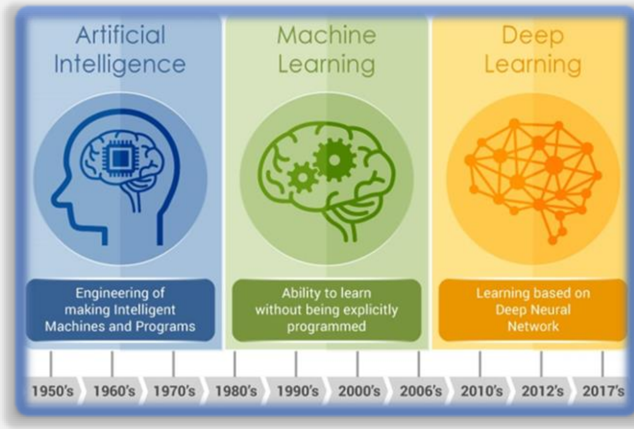
CENG 3511 Artificial Intelligence

Week 3
Advance Search Techniques

Instructor
Bekir Taner Dinçer
Teaching Assistant
Selahattin Aksoy



MUĞLA SITKI KOÇMAN UNIVERSITY
COMPUTER ENGINEERING
2024 – FALL

1

Outline

- **Uninformed Search Strategies**
  - Breadth-First Search (BFS),
  - Depth-First Search (DFS),
  - Depth-Limited Search (DLS),
  - Iterative Deepening Depth-First Search (IDDFS)
  - Uniform Cost Searches (UCS)
- **Informed Search Strategies**
  - Greedy Search,
  - A* Search
- **Local Search Algorithms**
  - Hill Climbing,
  - Simulated Annealing,
  - Genetic Algorithms

2

Overview of Search Algorithms in AI

Search algorithms are providing a systematic way to **explore problem spaces (search trees)** and **find optimal or near-optimal solutions**.

Optimal Solution
refers to
the **best possible solution** to a problem,
typically within **constraints** or **limitations**.

Search Problem Revisited

Key Components:

State Space: The set of all possible states that can be reached from the initial state.

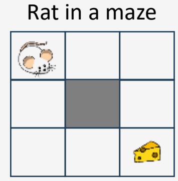
Actions: The possible operations that can be performed to transition from one state to another.

Initial State: The starting point of the search.

Goal State: The desired end state that signifies a solution.

Solution: A sequence of actions that leads from the initial state to the goal state.

Rat in a maze



Environment:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

State Space: { all cells except for (1,1) }

Actions: "UDLR"

- U(p), D(own), L(eft), R(ight)

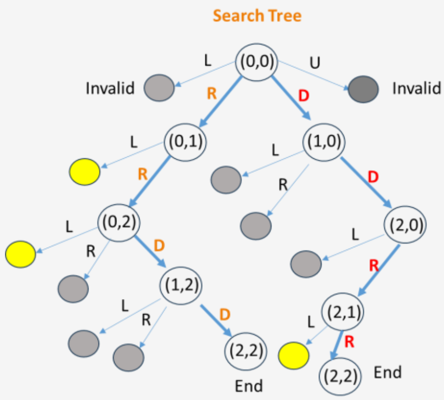
Initial State: (0,0)

Goal State: (2,2)

Solution: "RRDD" or "DDRR"

Search Problem Revisited

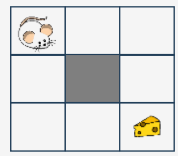
All possible solutions can be represented as



All Possible

Solutions

Rat in a maze



Environment:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

State Space: { all cells except for (1,1) }

Actions: “UDLR”

•U(p), D(own), L(eft), R(ight)

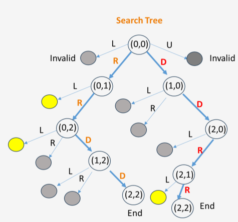
Initial State: (0,0)

Goal State: (2,2)

Solution: “RRDD” or “DDRR”

5

Search Problem Revisited



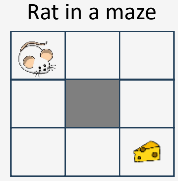
+

Formal Problem Definition:

- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state *s*
- **Cost**(s,a): cost of action *a*, given state *s*
- **Next**(s,a): next state to *s* given action *a*
- **IsGoal**(s): is the state *s* the target state

drive

Rat in a maze



Environment:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

State Space: { all cells except for (1,1) }

Actions: “UDLR”

•U(p), D(own), L(eft), R(ight)

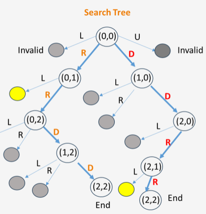
Initial State: (0,0)

Goal State: (2,2)

Solution: “RRDD” or “DDRR”

6

Search Problem Revisited



+

Formal Problem Definition:

- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state *s*
- **Cost**(s,a): cost of action *a*, given state *s*
- **Next**(s,a): next state to *s* given action *a*
- **IsGoal**(s): is the state *s* the target state

Given a Search Tree and a Problem Definition,

a **Search Problem** turns into
a problem of
finding the best **Search Strategy**

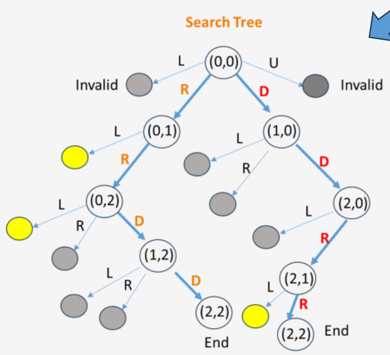
to **systematically explore the search tree** and
find optimal or near-optimal solutions

MSKU CENG | CENG-3511 Artificial Intelligence

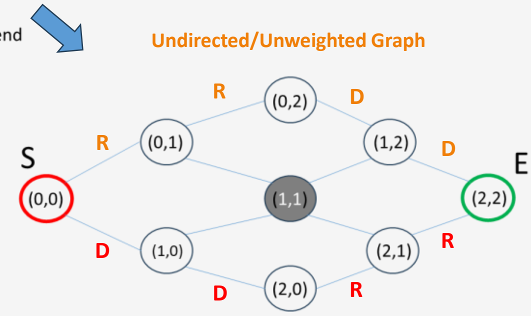
7

Search Problem Revisited

start				
start	(0,0)	(0,1)	(0,2)	
	(1,0)	(1,1)	(1,2)	
	(2,0)	(2,1)	(2,2)	end



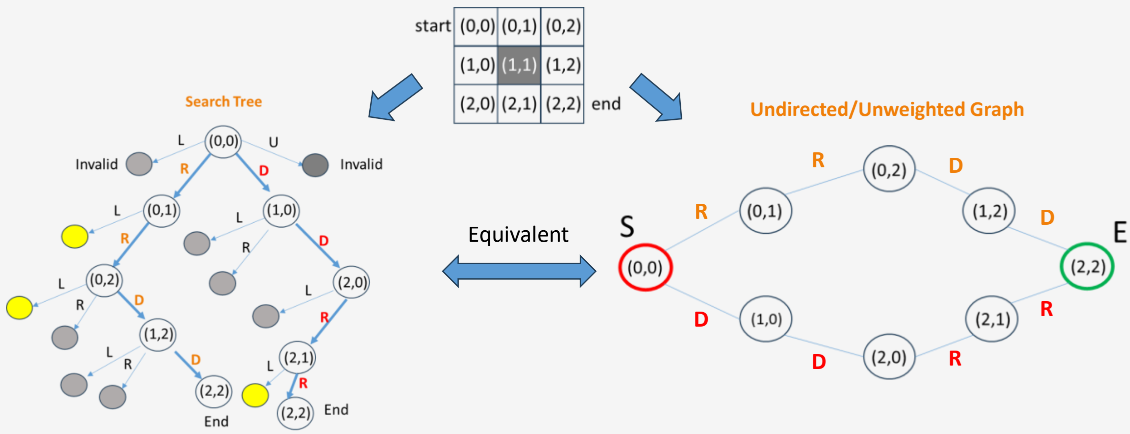
Equivalent



MSKU CENG | CENG-3511 Artificial Intelligence

8

Search Problem Revisited



MSKU CENG | CENG-3511 Artificial Intelligence

9

Uninformed Search Strategies

BFS, DFS, Greedy, A* and Uniform Cost Search (UCS)

MSKU CENG | CENG-3511 Artificial Intelligence

14

Uninformed Search Strategies

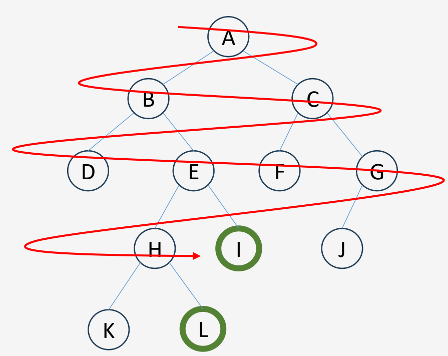
Search strategies that
explore the search space
without any domain-specific knowledge.

Blind search

Key Strategies
Breadth-First Search (BFS)
Depth-First Search (DFS)
Depth-Limited Search (DLS)
Iterative Deepening Depth-First Search (IDDFS)
Uniform Cost Search (UCS)

Breadth First Search

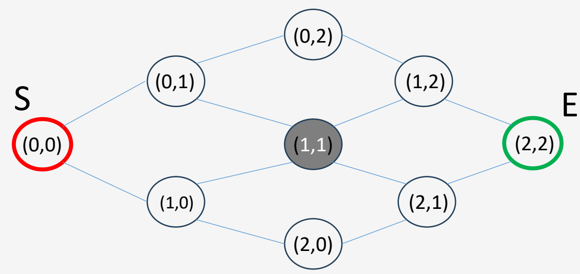
- A **breadth-first** search (BFS) explores neighbor nodes starting from the root node.
- For the example, after exploring **A**, then **B**, then **C**, the search continuous with **D**, **E**, **F**, **G**, **H**, and **I**.
- Nodes are explored as A B C D E F G H I J K L
- Here, the target state **I** will be found before the target state **L**
- **Best for**: Finding the **shortest path in an unweighted graph**; when all step costs are equal.



