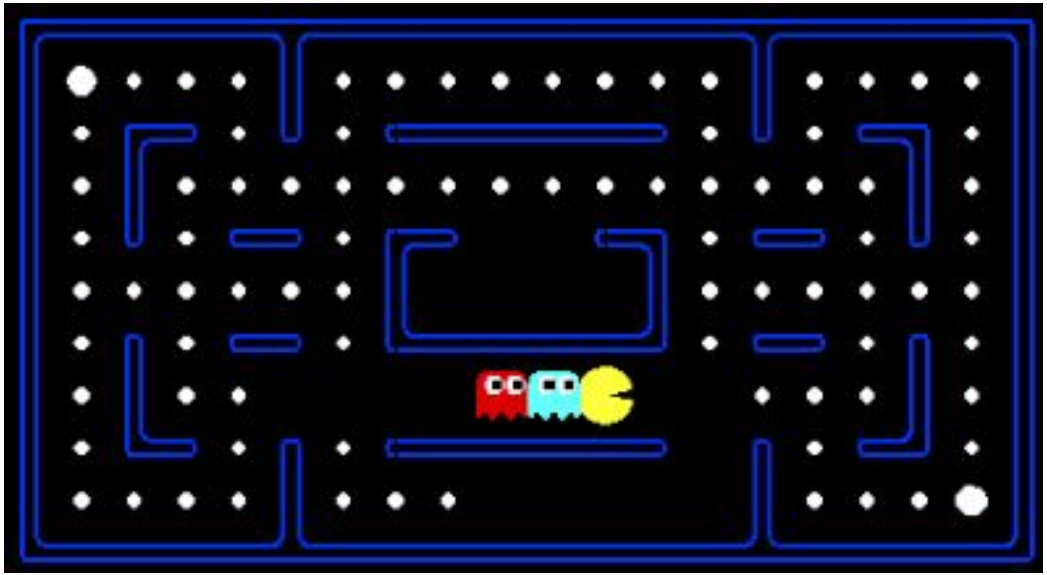
YS02 Artificial Intelligence Project 1: Search

Konstantinos Plas **kplas@di.uoa.gr**

Logistics

- Project: homework 1
- Deadline: 14/11/2022
- Submission: eclass

The PacMan Project



Search Problems

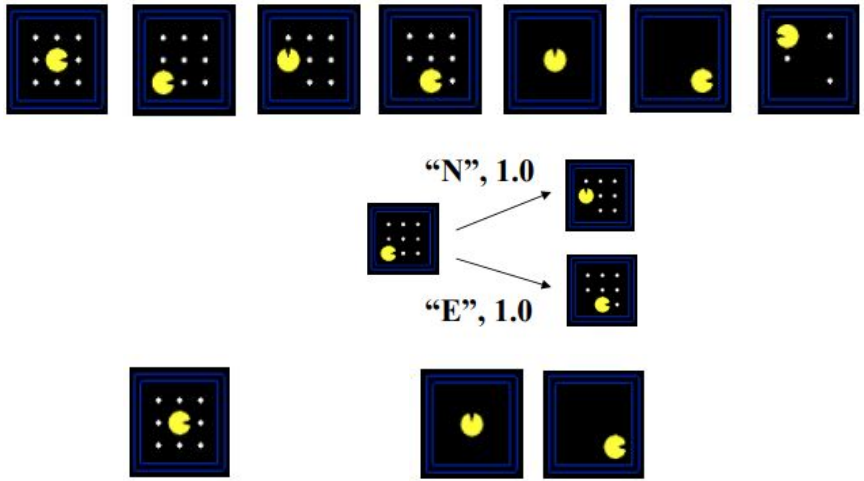
Consist of:

- State Space: all the possible states on the problem's world
- Start State: the starting state of the agent
- Goal State: the state of the problem the agent must reach
- Successor Function: function that takes as input a state and an action and outputs the next states the agent can reach and the cost of reaching them

