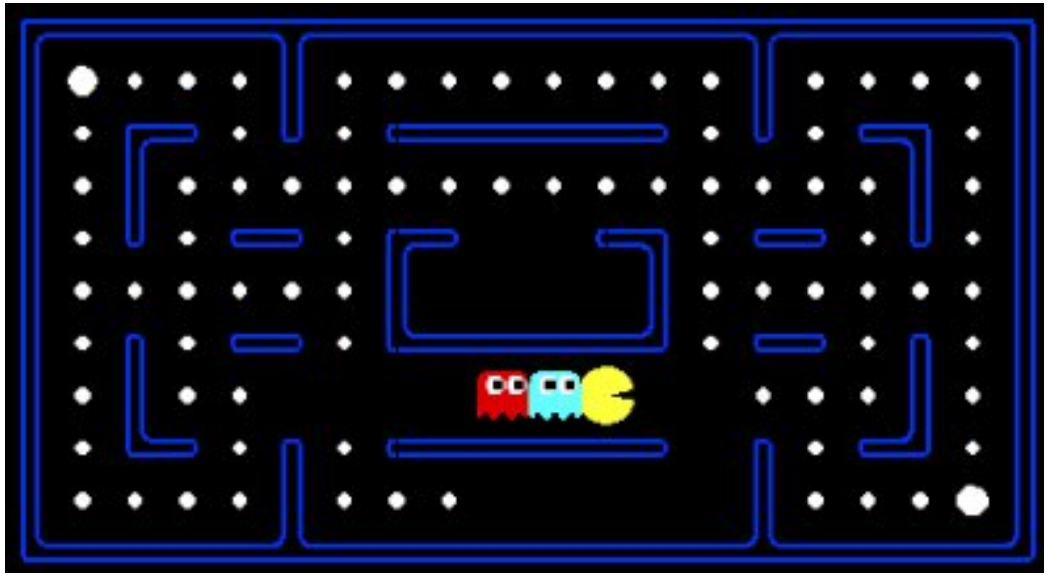
YS02 Artificial Intelligence
Project 1: Search

Mohammad Matin Marzie matinm@di.uoa.gr
Dimitris Tsantopoulos dimtsanto@di.uoa.gr
Nikolas Tourgelis nikolic8@di.uoa.gr
Konstantinos Chousos kchousos@di.uoa.gr

Logistics

- Project: Homework 1
- Deadline: 3/11/2026, 23:59
- Questions: On Piazza

The PacMan Project





Search Problems

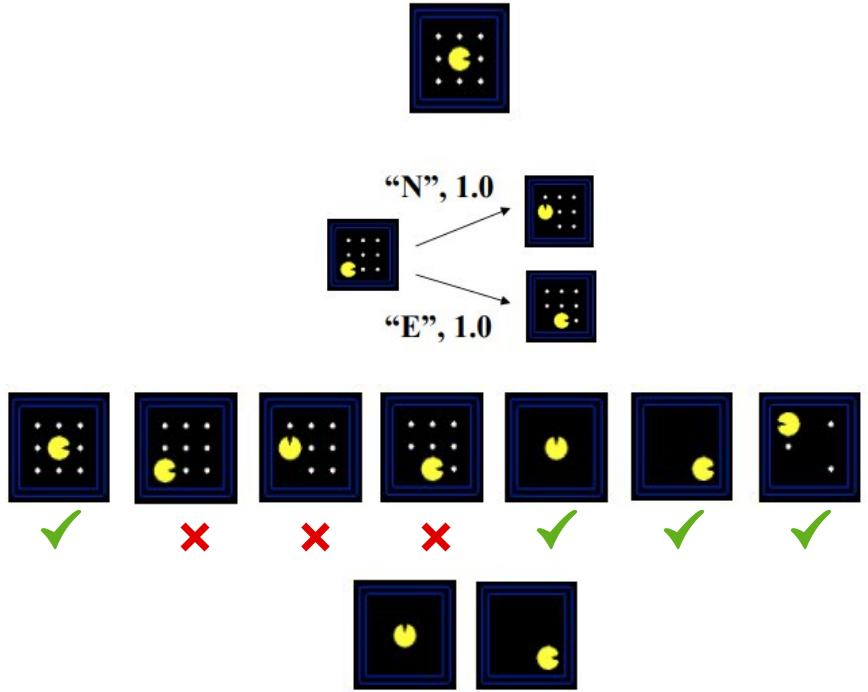
Consist of:

- Start State: the starting state of our problem
- Successor Function: function that takes as input a state and outputs the available actions
- State Space: all the possible states on the problem's world
- Goal State: the state of the problem the agent must reach

The solution is a sequence of actions from the Start State to the Goal State.