Maze Problem: Graph Representation

Rat in a maze

start	(0,0)	(0,1)	(0,2)	
	(1,0)	(1,1)	(1,2)	
	(2,0)	(2,1)	(2,2)	end



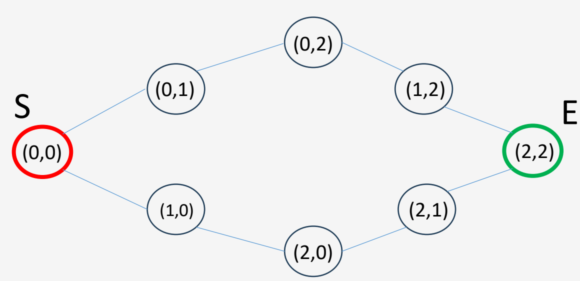
17

Maze Problem: Revised Graph

Search Space for Maze

Rat in a maze

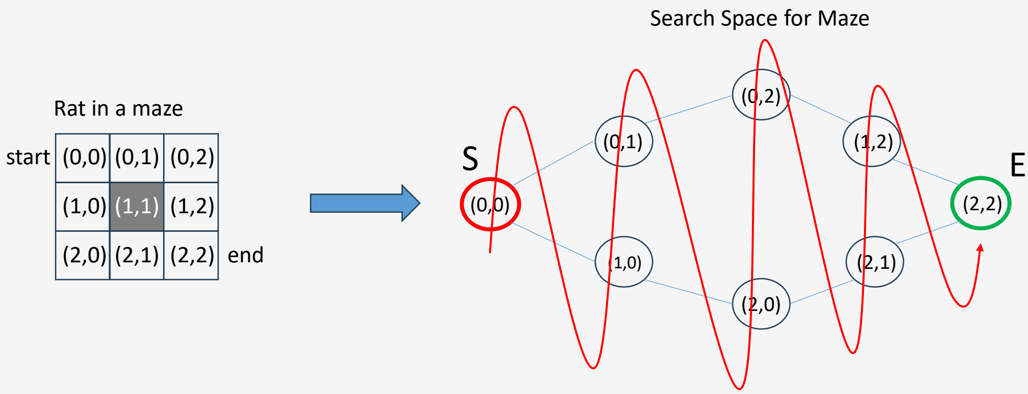
start	(0,0)	(0,1)	(0,2)	
	(1,0)	(1,1)	(1,2)	
	(2,0)	(2,1)	(2,2)	end



MSKU CENG | CENG-3511 Artificial Intelligence

18

Breadth-First Search



Breadth First Search

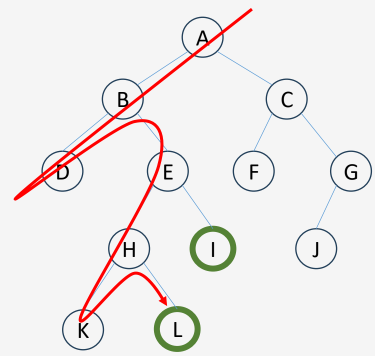
```
BFS(start_node):
    1. Initialize an empty queue, `Q`
    2. Initialize an empty set, `visited`

    3. Enqueue the `start_node` into the queue `Q`
    4. Mark `start_node` as visited (add it to the `visited` set)

    5. While Q is not empty:
        6.1. Dequeue a node `current_node` from the front of the queue
        6.2. Print current_node to show the traversal

        7. For each neighbor `neighbor` of `current_node`:
            8. If `neighbor` has not been visited:
                9. Mark `neighbor` as visited (add it to the `visited` set)
               10. Enqueue `neighbor` into the queue `Q`

    11. End the search
```

FIFO

History

Depth First Search

- A **Depth-first** search (DFS) explores a path all the way down to a leaf before **backtracking** and exploring another path.
- For the example, after exploring **A**, then **B**, then **D**, the search backtracks and tries another path from **B**.
- Nodes are explored as A B D E H K L I C F G J
- Here, **L** will be found before **I**
- **Best for**: exploring entire state spaces where **memory is a concern**.

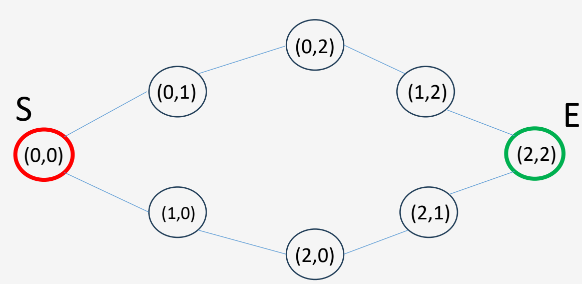


Depth-First Search

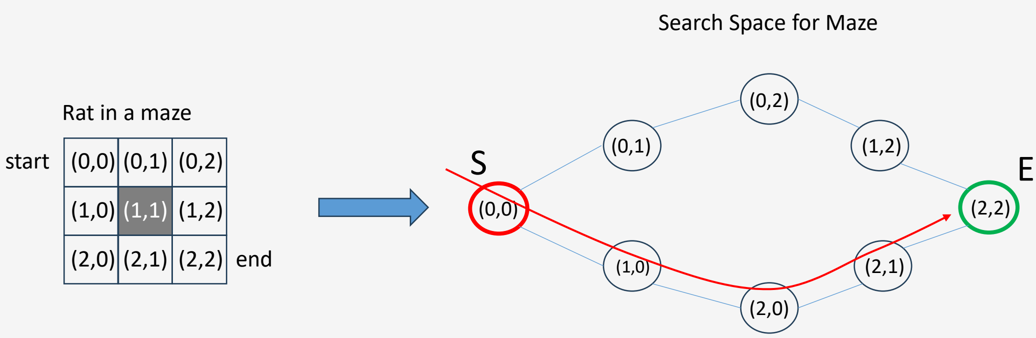
Rat in a maze

start	(0,0)	(0,1)	(0,2)	
	(1,0)	(1,1)	(1,2)	
	(2,0)	(2,1)	(2,2)	end

Search Space for Maze



Depth-First Search

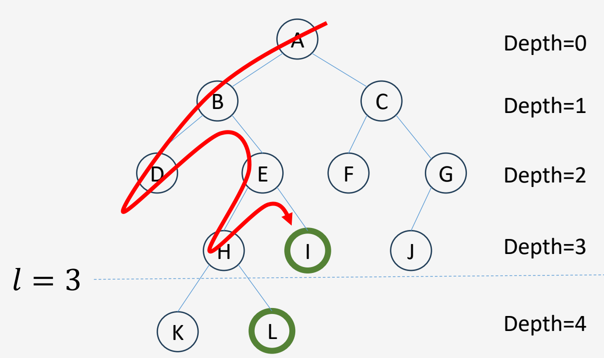


CENG-3511 Artificial Intelligence

23

Depth-Limited Search

- A **Depth-limited** search (DLS) is basically a **depth-first search** with limited depth l.
- This prevents infinite descent or loops in infinite-depth spaces or cycles.
- For $l = 3$, after exploring **A**, then **B**, then **D**, the search backtracks and tries another path from **B**. After **E**, then **H**, since depth limit is reached, it backtracks and tries another path from **E**, and find **I**.
- Nodes are explored as A B D E H I C F G J
- Here, **K** and **L** will not be explored
- **Best for** scenarios where a reasonable depth limit can be estimated, such as solving puzzles with depth constraints.

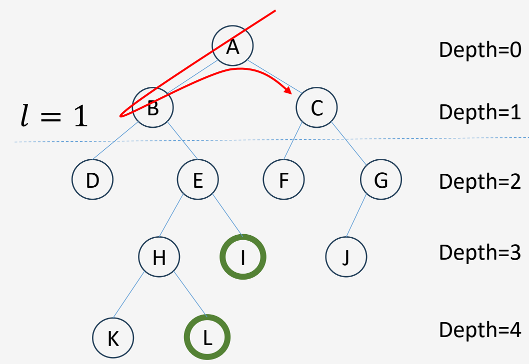


CENG-3511 Artificial Intelligence

24

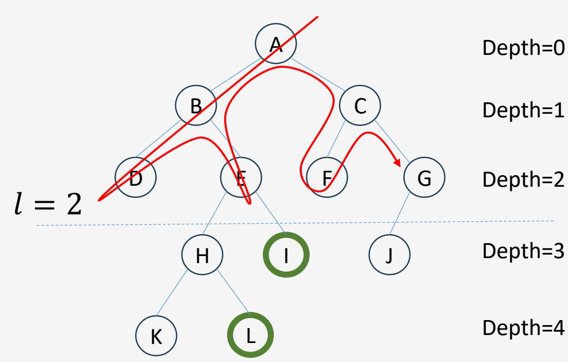
Iterative Deepening DFS (IDDFS)

- An **iterative deepening DFS** search (IDDFS) applies **depth-first search** with increasing depth limits l.



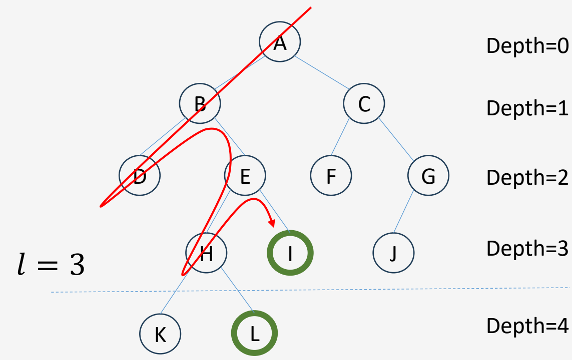
Iterative Deepening DFS (IDDFS)

- Combines the advantages of **BFS** and **DFS**.



Iterative Deepening DFS (IDDFS)

- **Best for** scenarios where **memory is limited**, and the **solution is unknown but exists** at some depth.



Comparison of Uninformed Search Strategies

Strategy	Space Complexity	Time Complexity	Pros	Cons
Breadth-First Search (BFS)	O(b^d)	O(b^d)	- Complete (will always find a solution if one exists)	- High memory usage due to storing all nodes at the current depth
			- Optimal if step costs are uniform	- Exponential time and space complexity
Depth-First Search (DFS)	O(bm)	O(b^m)	- Low memory usage (only needs to store nodes on the current path)	- Not optimal (may find suboptimal solutions)
			- Can be more efficient than BFS in certain cases (e.g., when the solution is deep)	- May get stuck in deep or infinite branches
Depth-Limited Search	O(bl)	O(b^l)	- Prevents infinite loops in DFS	- Requires a predefined depth limit; if limit is too low, it may not find the solution
			- More memory efficient than BFS	- May miss shallow solutions if the limit is too high
Iterative Deepening DFS (IDDFS)	O(bd)	O(b^d)	- Combines the advantages of BFS (completeness) and DFS (low memory)	- Redundant exploration of nodes from repeated depth-limited searches
			- Memory efficient and complete	- Can be slower due to multiple DFS passes through nodes at increasing depth

Notation:

b: Branching factor (number of successors at each node).

d: Depth of the shallowest solution.

m: Maximum depth of the search space (can be infinite in some cases).

l : Depth limit (used in Depth-Limited Search).