The solution is a sequence of actions from the start state to the goal state

Search Problems: Pacman

- State Space
- Successor Function
- Start and Goal state



Modeling a Search Problem

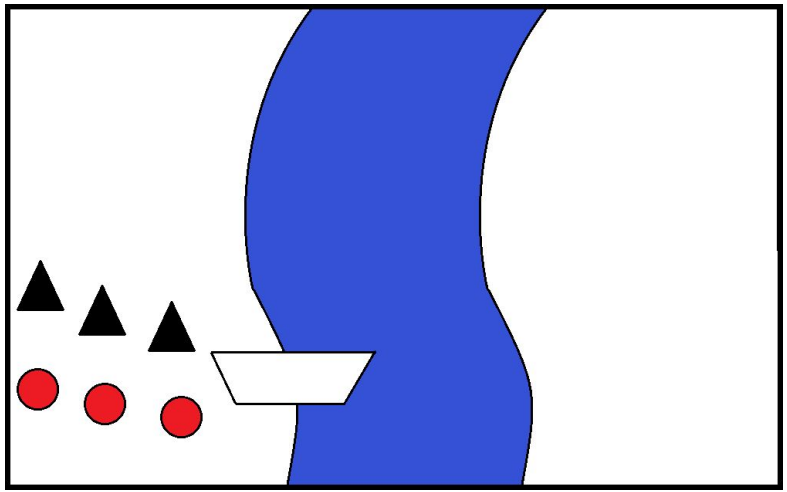
- Given a real world problem, how can we formulate it into a search problem?

- We will focus on two examples:
  - Missionaries and Cannibals
  - 8 puzzle problem

Modeling the Missionaries & Cannibals Problem

- **Problem:** On one bank of a river are **3** missionaries and **3** cannibals. There is **1** boat available that can carry at most **2** people and that they would like to use to cross the river. If the cannibals ever outnumber the missionaries on either of the river's banks or on the boat, the missionaries will get eaten.

  - Red circles represent the 3 cannibals
  - Black triangles represent the 3 missionaries



Modeling the Missionaries & Cannibals Problem

- **Goal:** Move all missionaries and cannibals to the other side of the river
  - **Question:** How can we formulate the given problem into a graph search problem?

- Remember what constitutes a search problem:
  - **State Space / States**
  - **Start State**
  - **Goal State**
  - **Successor Function**

Modeling the Missionaries & Cannibals Problem

- **State:** a good representation for a state of world would be a tuple with 6 numbers. The first three numbers *Ml*, *Cl*, *Bl* represent the number of missionaries, cannibals and boats on the left side of the river, and the last three numbers Mr, Cr, Br the number on the right side of the river. A state therefore could be represented as:

  State = (Ml, Cl, Bl, Mr, Cr, Br)

- **InitialState = *(3, 3, 1, 0, 0, 0)*** , the boat and all missionaries and cannibals are on the left side of the river.

- **GoalState = *(0, 0, 0, 3, 3, 1),*** all missionaries and cannibals reached across the river safely.

Modeling the Missionaries & Cannibals Problem

- **Actions:** move the boat across the river with 1 or 2 people.
- **Successor Function:** if the boat is on the left side of the river the states generated will be:

  { (Ml-1, Cl, 0, Mr+1, Cr, 1), (Ml-1, Cl-1, 0, Mr+1, Cr+1, 1), (Ml, Cl-1, 0, Mr, Cr+1, 1)
  (Ml-2, Cl, 0, Mr+2, Cr, 1), (Ml, Cl-2, 0, Mr, Cr+2, 1) }

  Accordingly, if the boat is on the right side:

  { (Ml+1, Cl, 1, Mr-1, Cr, 0), (Ml+1, Cl+1, 1, Mr-1, Cr-1, 0) . . . }

- **Question:** Are all the states the successor function produces legal?

Modeling the Missionaries & Cannibals Problem

- **Question:** Are all the states the successor function produces legal?
  - **Answer:** not always we must check whether a state has more cannibals than missionaries in either side of the river. When generating a successor the condition ***(Ml >= Cl AND Mr>= Cr)*** must be true, for it to be considered.
- For example if we consider the state (3, 3, 1, 0, 0, 0) the states generated from the successor function will be:

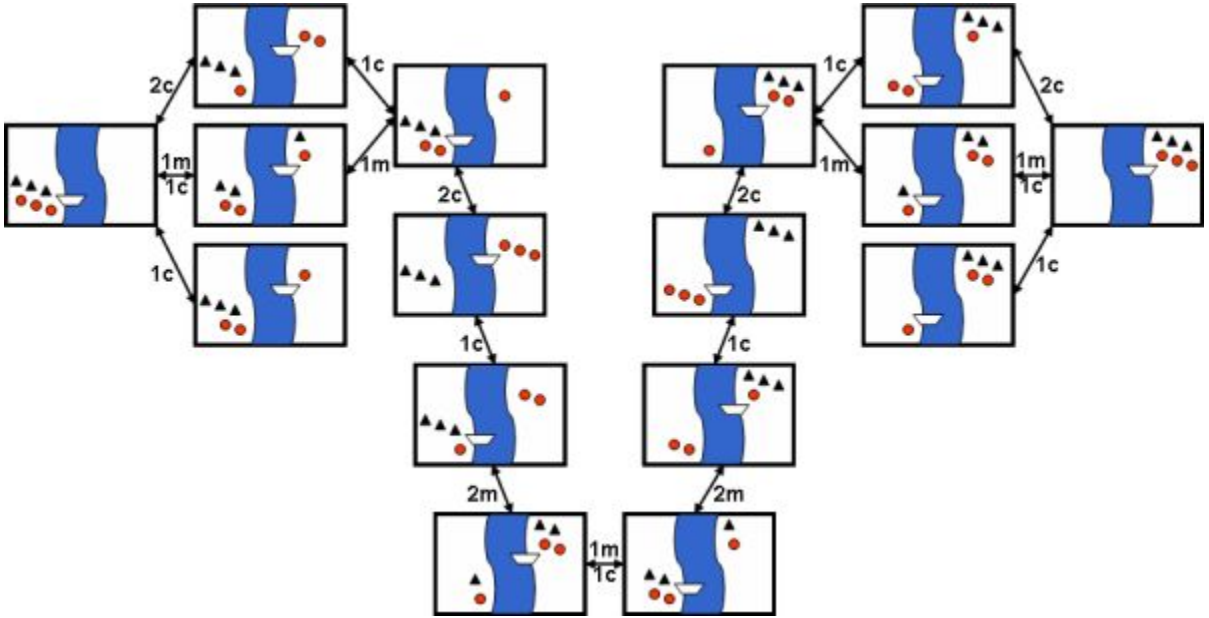
{(2, 3, 0, 1, 0, 1), (2, 2, 0, 1, 1, 1), (3, 2, 0, 0, 1, 1), (1,3,0,2,0,1), (3,1,0,0,2,1)}

The first and fourth generated states have more cannibals than missionaries on the left side of the river. These states are generated from illegal actions and are not considered: Thus the actual successors of (3, 3, 1, 0, 0, 0) will be:

{(2, 2, 0, 1, 1, 1), (3, 2, 0, 0, 1, 1), (3,1,0,0,2,1)}

Modeling the Missionaries & Cannibals Problem

The search space of missionaries and cannibals problem:



Modeling the 8 puzzle problem

1	3	2
5		6
8	7	4

Initial State : I

1	2	3
4	5	6
7	8	

Goal State : G

- **Goal:** Move the tiles using the empty space to reach the final goal G

Modeling the 8 puzzle problem

- **State Space :** All possible combinations for the puzzle → 9!/2 = 181, 440 states
    - Let ***S*** be the { All possible combination of 8 puzzle }

- **State =** ***((x,y) , P)*** where:
    - ***P ∈ S*** , ***P*** is the current image of the puzzle
        - ***P = { p_{11}, p_{12}, p_{13}, p_{21}, …, p_{33}}***, where p_i□ is the value of the tile in (i,j)
    - ***(x,y)***, is the coordinates of the empty space in ***P***

- **InitialState =** ***((2,2) , I)***

- **GoalState =** ***((3,3), G)***

Modeling the 8 puzzle problem

- **Actions =** *{UP, DOWN, LEFT, RIGHT}*, swap the empty space with a tile either up, down, left or right
- **Successor Function :** The successor function for a given state $s_i = ((x_i, y_i), P_i)$ outputs
  - $succ(s_i) = \{ ((x_i - 1, y_i), P_1), ((x_i + 1, y_i), P_2), ((x_i, y_i - 1), P_3), ((x_i, y_i + 1), P_4) \}$, where $(x_i - 1, y_i)$ indicates that the empty space was swapped with the tile above it, $(x_i + 1, y_i)$ with the tile below it etc.
  - P_1, is the puzzle image produced if the empty tile was swapped with the tile above it etc.
- **isGoalState :** function that for a given state s_i outputs:

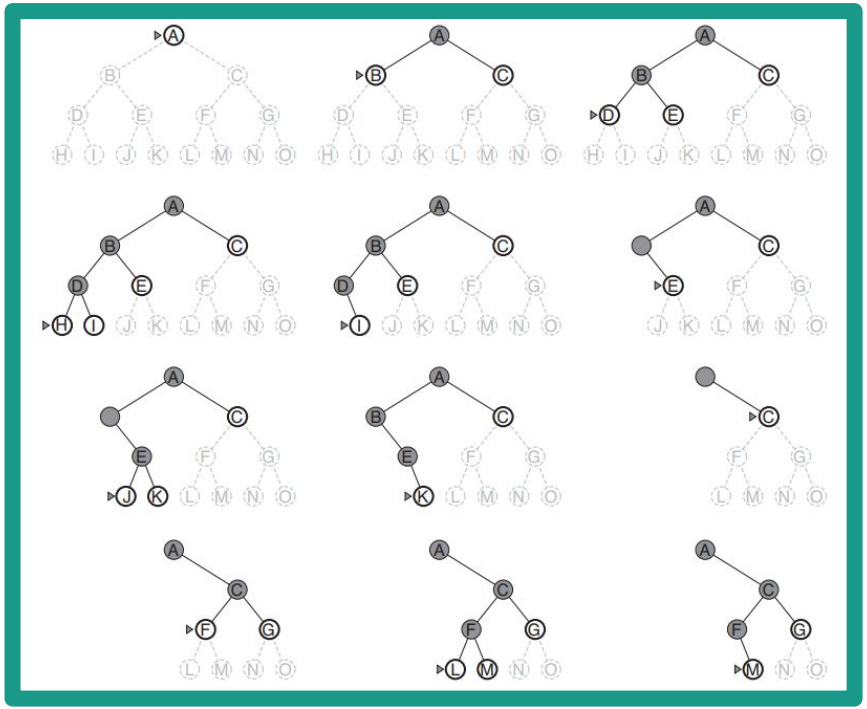
$$isGoalState(s_i) = isGoalState((x_i, y_i), P_i) = \begin{cases} 1, & \text{if } P_i = G \\ 0, & \text{otherwise} \end{cases}$$

Graph Search Algorithms

- After we formulate a real world problem into a search problem we can utilize graph search algorithms to solve it:
  - DFS
  - BFS
  - UCS
  - A*

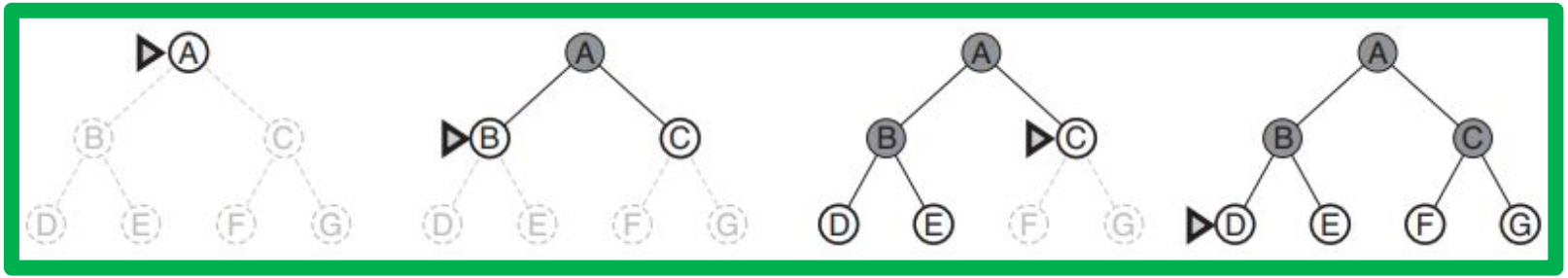
Depth First Search (DFS)

- Strategy: expand nodes depth-wise until a node has no successors
- Implementation: frontier is a stack



Breadth First Search (BFS)