Search Problems: Pacman

- Start State
  - Pacman begins from the middle of the grid
- Successor Function
  - Pacman can move vertically or horizontally, but is blocked by walls
- State Space
  - All possible states of our “problem world” starting from the Start State and acting only as the Successor Function allows
- Goal State
  - Pacman has eaten all the food



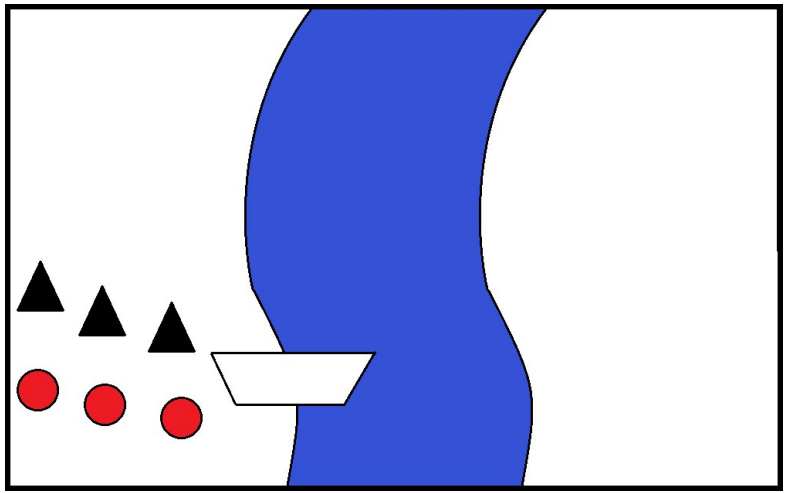
Modeling a Search Problem

- 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, the missionaries will get eaten.



Red circles represent cannibals
Black triangles 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:
  - Start State: where do we start?
  - Successor Function: what actions can we take?
  - State Space: which are all the valid states of our world?
  - Goal State: what do we want to accomplish?



Modeling the Missionaries & Cannibals Problem

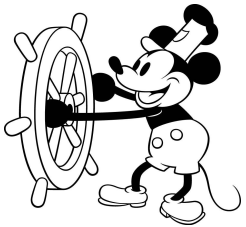
- **State:**
  - a tuple of 6 numbers
    - M_L, C_L, B_L the number of missionaries, cannibals and boats on the left side of the river
    - M_R, C_R, B_R the number of missionaries, cannibals and boats on the right side of the river
  - **State =** $(M_L, C_L, B_L, M_R, C_R, B_R)$

- StartState = $(3, 3, 1, 0, 0, 0)$, 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 crossed the river without any "*accidents*".

Modeling the Missionaries & Cannibals Problem

- **Actions:** move the boat across the river with 1 or 2 people.
  - if the boat is on the left side of the river the possible actions are:



    - *move 1 missionary to the right side* $(M_L\text{-}1, C_L, 0, M_R\text{+}1, C_R, 1)$
    - *move 2 missionaries to the right side* $(M_L\text{-}2, C_L, 0, M_R\text{+}2, C_R, 1)$
    - *move 1 cannibal to the right side* $(M_L, C_L\text{-}1, 0, M_Rr, C_R\text{+}1, 1)$
    - *move 2 cannibals to the right side* $(M_L, C_L\text{-}2, 0, M_R, C_R\text{+}2, 1)$
    - *move 1 missionary and 1 cannibal to the right side* $(M_L\text{-}1, C_L\text{-}1, 0, M_R\text{+}1, C_R\text{+}1, 1)$
  - likewise for the right side…
- Question: Are all these actions legal?
  - hint: if they are all legal, what do we need the Successor Function for?
  - reminder: the Successor Function takes as input a state and outputs the available actions.