***optimal** refers to the ability of the algorithm to find the **best** solution, often the one with the **lowest cost** (or shortest path), when multiple solutions exist.

MSKU CENG | CENG-3511 Artificial Intelligence

Comparison of Uninformed Search Strategies

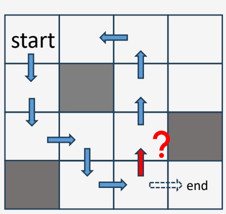
Strategy	Best Suited For
Breadth-First Search (BFS)	Finding the shortest path in an unweighted graph; when all step costs are equal.
	Useful in shallow search trees or when the solution is close to the root.
Depth-First Search (DFS)	Searching deeper solutions or paths; exploring entire state spaces where memory is a concern.
	Suitable when the solution is likely far from the root or in deep problem spaces.
Depth-Limited Search	Best for scenarios where a reasonable depth limit can be estimated, such as solving puzzles with depth constraints.
	Useful when a specific depth is known or can be assumed to contain the solution.
Iterative Deepening DFS (IDDFS)	Effective for scenarios where memory is limited, and the solution is unknown but exists at some depth.
	Best when you want completeness and optimality like BFS but with less memory usage. Suitable for large depth-limited problem spaces, such as game tree searches.

29

Homework: Solution

CENG3511-AI-Lab2-ModelBasedReflexAgent-Solution.ipynb

4x4 Maze



Obstacles/Walls:
{(3,0), (1,1), (2,3)}

The agent implementation fails to find a solution

Why? **Fix it as a Homework**

```
# Define the maze parameters
environment = [ [1,1,1,1], [1,0,1,1], [1,1,1,0], [0,1,1,1] ]
initial_state = (0, 0)
goal_state = (3, 3)

# Create the agent
agent = RatRobot(environment, initial_state, goal_state)

# Run the agent and print the actions
actions = agent.run()
print("Actions taken:", actions)
```

```
No action is possible! !!! Robot Stucked !!!
Actions taken: ['down', 'down', 'right', 'down', 'right', 'up', 'up', 'up', 'left']
```

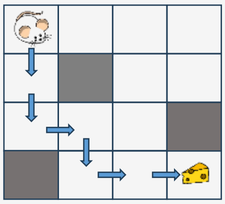
MSKU CENG | CENG-3511 Artificial Intelligence

30

Solution using BFS

CENG3511-AI-Lab2-ModelBasedReflexAgent-Solution.ipynb

4x4 Maze

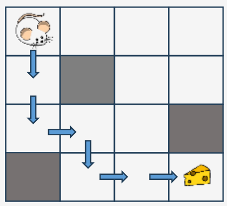


Obstacles/Walls:
{(3,0), (1,1), (2,3)}

```
Maze Environment:
[1, 1, 1, 1]
[1, 0, 1, 1]
[1, 1, 1, 0]
[0, 1, 1, 1]

Path found: [(1, 0), (2, 0), (2, 1), (3, 1), (3, 2), (3, 3)]
Actions : ['DOWN', 'DOWN', 'RIGHT', 'DOWN', 'RIGHT', 'RIGHT']
```


Uniform Cost Search

Cost of an Action



Solution based on BFS

Path found: [(1, 0), (2, 0), (2, 1), (3, 1), (3, 2), (3, 3)]

Actions : ['Down', 'Down', 'Right', 'Down', 'Right', 'Right']

It is assumed

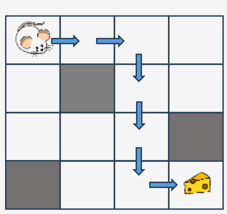
Cost(s,a) = 0, for all a

⇒ Cost(*path*) = Cost((0,0), down) + Cost((1,0), down) + Cost((2,0), right) + …
⇒ = 0 + 0 + 0 + …

Formal Problem Definition:

- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state *s*
- **Cost**(s,a): cost of action *a*, given state *s*
- **Next**(s,a): next state to *s* given action *a*
- **IsGoal**(s): is the state *s* the target state

Cost of an Action



Another solution

Path found: [(0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3)]

Actions : ['Right', 'Right ', 'Down', 'Down', 'Down', 'Right']

It is assumed

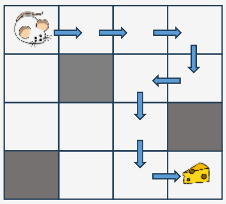
Cost(s,a) = 0, for all a

⇒ Cost(*path*) = Cost((0,0), right) + Cost((0,1), right) + Cost((0,2), down) + …
⇒ = 0 + 0 + 0 + …

Formal Problem Definition:

- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state *s*
- **Cost**(s,a): cost of action *a*, given state *s*
- **Next**(s,a): next state to *s* given action *a*
- **IsGoal**(s): is the state *s* the target state

Cost of an Action



Another solution possible

Path found: [(0, 1), (0, 2), (0, 3), (1, 3), (1, 2), (2, 2), (3, 2), (3, 3)]

Actions : ['Right', 'Right ', 'Right ', 'Down', 'Left', 'Down', 'Down', 'Right']

It is assumed

Cost(s,a) = 0, for all a

⇒ Cost($path$) = Cost((0,0), right) + Cost((0,1), right) + Cost((0,2), down) + …

⇒ = 0 + 0 + 0 + …

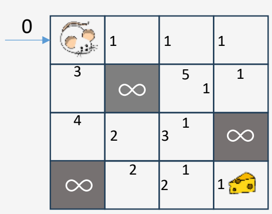
Formal Problem Definition:

- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state s
- **Cost**(s,a): cost of action a, given state s
- **Next**(s,a): next state to s given action a
- **IsGoal**(s): is the state s the target state

Since the cost of any action is 0,

there is no precedence among possible solutions

Cost of an Action

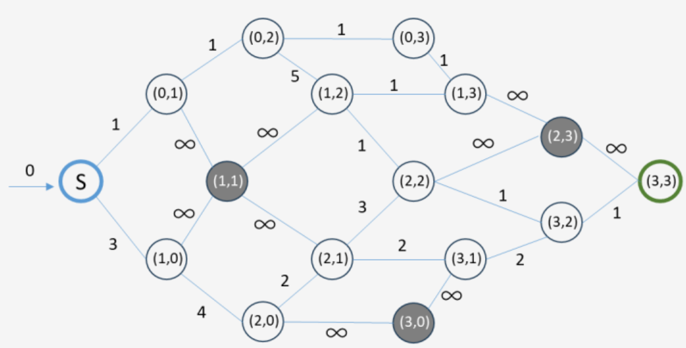


Formal Problem Definition:

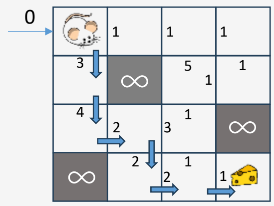
- s_{start} : starting state
- s_{goal} : goal state
- **Actions(s)**: possible actions given state s
- **Cost**(s,a): cost of action a, given state s
- **Next**(s,a): next state to s given action a
- **IsGoal**(s): is the state s the target state

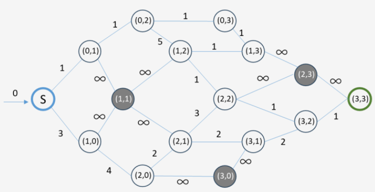
What if

Cost(s,a) > 0, for all a



Cost of an Action





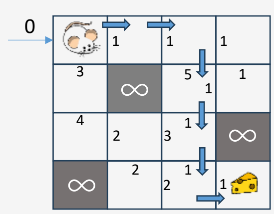
Solution based on BFS

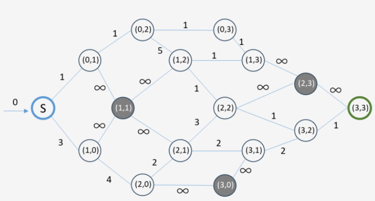
Actions : ['Down', 'Down', 'Right', 'Down', 'Right', 'Right']

Cost(*path*) = 3 + 4 + 2 + 2 + 2 + 1 = 14

38

Cost of an Action





Solution based on BFS

Actions : ['Down', 'Down', 'Right', 'Down', 'Right', 'Right']

Cost(*path*) = 3 + 4 + 2 + 2 + 2 + 1 = 14

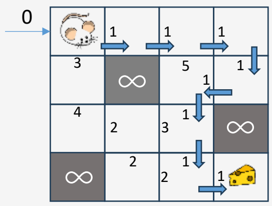
Solution 2

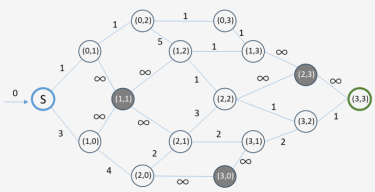
Actions : ['Right', 'Right ', 'Down', 'Down', 'Down', 'Right']

Cost(*path*) = 1 + 1 + 5 + 1 + 1 + 1 = 10

39

Cost of an Action





<u>Solution based on BFS</u>

Actions : ['Down', 'Down', 'Right', 'Down', 'Right', 'Right']

Cost(*path*) = 3 + 4 + 2 + 2 + 2 + 1 = 14

<u>Solution 2</u>

Actions : ['Right', 'Right ', 'Down', 'Down', 'Down', 'Right']

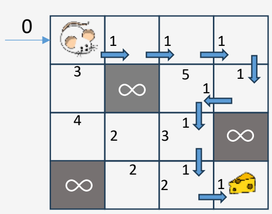
Cost(*path*) = 1 + 1 + 5 + 1 + 1 + 1 = 10

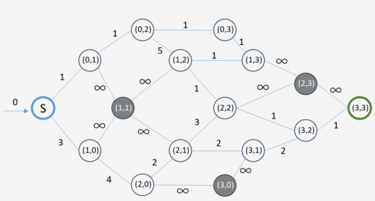
<u>Solution 3</u>

Actions : ['Right', 'Right ', 'Right ', 'Down', 'Left', 'Down', 'Down', 'Right']

Cost(*path*) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8

Cost of an Action





<u>Solution based on BFS</u>

Actions : ['Down', 'Down', 'Right', 'Down', 'Right', 'Right']

Cost(*path*) = 3 + 4 + 2 + 2 + 2 + 1 = 14

<u>Solution 2</u>

Actions : ['Right', 'Right ', 'Down', 'Down', 'Down', 'Right']