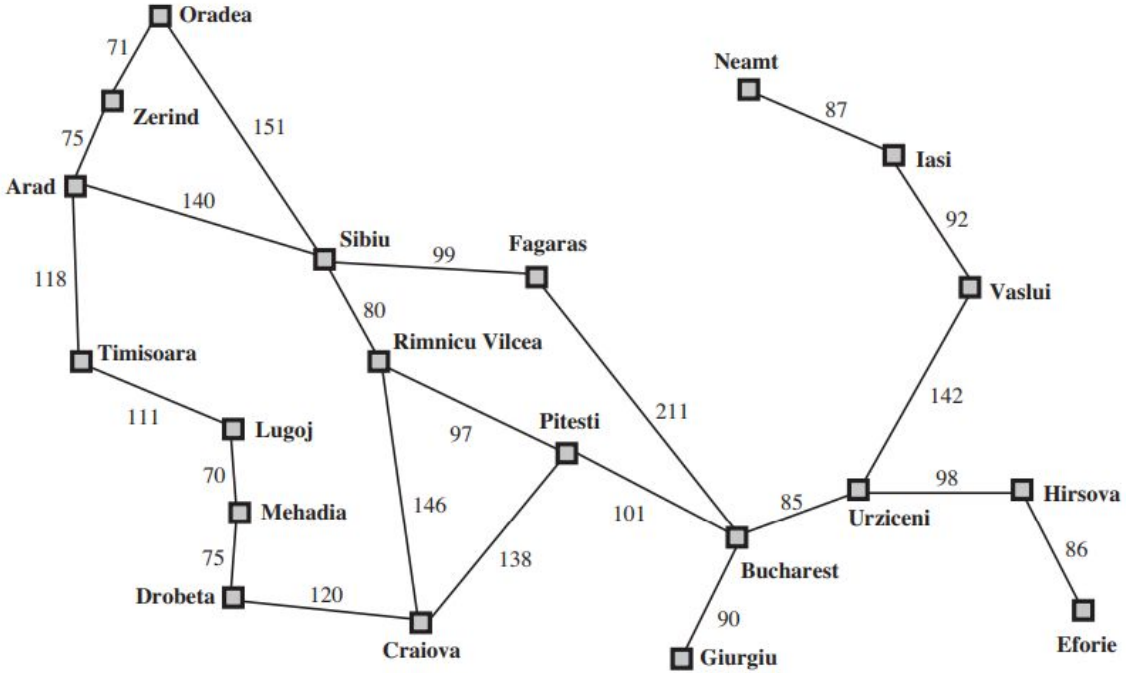
- Strategy: expand nodes layerwise
- Implementation: frontier is a queue