Modeling the Missionaries & Cannibals Problem

- Question: Are all theoretically possible actions always 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 ***(M_L >= C_L AND M_R >= C_R)*** must be true, for it to be considered (responsibility of the Successor Function).
- For example if we consider the Start State *(3, 3, 1, 0, 0, 0)* and any possible action we get the following states:

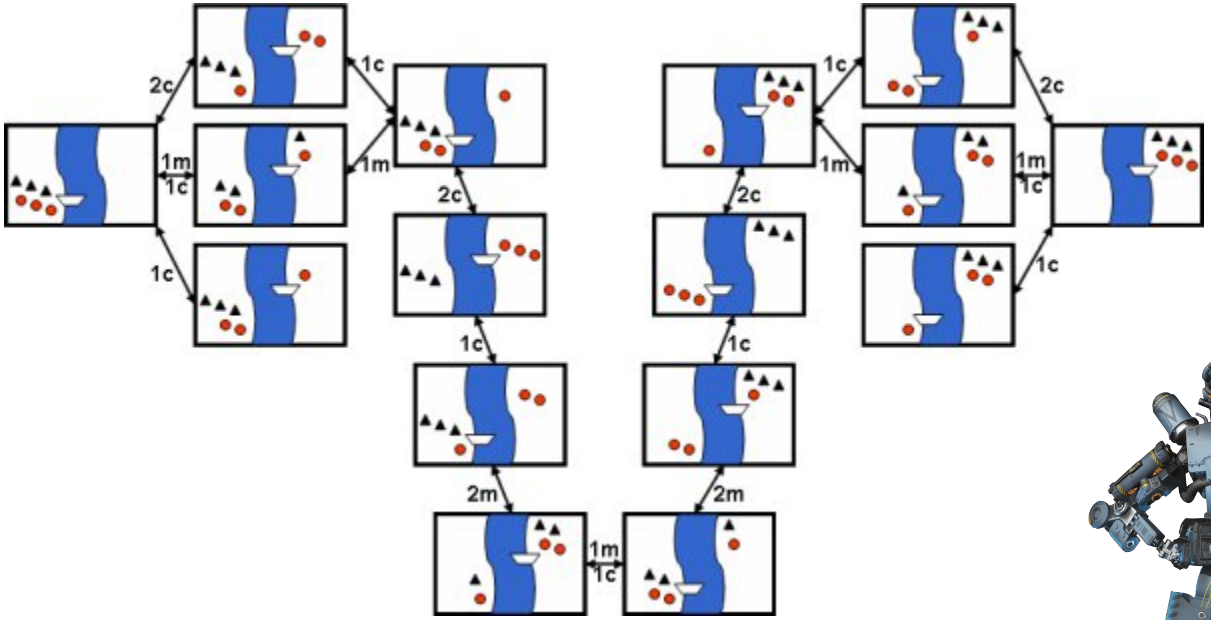
{(2, 3, 0, 1, 0, 1), (2, 2, 0, 1, 1, 1), (3, 2, 0, 0, 1, 1), (1,3,0,2,0,1), (3,1,0,0,2,1)}

- 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 generated by the Successor Function are:

{ (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

Start State I

1	2	3
4	5	6
7	8	

Goal State G

Modeling the 8 puzzle problem

- **State Space *(S)*:** All possible solvable combinations for the puzzle → 9!/2 = 181.440 states

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

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

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

Modeling the 8 puzzle problem

- **Actions =** swap the empty space with one of its neighbors (up, down, left, right)
- 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:

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

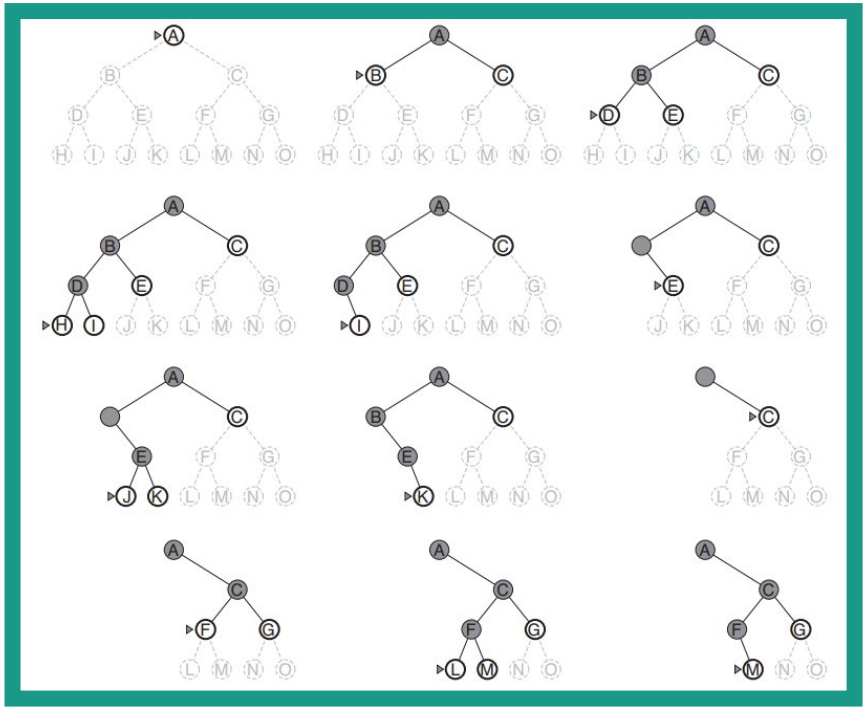
Graph Search Algorithms

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



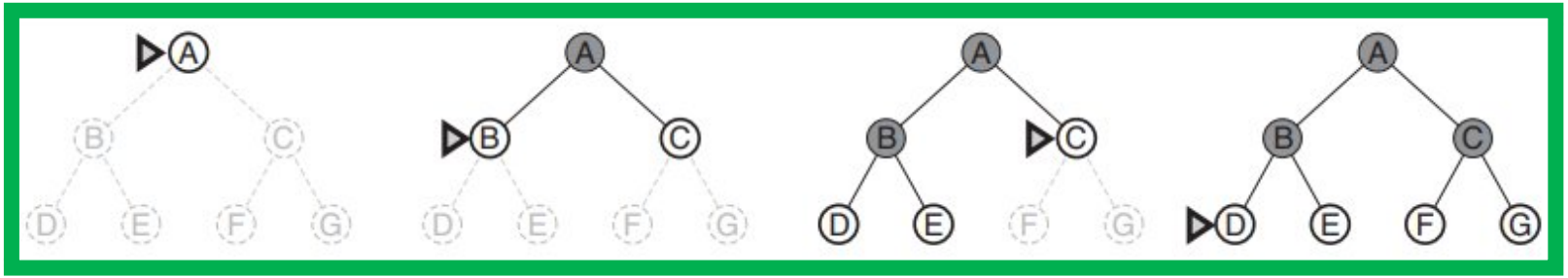
Depth First Search (DFS)

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



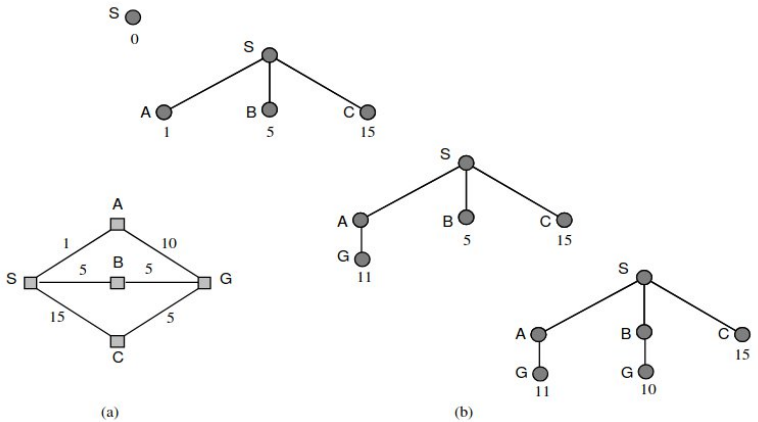
Breadth First Search (BFS)

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



Uniform Cost Search (UCS)

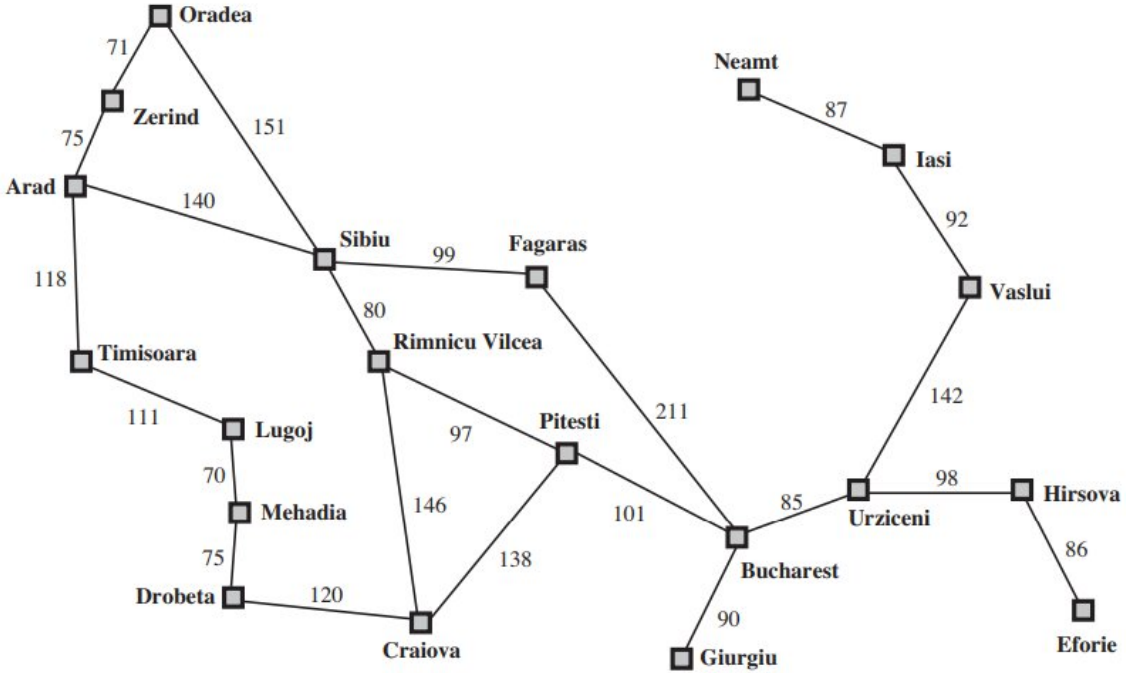
- UCS : sorts nodes according to cost g(n)
  - Like BFS (*min-depth*) but for Graphs with different path costs (*min-cost*)