Cost(*path*) = 1 + 1 + 5 + 1 + 1 + 1 = 10

Minimum Cost

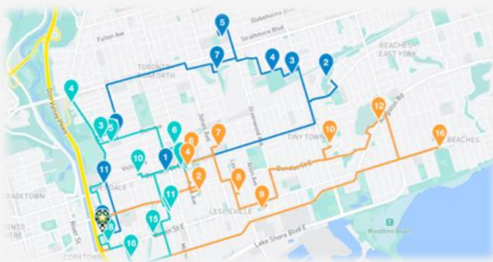
<u>Solution 3</u>

Actions : ['Right', 'Right ', 'Right ', 'Down', 'Left', 'Down', 'Down', 'Right']

Cost(*path*) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8

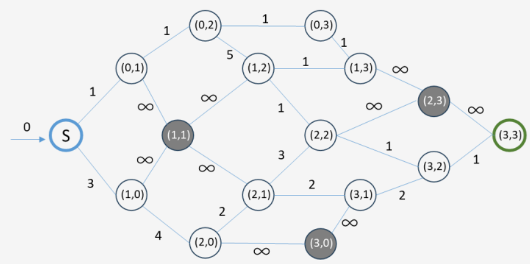
Cost of an Action

Transportation/Navigation Problems



- **Travelling Salesman Problem** (TSP)
  - logistics and planning problems
  - the goal is to visit a set of locations in the shortest possible time or distance.
  - **Nodes** represent Locations (Cities or Houses); **Edges** represent roads or streets

- **Chinese Postman Problem** (CPP)
  - mail delivery routes, garbage collection routes
  - the goal is to minimize the travel distance while covering all streets
  - **Edges** represent streets; **Nodes** represent "cross-roads" or "road branching"



Uniform Cost Search

Approach

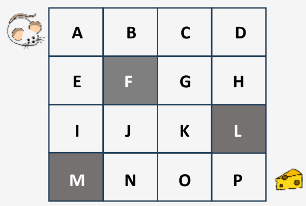
- UCS expands nodes based on **the lowest cumulative path cost** from the start node.
- Uses **a priority queue** (or min-heap) to keep track of nodes to be expanded, where **nodes are prioritized based on their path cost**.

Properties

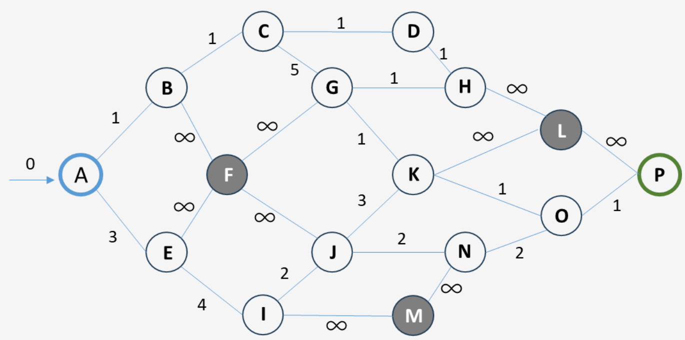
- If **Cost**(s, a) = $c > 0$, for all state s and action a,
  - It finds the shortest path in a weighted graph, i.e., **BFS**.
- if the path costs are non-negative ($c > 0$) and search space is finite
  - UCS is
    - **optimal**: finds the shortest path in terms of cost, and
    - **complete**: finds a solution if one exists.
- It ensures that the path with the minimum total cost is found.

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb



Graph Representation of the Maze Problem



MSKU CENG | CENG-3511 Artificial Intelligence

Uniform Cost Search

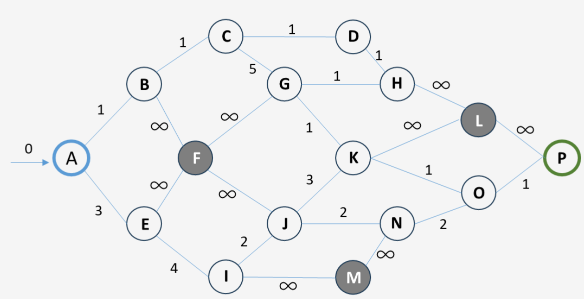
CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(0, [A])]

Visited = {}



MSKU CENG | CENG-3511 Artificial Intelligence

Uniform Cost Search

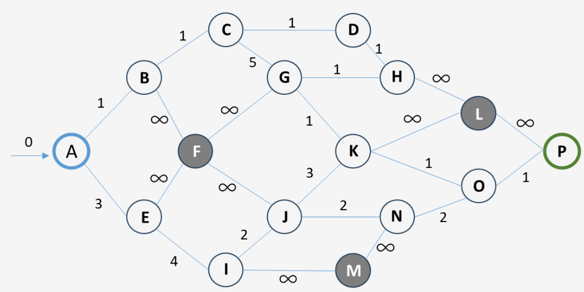
CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [ ]   cost = 0, path = [A]

Visited = {}



MSKU CENG | CENG-3511 Artificial Intelligence

46

Uniform Cost Search

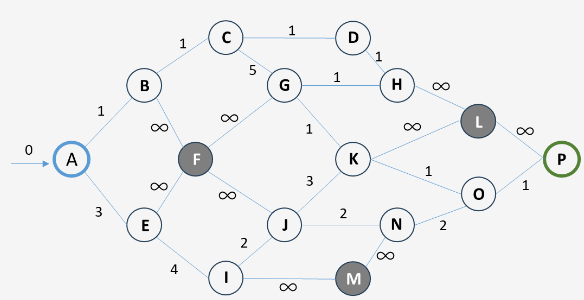
CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [ ]   cost = 0, path = [A]

Visited = {}   current_node = path[-1] = A



MSKU CENG | CENG-3511 Artificial Intelligence

47

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node: If A == P
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

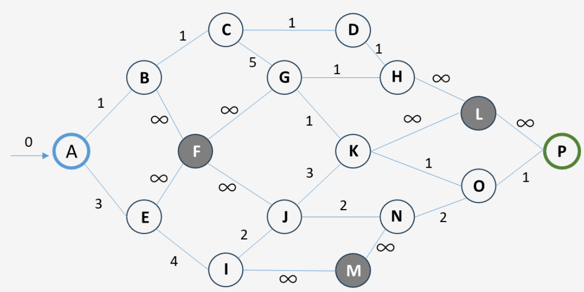
Goal = P

PQ = [ ]

Visited = {}

cost = 0, path = [A]

current_node = path[-1] = A



48

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`: If A in Visited
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

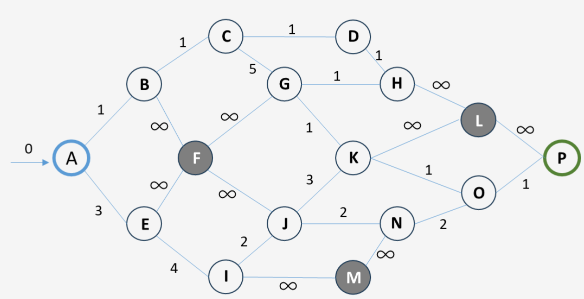
Goal = P

PQ = [ ]

Visited = {}

cost = 0, path = [A]

current_node = path[-1] = A



49

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

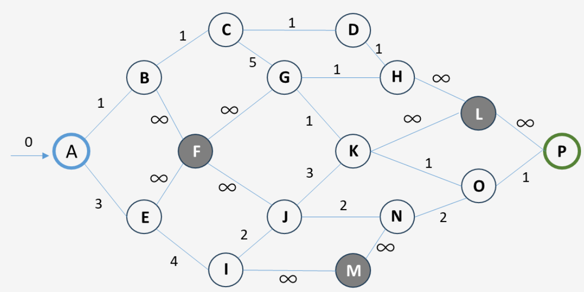
Goal = **P**

PQ = [ ]

Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A



Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = **P**

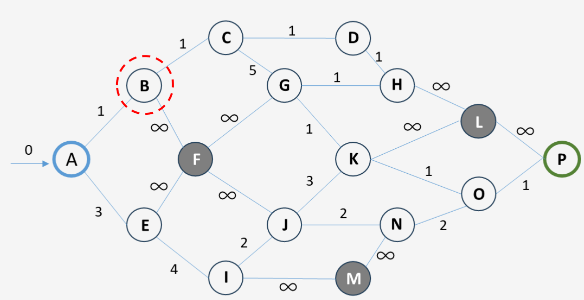
PQ = [ ]

Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A

new_cost(B) = 0 + 1 = 1



Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [ ]

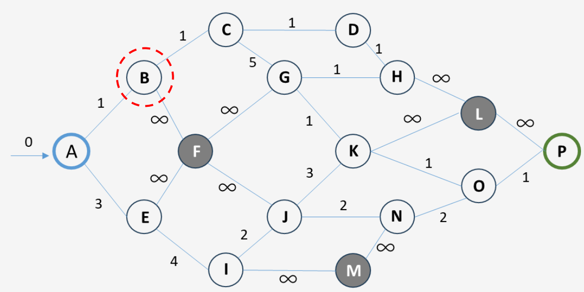
Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A

new_cost = 0 + 1 = 1

new_path = [A, B]



MSKU CENG | CENG-3511 Artificial Intelligence

52

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(1, [A, B]]

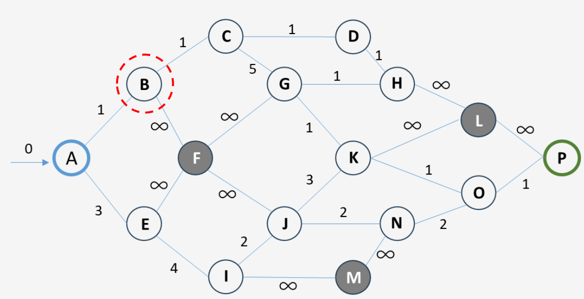
Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A

new_cost = 0 + 1 = 1

new_path = [A, B]



MSKU CENG | CENG-3511 Artificial Intelligence

53

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

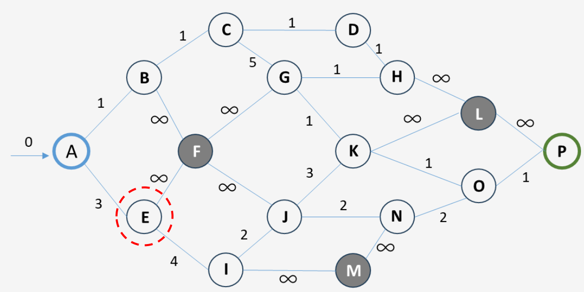
Goal = P

PQ = [(1, [A, B]]

Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A



CENG-3511 Artificial Intelligence

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(1, [A, B]),
(3, [A, E])]