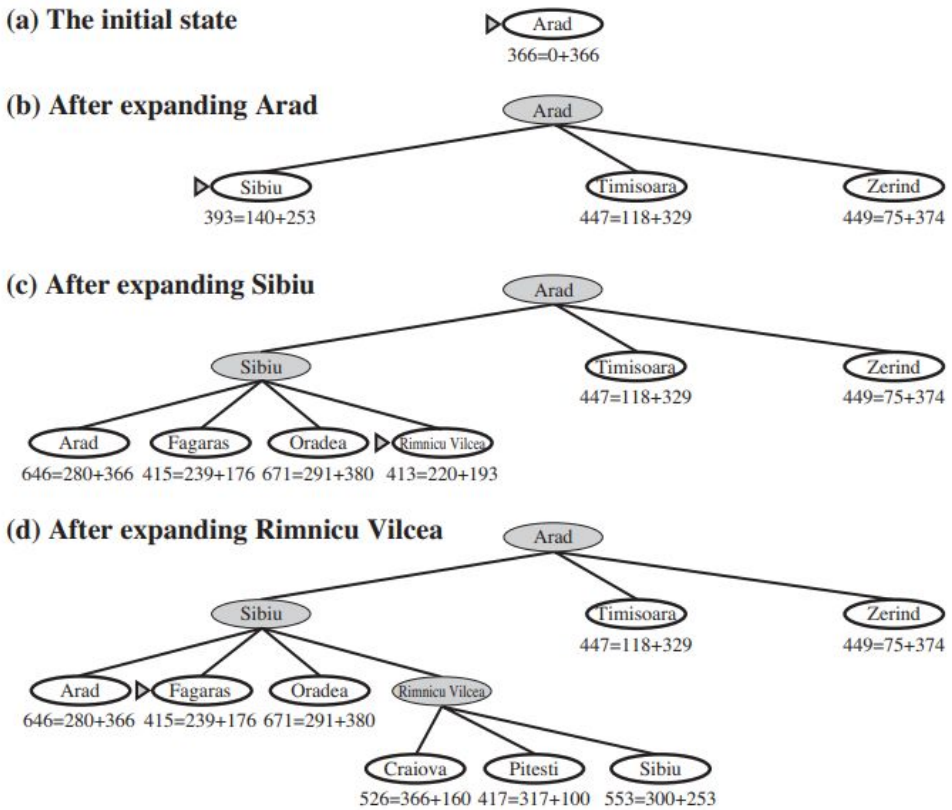


A* (A star)

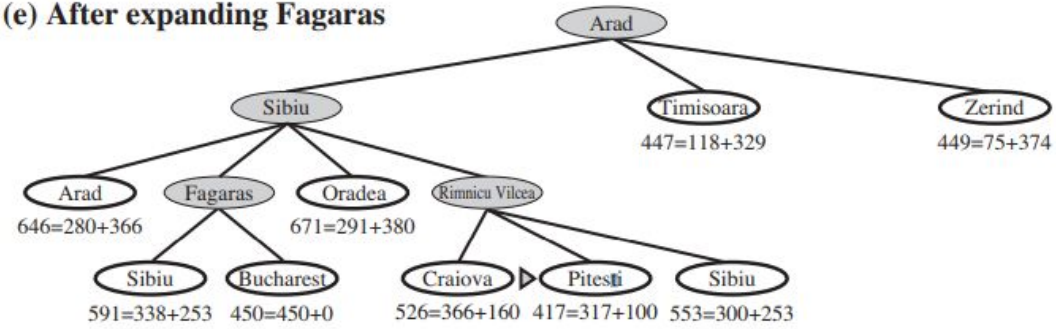
- UCS : sorts nodes according to cost g(n)
- A* : expansion of UCS, nodes are sorted based on the sum g(n) + h(n)
    - g(n): cost to reach a node n from the root node
    - h(n): heuristic function to approximate the solution

A* : Execution Example

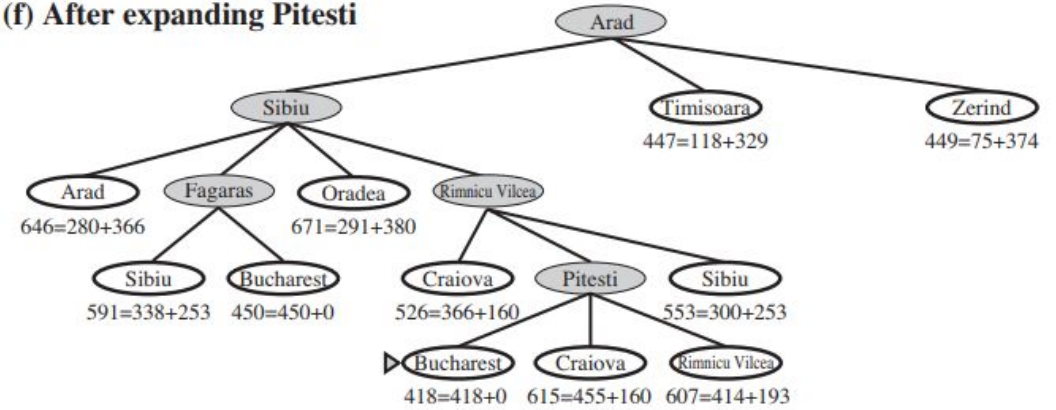



(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
393=140+253
447=118+329
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253