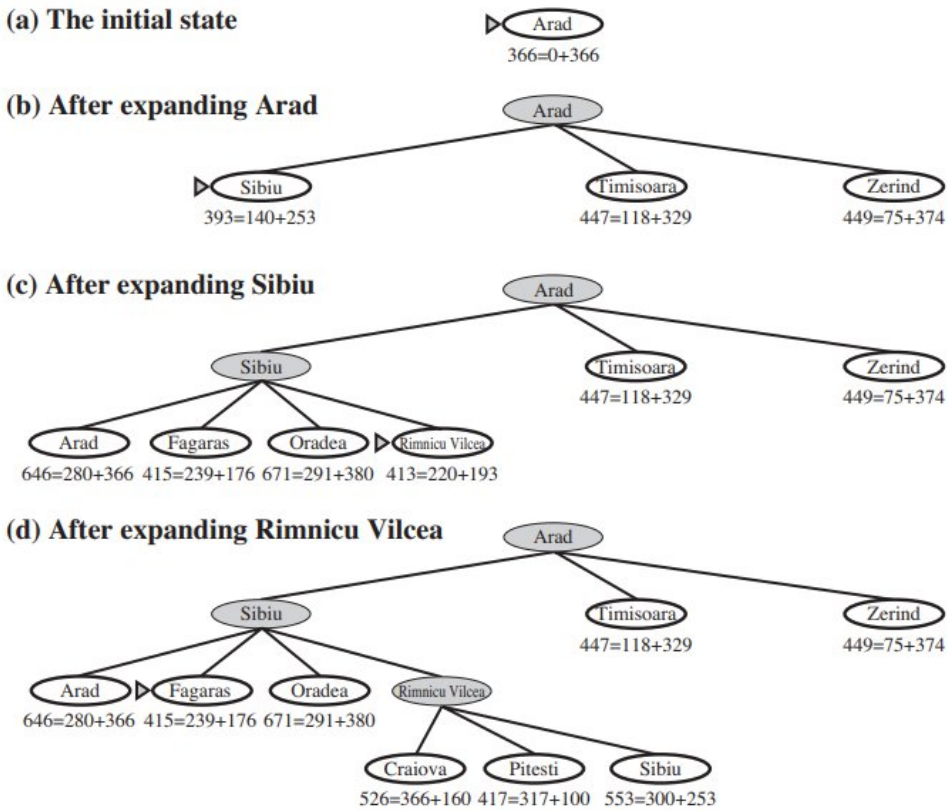
A* (A star)