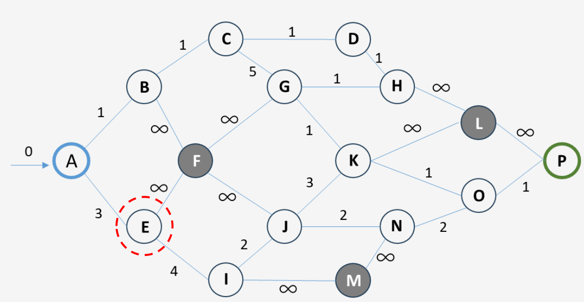
Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A

new_cost = 0 + 1 = 3

new_path = [A, E]



CENG-3511 Artificial Intelligence

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(1, [A, B]),
(3, [A, E])]

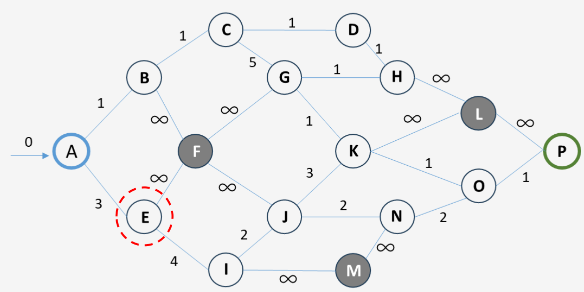
Visited = {A}

cost = 0, path = [A]

current_node = path[-1] = A

new_cost = 0 + 1 = 3

new_path = [A, E]



56

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E])]

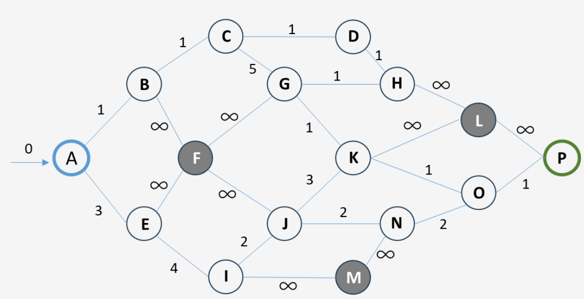
Visited = {A}

cost = 1, path = [A, B]

current_node = path[-1] = A

new_cost =

new_path =



57

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E])]

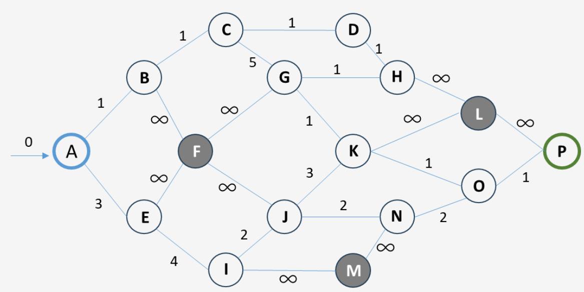
cost = 1, path = [A, B]

current_node = path[-1] = B

Visited = {A}

new_cost =

new_path =



MSKU CENG | CENG-3511 Artificial Intelligence

58

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E])]

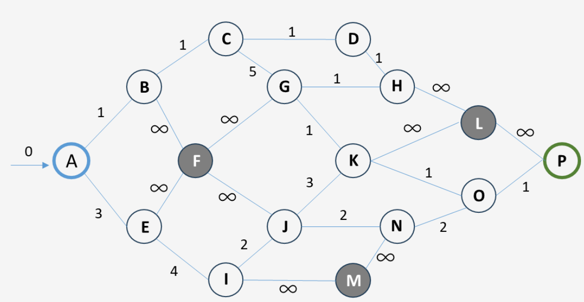
cost = 1, path = [A, B]

current_node = path[-1] = B

Visited = {A, B}

new_cost =

new_path =



MSKU CENG | CENG-3511 Artificial Intelligence

59

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = **P**

PQ = [(3, [A, E])]

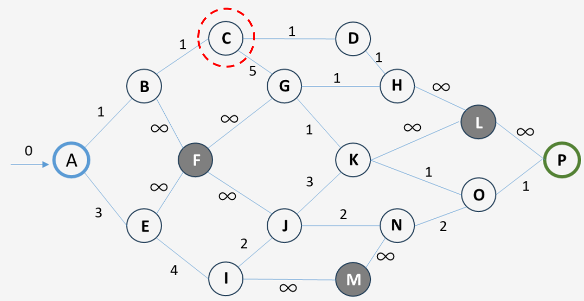
cost = 1, path = [A, B]

current_node = path[-1] = B

Visited = {A, B}

new_cost = 1 + 1 = 2

new_path =



60

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
              (new_cost = current cost + neighbor's cost).
            - Create a new path
              by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = **P**

PQ = [(3, [A, E])]

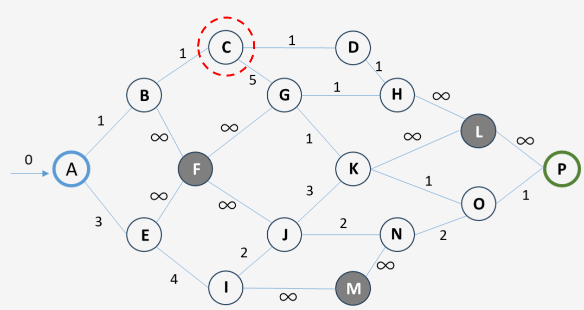
cost = 1, path = [A, B]

current_node = path[-1] = B

Visited = {A, B}

new_cost = 1 + 1 = 2

new_path = [A, B, C]



61

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E]),
(2, [A,B,C])]

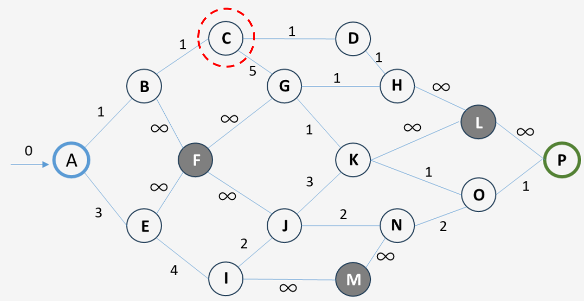
Visited = {A, B}

cost = 1, path = [A, B]

current_node = path[-1] = B

new_cost = 1 + 1 = 2

new_path = [A, B, C]



CENG-3511 Artificial Intelligence

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E]),
(2, [A,B,C])]

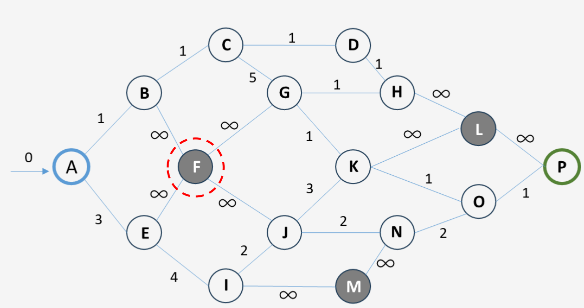
Visited = {A, B}

cost = 1, path = [A, B]

current_node = path[-1] = B

new_cost = 1 + inf = inf

new_path = [A, B, C]



CENG-3511 Artificial Intelligence

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(3, [A, E]),
(2, [A,B,C]),
(inf, [A,B,F])]

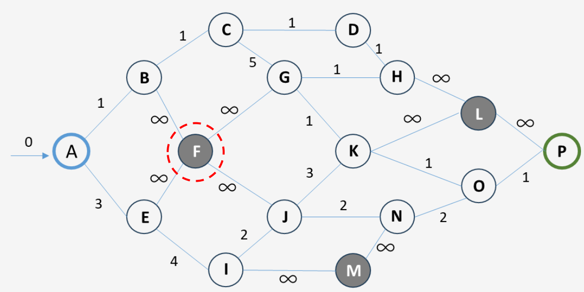
Visited = {A, B}

cost = 1, path = [A, B]

current_node = path[-1] = B

new_cost = 1 + inf = inf

new_path = [A, B, F]



Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb


```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(2, [A,B,C]),
(3, [A, E]),
(inf, [A,B,F])]

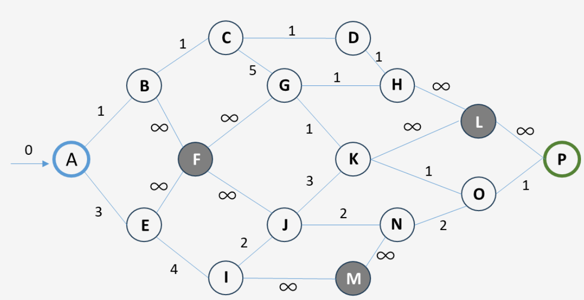
Visited = {A, B}

cost = 1, path = [A, B]

current_node = path[-1] = B

new_cost = 1 + inf = inf

new_path = [A, B, F]



Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
        i. Return the path and the cumulative cost.

    d. If current_node is not in `visited`:
        i. Add current_node to `visited`.
        ii. For each neighbor of current_node:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Goal = P

PQ = [(2, [A,B,C]),
(3, [A, E]),
(inf, [A,B,F])]

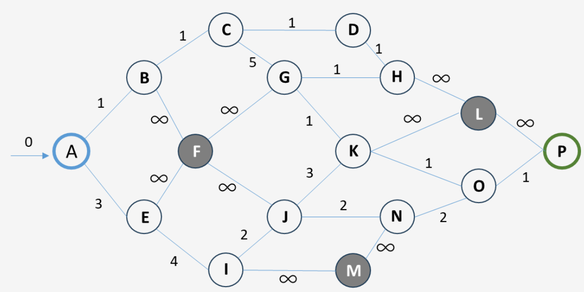
Visited = {A, B}

cost = 2, path = [A, B, C]

current_node = path[-1] =

new_cost =

new_path =



MSKU CENG | CENG-3511 Artificial Intelligence

66

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

```
1. Initialize a queue with (cost 0, path [start node]).
2. Create an empty set `visited` to track visited nodes.
3. While the queue is not empty:
    a. Pop element (cost, path) from the queue.
    b. Set current_node to the last node in the path.

    c. If current_node is the goal node:
          Return the path and the cumulative cost.

    d. If      ent   de is   ot in `   sited`:
        i. Add   ur   ent_no   to `vi   ited`.
              ea   nei   bor of c   ent   ode:
            - Calculate the new cumulative cost
                (new_cost = current cost + neighbor's cost).
            - Create a new path
                by appending the neighbor to the current path.
            - Add (new_cost, new_path) to the queue.

    e. Sort the queue based on the cost (lowest cost first).

