored:
            explored.add(state)
            if problem.isGoalState(state):
                return actions
            for nextState, nextActions, nextCost in problem.getSuccessors(state):
                frontier.push((nextState, actions + [nextActions]), nextCost + heuristic(nextState, problem))
    return []
```