- 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

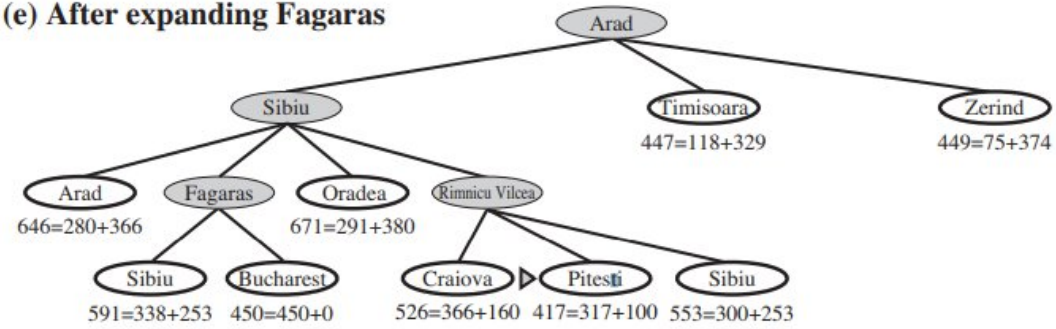
A* : Execution Example



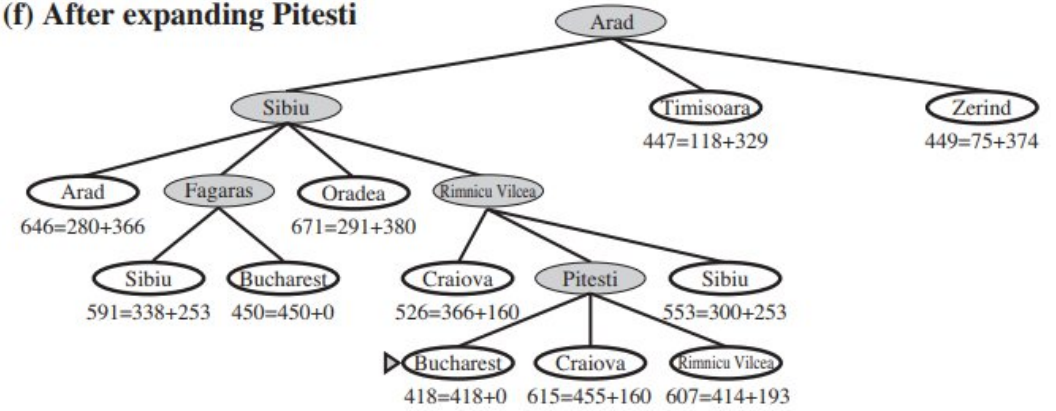
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374


(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

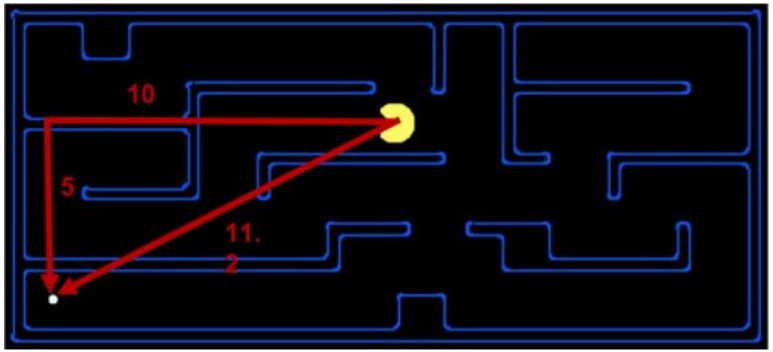
- For a given state, estimates the cost from that state to the Goal State
- **Trivial:** Always returns 0 (same as UCS) or always returns the true cost
- **Admissible:** It does not overestimate the cost to reach the goal state
  - $0 \leq \text{heuristic_cost}(s) \leq \text{true_cost}(s)$
- **Consistent:** The estimation is less than or equal to the estimation of a neighboring state plus the cost to reach that state
  - $h(s) \leq c(s, a, s') + h(s')$
  - intuition: That is, you don't think that it costs 5 from B to the goal, 2 from A to B, and yet 20 from A to the goal.
- All consistent heuristics are admissible. The opposite is not necessarily true.
- Consistent heuristics make our algorithm faster, because we don't need to revisit nodes (in Graph Search).

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)