4. If the goal node is not found after the queue is empty,
    return None (no path found).
```

Soon

Goal = P

PQ = [(2, [A,B,C]),
(3, [A, E]),
(inf, [A,B,F])]

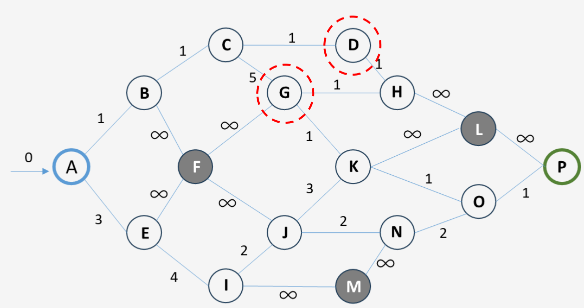
Visited = {A, B}

cost = 2, path = [A, B, C]

current_node = path[-1] = C

new_cost =

new_path =



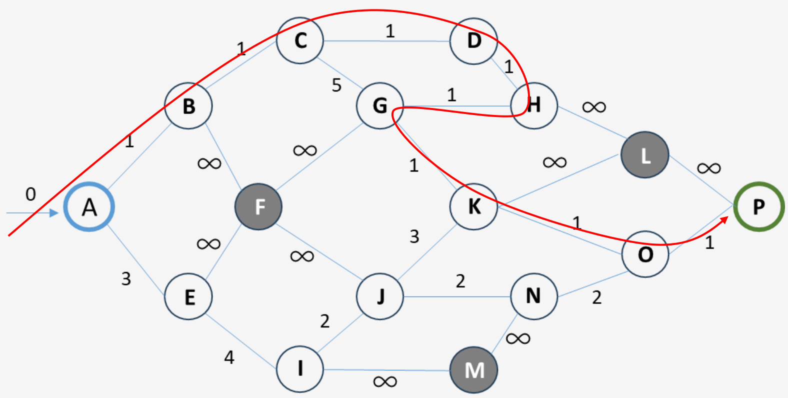
MSKU CENG | CENG-3511 Artificial Intelligence

67

Uniform Cost Search

CENG3511-AI-Lab2-UniformCostSearch-UCS.ipynb

Path: ['A', 'B', 'C', 'D', 'H', 'G', 'K', 'O', 'P']
Total Cost: 8



Informed Search Strategies

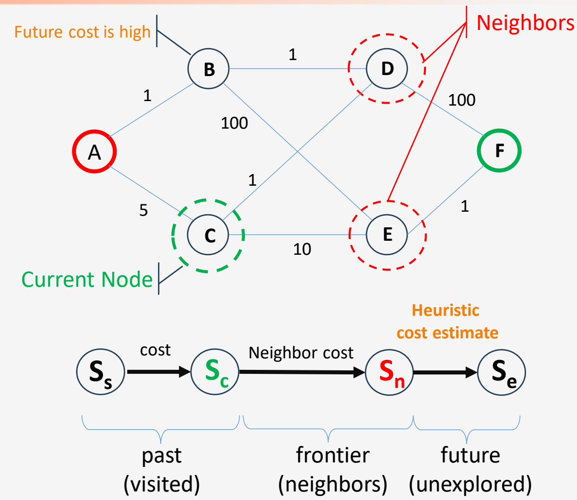
Greedy Search, A* Search.

Informed Search Strategies

Definition:
Informed search strategies use additional information (**heuristics**) to make the search process more efficient, **reducing the number of nodes explored**.

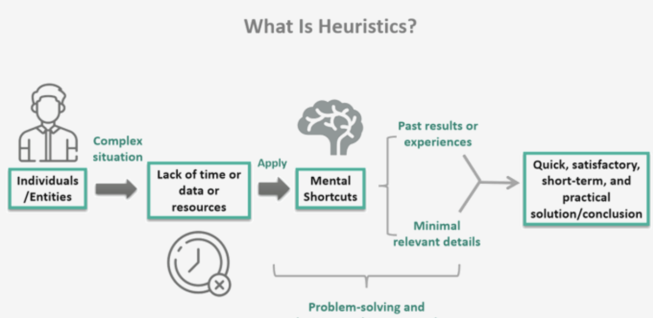
Key Insight:
Heuristics guide the search towards the goal by evaluating which nodes are "more promising" based on a **heuristic function.**

Comparison with Uninformed Search:
Unlike uninformed search, which explores blindly, informed search uses heuristics to focus exploration.



What is Heuristics in general?

Heuristics are **mental shortcuts for solving problems in a quick way that delivers a result that is sufficient enough to be useful given time constraints.**



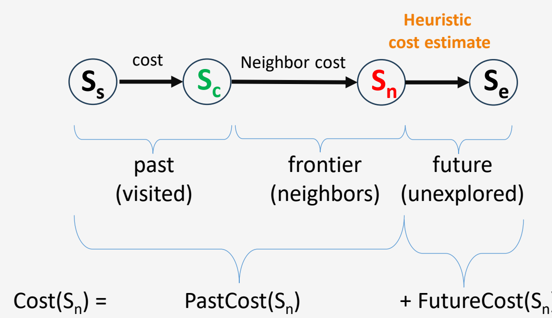
Heuristic Functions

Role of Heuristics in Search:

- Heuristics are functions that estimate the cost of reaching the goal from a given node.
- A good heuristic leads to fewer explored nodes and faster problem-solving.

Role in Efficiency:

- Heuristics can drastically reduce search time by pruning paths that are unlikely to lead to optimal solutions.



MSKU CENG | CENG-3511 Artificial Intelligence

75

Heuristic Search Strategies

Greedy Best-First Search (Greedy BFS):
Selects the node that appears to be closest to the goal according to the heuristic function. However, it is not guaranteed to find the shortest path.

A* Search:
Combines the cost of the path so far (*g(n)*, where n is a neighbor node to the current node) with the heuristic estimate of the remaining cost to reach the goal (*h(n)*).

$$f(n) = g(n) + h(n)$$

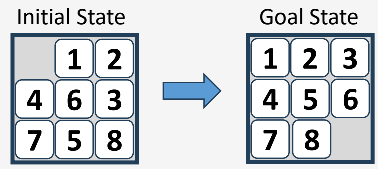
$$cost(n) = past(n) + future(n)$$

MSKU CENG | CENG-3511 Artificial Intelligence

76

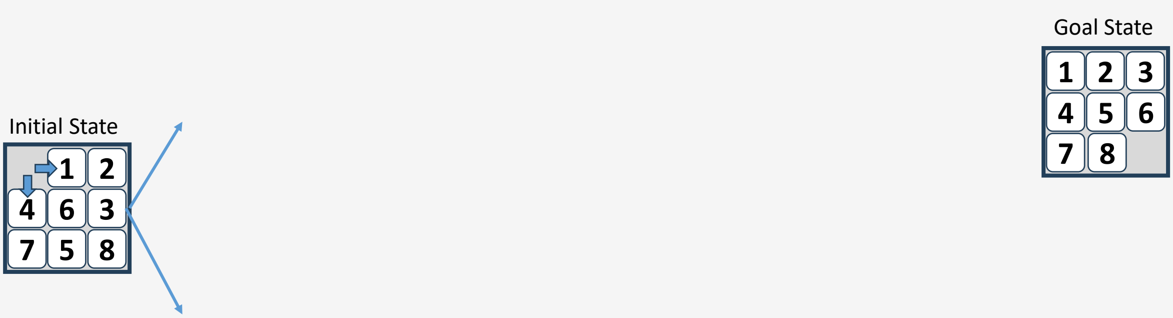
8-Puzzle Problem

- The 8-Puzzle problem is a classic sliding puzzle that involves a 3x3 grid with 8 numbered tiles and an empty space.
- The goal is to rearrange the tiles in ascending numerical order by sliding them into adjacent empty spaces.



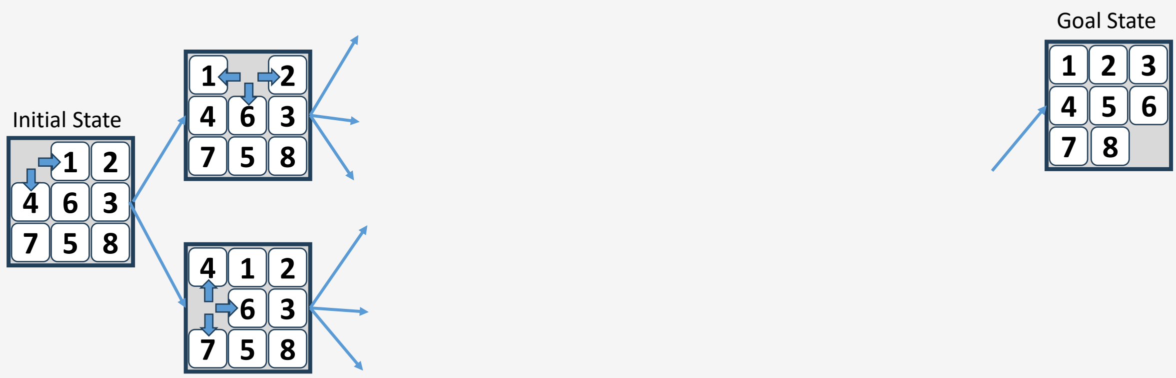
78

8-Puzzle Problem



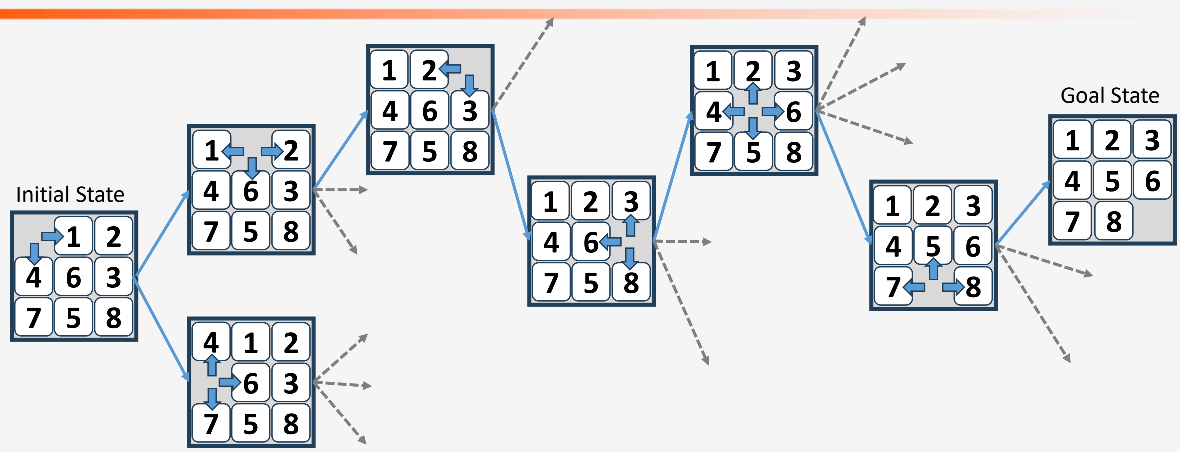
79

8-Puzzle Problem



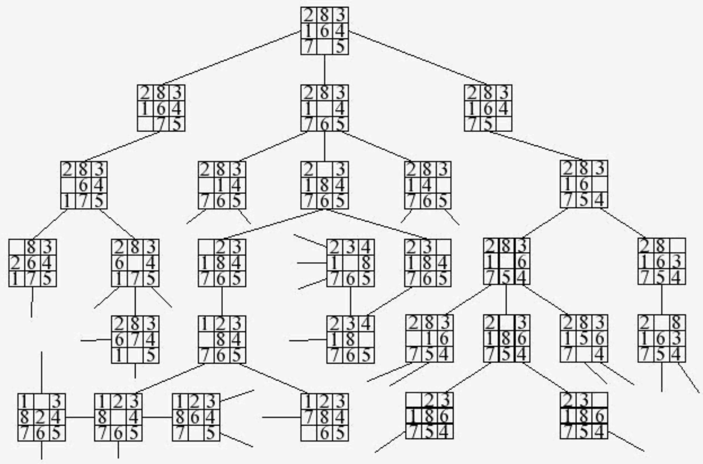
80

8-Puzzle Problem

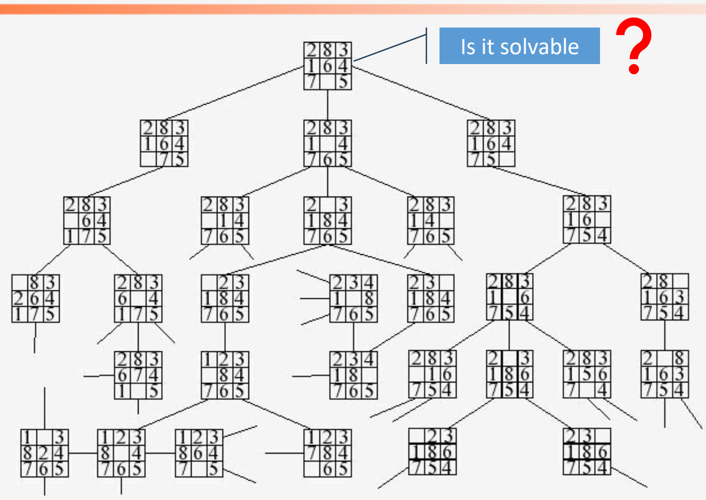


81

How deep the search tree could be?

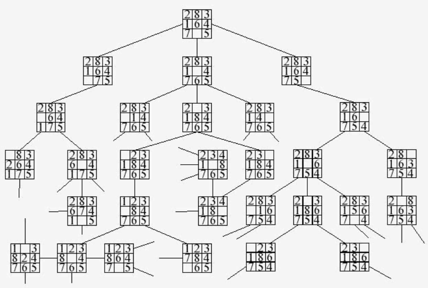


How to check 8-Puzzle config is solvable?



How to check 8-Puzzle config is solvable?

Answer is NO!



How to check?