(e) After expanding Fagaras



(f) After expanding Pitesti



Heuristic Function

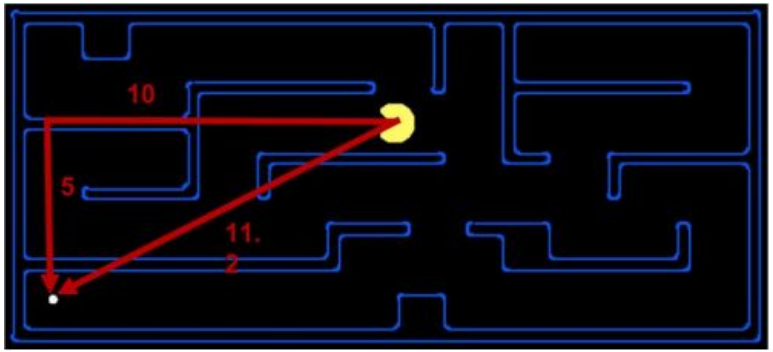
- For a given state, estimates the cost from that state to the goal state
- **Consistent:** The estimation is less than or equal to the estimation of a neighboring state plus the cost to reach that state
  - $h(s) \leq c(s, a, s') + h(s')$
- **Admissible:** It does not overestimate the cost to reach the goal state
  - $0 \leq h(s) \leq h^*(s)$
- All consistent heuristics are admissible. The opposite is necessarily not true.

Heuristic Function : How to choose a heuristic

- A heuristic is formulated based on the problem we try to solve
- Non consistent functions may prevent the search algorithms from exploring “good” paths.
- We can easily formulate a consistent heuristic if we consider a simpler problem (relaxation).

Heuristic Function : Pacman

- **Euclidean Distance**
  - Euclidean distance from the goal
  - For the given example ≈ 11.2
- **Manhattan Distance**
  - Manhattan distance from the goal
  - For the given example = 15
- The actual distance is greater because of obstacles
- By simplifying the problem it is easier to find “good” heuristics



Heuristic Function : 8 puzzle

- **Hamming Distance**
  - Tiles out of place
  - For the given example = 7
- **Manhattan Distance**
  - Manhattan distance of each tile for the goal position
  - For the given example = 10
    - h = 0+1+1+3+1+0+1+1+2

1	3	2
5		6
8	7	4

1	2	3
4	5	6
7	8	

Project 1

- ★ Important files:
  - pacman.py → Pacman main file (GameState classes)
  - game.py → The logic behind Pacman environment (Agent,Direction classes)
  - util.py → Useful structure classes (Stack, Queue, PriorityQueue classes)
- ★ Files to edit:
  - search.py → Here you will implement the search algorithms (Q1-Q4)
  - searchAgents.py → Search based agents (Q5-Q8)

Project 1 : Questions 1-4

```
Algorithm: GRAPH_SEARCH:
 frontier = {startNode}
 expanded = {}
 while frontier is not empty:
     node = frontier.pop()
    if isGoal(node):
          return path_to_node
    if node not in expanded:
          expanded.add(node)
          for each child of node's children:
               frontier.push(child)
 return failure
```

- Generic algorithm:
  - DFS (Q1)
  - BFS (Q2)
  - UCS (Q3)
  - A* (Q4)
- Different frontiers for each algorithm:
  - Stack (DFS)
  - Queue (BFS)
  - PriorityQueue (UCS, A*)
- Node design is important
- Expanded should be a python set

Project 1: Question 5

- **Goal:** Define an abstract representation of the Corners Problem
  - How can we model this search problem?
  - Create a representation for start and goal state
  - Design the successor function [expand]
    - Return the next possible states, the actions required to reach them and their cost
    - Consider also the possibility that the next state is the goal state

Project 1: Question 6

- **Goal:** Write a non-trivial, non-decreasing consistent heuristic

➢ How to design a heuristic for the corners problem ?
  - Consider an intermediate state of the problem
  - Get the unvisited nodes
  - Think of ways to compute the distance to the nodes
  - Visit the corner that is closer

Project 1: Question 7

- **Goal:** Eat all the pacman food in as little steps as possible

➢ Key items to use in foodHeuristic:
  - **foodGrid.asList:** Get a list of food coordinates
  - **problem.heuristicInfo:** A dictionary provided to store the information required to be reused in other calls of the heuristic

Project 1: Question 8

- **Goal:** Write an agent that always greedily eats the closest dot

➢ Functions you will need to implement:
  - **ClosestDotSearchAgent.findPathToClosestDot :** Returns a path to the closest dot starting from gameState (Hint: You've already implemented that)
  - **AnyFoodSearchProblem.isGoalState:** Returns whether we have reached the goal state