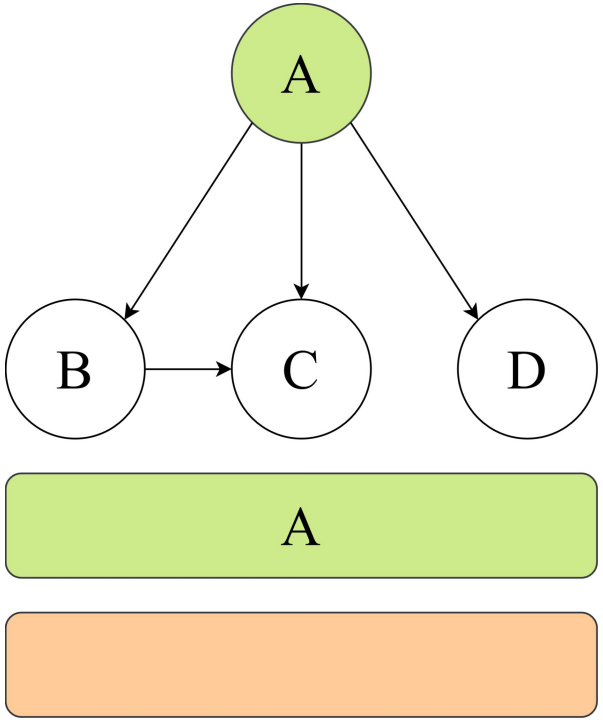
Project 1 : Questions 1-4

```
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) (Pseudocode)
- Different frontiers for each algorithm:
  - Stack (DFS)
  - Queue (BFS)
  - PriorityQueue (UCS, A*)
- Expanded should be a **Set**
- **Keep in mind**: The autograder expects a specific number of nodes to be expanded.
  - Controlled by problem.getSuccessors, don't forget print statements

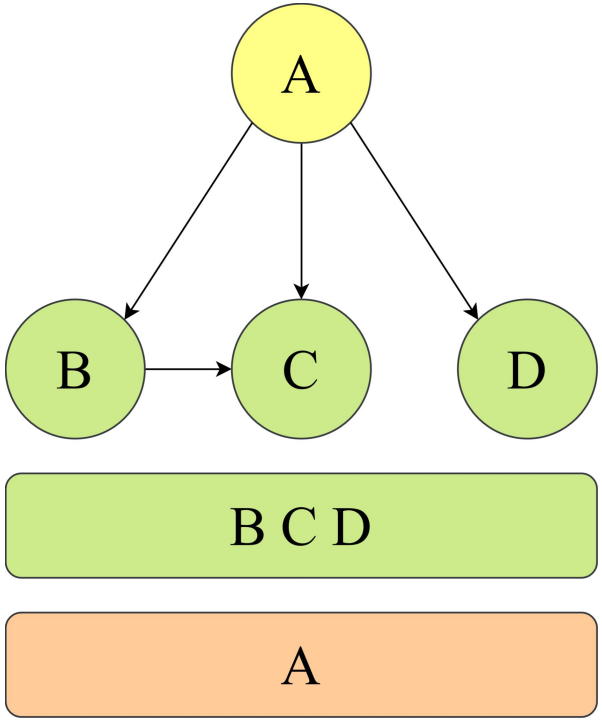
Project 1 : Questions 1-4 - DFS

```
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
```



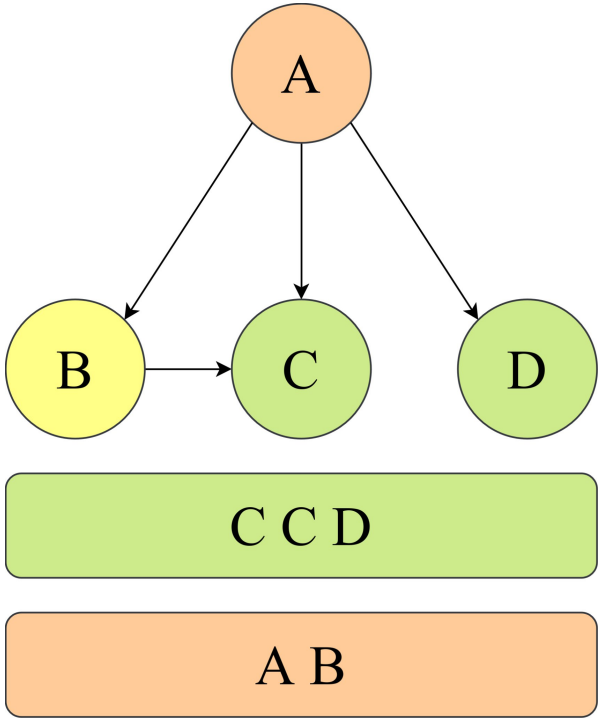
Project 1 : Questions 1-4 - DFS