```python
def count_inversions(board):
    """
    Flattens the board and counts the number of inversions.

    The inversion count refers to the number of tile pairs that are in
    the wrong order. If the inversion count is odd, the puzzle is unsolvable
    if even, it's solvable.
    """
    flat_board = [tile for row in board for tile in row if tile != 0]  # Fla
    inversions = 0
    for i in range(len(flat_board)):
        for j in range(i + 1, len(flat_board)):
            if flat_board[i] > flat_board[j]:
                inversions += 1
    return inversions
```


Heuristic Functions

- **Manhattan Distance**
  - a measure of the total distance between each tile and its goal position in the puzzle
  - It is calculated by summing the horizontal and vertical distances between the current position of each tile and its target position

range(1,9)

1 2 3 4 5 6 7 8 9

self.board = [0, 1, 2, 4, 6, 3, 7, 5, 8]

```python
def manhattan_distance(self):
    distance = 0
    for i in range(1, 9):
        x1, y1 = divmod(self.board.index(i), 3)
        x2, y2 = divmod((i - 1), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

2D Square Matrix Traversal with one loop

```python
result = divmod(10, 3)
print(result)  # Output: (3, 1)
```

In this example, 10 is divided by 3. The quotient is 3 (10 divided by 3 equals 3 with a remainder of 1) and the remainder is 1.

Heuristic Functions

- **Number of Wrong Tiles**
  - a measure of how far the current state is from the goal state
  - how many tiles (except the blank tile `0`) are not in their correct positions compared to the goal state.

Can you suggest another heuristic?

Initial state = [0, 1, 2, 4, 6, 3, 7, 5, 8]

Goal state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

```
def num_wrong_tiles(self, goal_state):
    """
    Calculates the number of tiles in the wrong position
    compared to the goal state.
    """
    wrong_tiles = 0
    for r in range(3):
        for c in range(3):
            if self.board[r][c] != goal_state[r][c] \
               and self.board[r][c] != 0:
                wrong_tiles += 1
    return wrong_tiles
```

2D Square Matrix Traversal with 2 loops

Greedy BFS vs A* Search

Criteria	Greedy Best-First Search (GBFS)	*A Search**
Heuristic Use	Only uses the heuristic (h).	Uses both path cost (g) and heuristic (h).
Goal	Fast movement towards goal using heuristic.	Balances between shortest path and heuristic.
Optimality	**Not optimal**.	**Always optimal** if the heuristic is admissible.
Completeness	**Incomplete**. May fail to find a solution.	**Complete**. Guaranteed to find a solution.
Speed	Typically faster than A*.	Slower than GBFS due to evaluating path cost.
Memory Usage	Usually lower than A*.	Typically higher due to storing both g and h.
Handling Loops	Prone to getting stuck in loops.	Handles loops better by considering path cost.
Use Case	Quick, approximate solutions.	Accurate, optimal solutions where path cost matters.

Greedy BFS vs A* Search

Major Differences:

- **Optimality & Completeness**:
  - A* is always optimal and complete (if heuristic is **admissible** and **consistent**), while GBFS is not.
- **Heuristic Use**:
  - A* combines heuristic with path cost, GBFS only uses heuristic.
- **Speed**:
  - GBFS is often faster but less reliable.
- **Memory**:
  - A* generally uses more memory.

Example Use Cases:

- **Greedy BFS**: Useful in applications where finding a quick, approximate solution is more important than finding the best or shortest path (e.g., some real-time applications).
- **A***: Used in scenarios where the shortest path is required and optimality matters (e.g., route planning, pathfinding in games).

Admissible and Consistent Heuristics

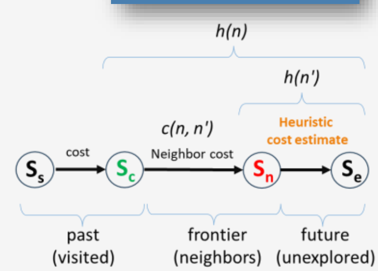
A heuristic is **consistent**, if the estimated cost *h(n)* to reach the goal from *n* is not greater than the cost to reach a neighboring node *n'* plus the estimated cost from *n'* to the goal.

$$h(n) \leq h(n') + c(n, n')$$

Where:

h(n) is the heuristic value for node *n*.
h(n') is the heuristic value for the successor node *n'*.
c(n, n') is the actual cost of moving from node *n* to *n'*.

$$h(n) \leq h(n') + c(n, n')$$



* An **admissible heuristic** is a heuristic that **never overestimates** the cost of reaching the goal in a search algorithm:

Consistency => Admissibility

Manhattan Distance is an admissible heuristic for grid-based pathfinding because it calculates the minimum number of moves required to reach the goal, assuming no obstacles.

Since it doesn't account for obstacles (which would increase the actual path cost), it never overestimates the true distance.

A* vs UCS

Criteria	Uniform-Cost Search (UCS)	*A Search**
Heuristic Use	No heuristic, uses path cost (g).	Combines path cost (g) and heuristic (h).
Optimality	Always optimal.	Always optimal with an admissible heuristic.
Completeness	Always complete.	Always complete.
Speed	Slower, explores nodes purely by cost.	Faster with a good heuristic.
Memory Usage	Higher due to broader exploration.	More efficient if heuristic is well-suited.
Use Case	No heuristic available, cost-based searches.	Best when a good heuristic is available.

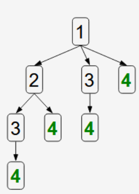
- ✓ **UCS is useful when no good heuristic is available** and you only care about finding the lowest-cost path.
- ✓ **A* is a more efficient alternative when you have an admissible heuristic** to guide the search, making it generally faster than UCS while maintaining optimality.

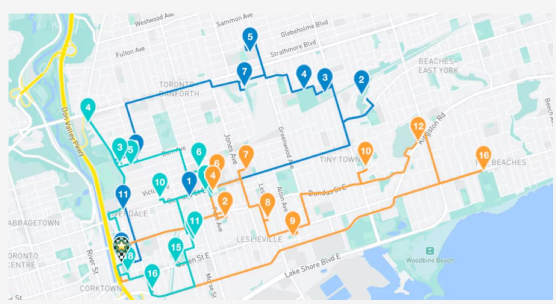
Example: Route Finding

Goal:

- Find the minimum cost path from city *i* to city *j*, only moving forward.
- The traveling cost, **cost**(*i*,*j*) is constant for all pair of cities (*i*,*j*), i.e. cost(*i*,*j*)=1

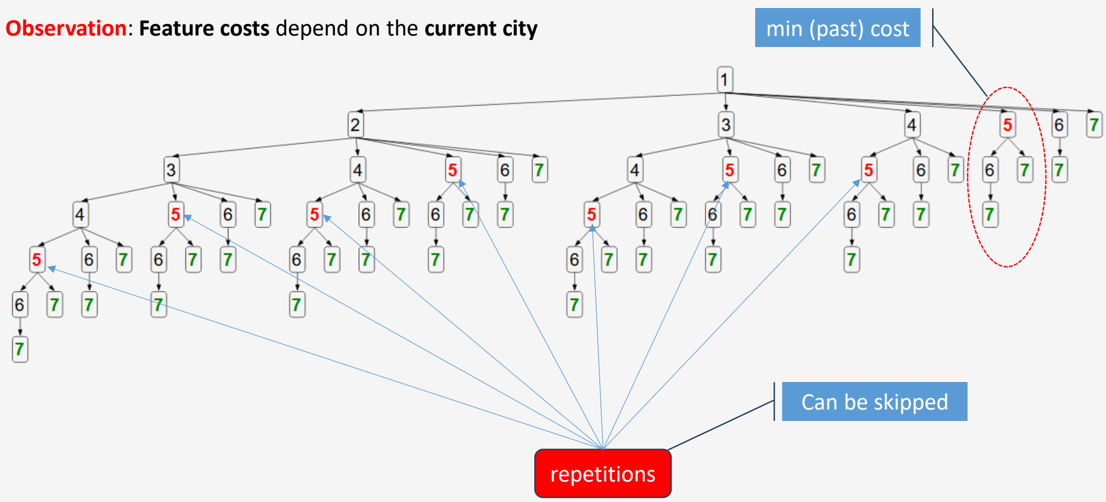
Say *i*=**1** and *j*=**4**, then search tree will be:





Example: Route Finding

Observation: **Feature costs** depend on the **current city**



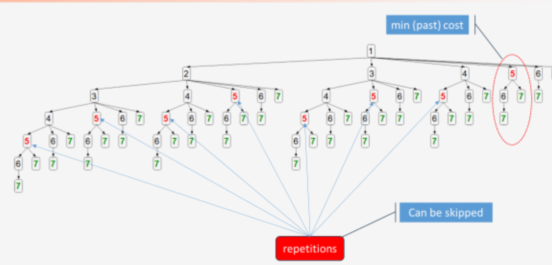
Dynamic Programming

Cost Memoization

- The min cost of reaching to city 5 from the start (past cost) can be calculated and all the paths to the city 5 from the start with higher costs can be discarded.