```
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
```



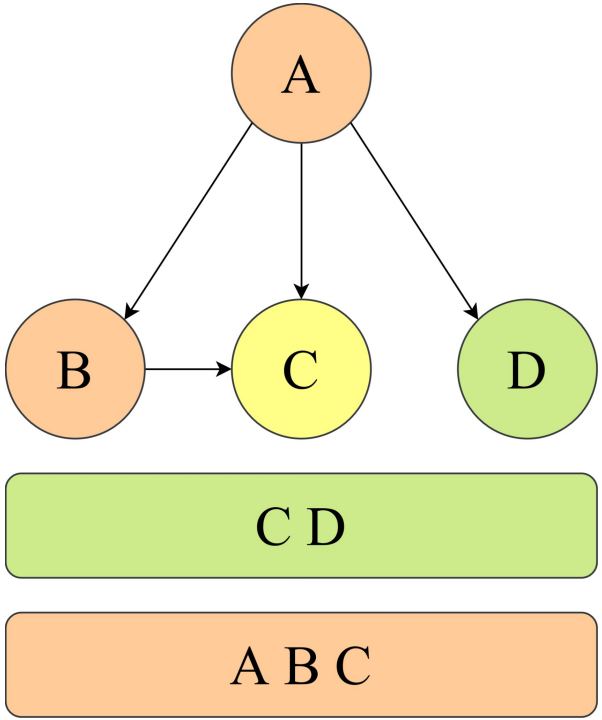
Project 1 : Questions 1-4 - DFS

```
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
```



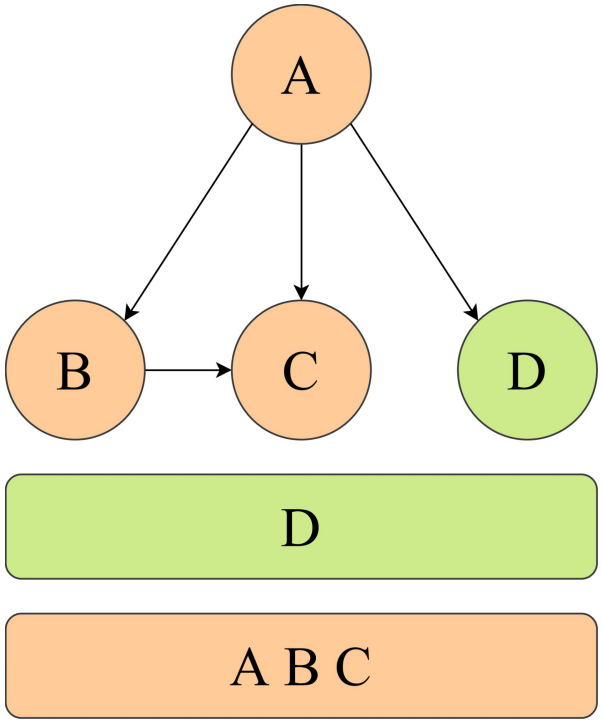
Project 1 : Questions 1-4 - DFS

```
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
```



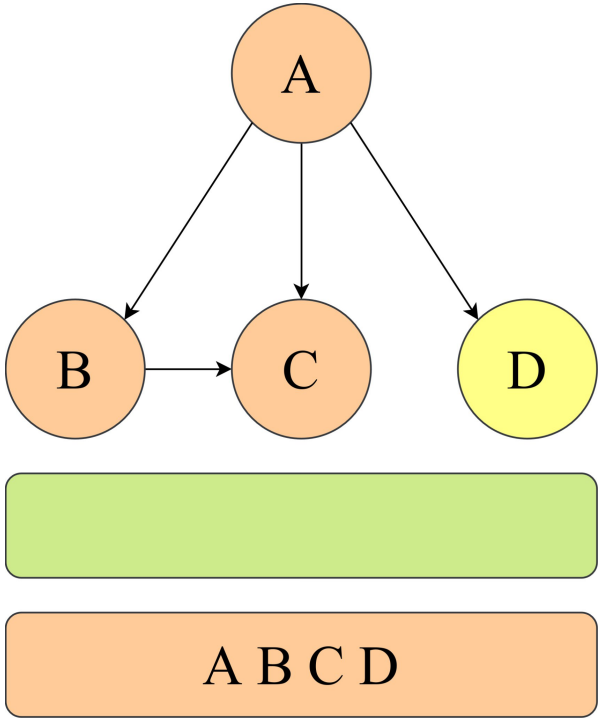
Project 1 : Questions 1-4 - DFS

```
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
```



Project 1 : Questions 1-4 - DFS

```
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
```



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 **admissible** 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

➢ Note: if you encounter problems, make sure that your solution to Question 5 does not have any subtle problems

Project 1: Question 7

- **Goal:** Write a non-trivial, non-decreasing **admissible** heuristic to eat all the food in as few steps as possible. In other words, you are asked to write a heuristic that estimates as closely as possible the number of steps that Pacman must take to eat all the food.

➢ You can get the full grade in around 10 lines of code.
➢ **Note:** The use of mazeDistance as a heuristic is forbidden! This is a trivial heuristic. You can use it as part of your solution, but not as your solution.

➢ 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