+ Heuristic Memoization

- The future costs (the minimum cost of reaching a end state) of a state only depends on the current city! So therefore, all the subtrees rooted at city 5, for example, have the same minimum future cost!
- If we can just do that computation once, then we will have saved big time.



Memoization and reusing previously computed results in A*

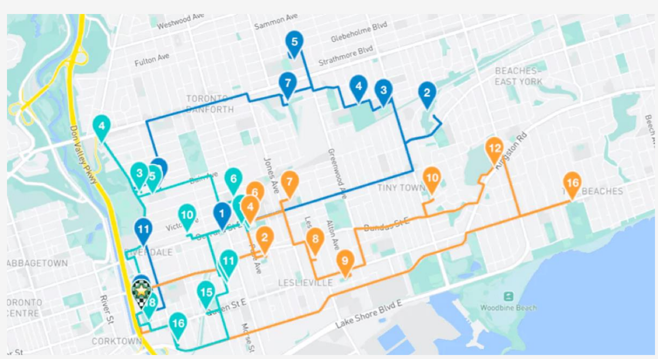
CENG3511-AI-Lab2-RouteFinding-DynamicProgramming.ipynb

Dynamic Programming Pros & Cons

Aspect	*A with Memoization**	Vanilla A*
Speed	Faster for expensive heuristics.	Can be slower due to repeated computations.
Memory Usage	Higher due to caching heuristics.	Lower memory usage, no caching.
Implementation	More complex, requires cache management.	Simpler, no extra complexity.

Homework

Discuss the use of Memoization for A* Search with respect to Route Planning.



Local Search Strategies

Hill Climbing, Simulated Annealing & Genetic Algorithms

MSKU CENG | CENG-3511 Artificial Intelligence

99

Local Search Strategies

Goal

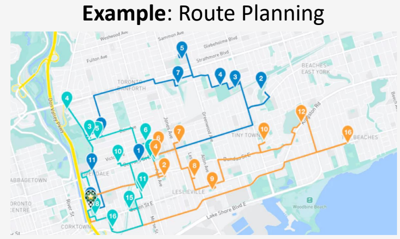
- to find a **near-optimal solution** to a **complex optimization problem**

Definition

Local search algorithms are a class of optimization algorithms that **explore the search (solution) space by making small changes to a current solution**

That is, they start by a particular solution to the problem and keep trying successive solutions derived from the initial solution by making small changes (step-by-step).

Example: Route Planning



The problem of finding the shortest route between cities in a map.

Local Search Algorithms
try to find a near-optimal solution,
a sequence of cities
minimizing the total distance traveled,
under a constraint (run time, # of trials, etc.),
which perhaps may or may not be the shortest path
(i.e., near-optimal solution).

MSKU CENG | CENG-3511 Artificial Intelligence

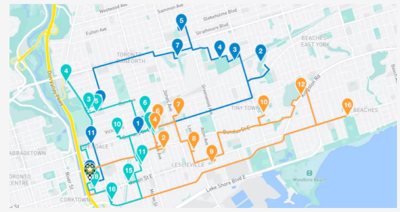
100

Local Search Strategies

To find a sequence of cities, for the example,

- **Hill Climbing**
  - takes steps towards the nearest neighbor with a lower distance.
- **Simulated Annealing**
  - allows for "bad" moves with a certain probability **to escape local optima**.
- **Genetic Algorithms**
  - combine and modify solutions to create new, potentially better ones.

Example: Route Planning



The problem of **finding the shortest route** between cities in a map.

Local Search Algorithms
try to find a near-optimal solution,
a sequence of cities
minimizing the total distance traveled,
under a constraint (run time, # of trials, etc.),
which perhaps may or may not be the shortest path
(i.e., near-optimal solution).

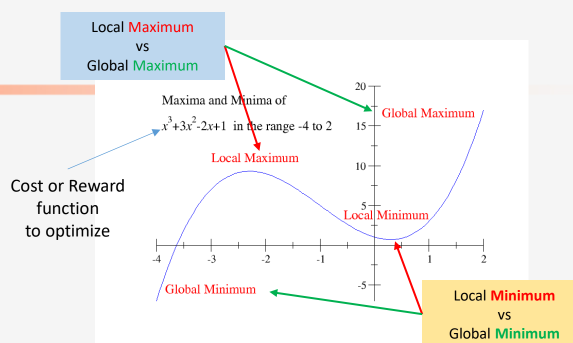
Local & Global Optima

Local Optima

- a solution that is better than all its neighboring solutions

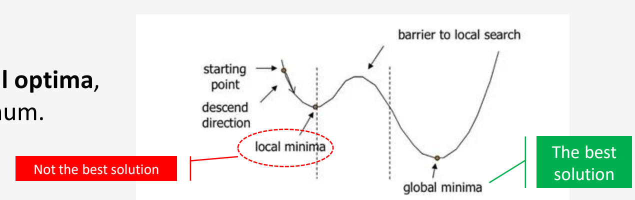
Global Optima

- **the best possible solution** among all feasible solutions



Challenge

Local search algorithms **often get stuck in local optima**, preventing them from finding the global optimum.



Hill Climbing

```
**Algorithm:**
1. Start with a random initial solution.
2. If there is a neighboring solution with a better objective value, move to it.
3. Repeat step 2 until no better neighboring solution is found.
```

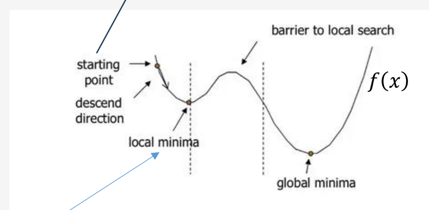
```python
def hill_climbing(objective_function, initial_solution):
    current_solution = initial_solution
    while True:
        neighbors = generate_neighbors(current_solution)
        best_neighbor = find_best_neighbor(neighbors, objective_function)

        best_neighbor_value = objective_function(best_neighbor)
        current_solution_value = objective_function(current_solution)

        if best_neighbor_value <= current_solution_value:
            break
        current_solution = best_neighbor
    return current_solution
```

return value

Objective function

any real valued function $f(x)$

initial_solution

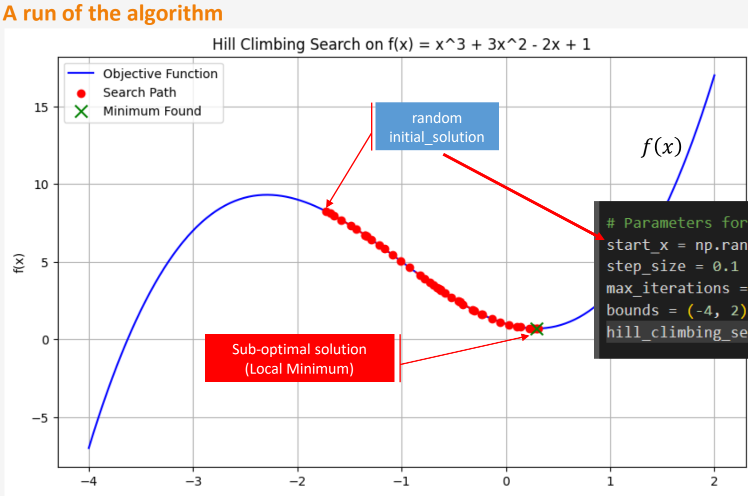


103

Hill Climbing

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb

A run of the algorithm




```python
# Parameters for the hill climbing algorithm
start_x = np.random.uniform(-4, 2)  # Random start in the range
step_size = 0.1  # Step size for the search
max_iterations = 1000  # Maximum number of iterations
bounds = (-4, 2)  # Search bounds
hill_climbing_search(start_x, step_size, max_iterations, bounds)
```

Objective function

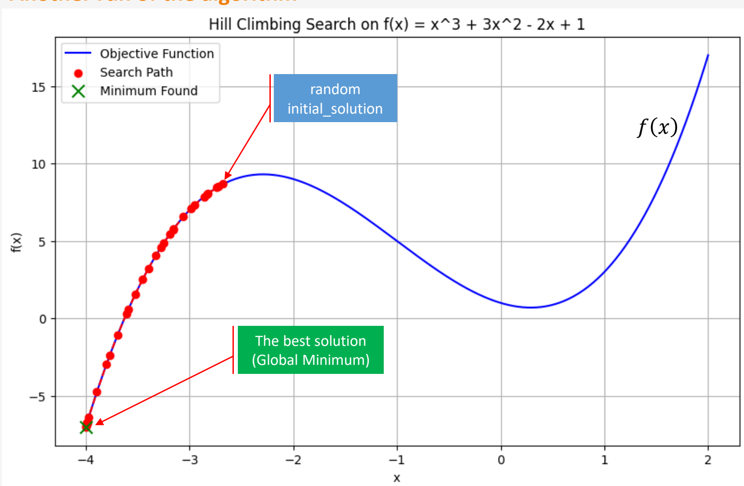
$$f(x) = x^3 + 3x^2 - 2x + 1$$

104

Hill Climbing

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb

Another run of the algorithm



Objective function

$$f(x) = x^3 + 3x^2 - 2x + 1$$

105

Simulated Annealing

Key Idea

- to avoid local optima via accepting “bad” moves by chance

Definition

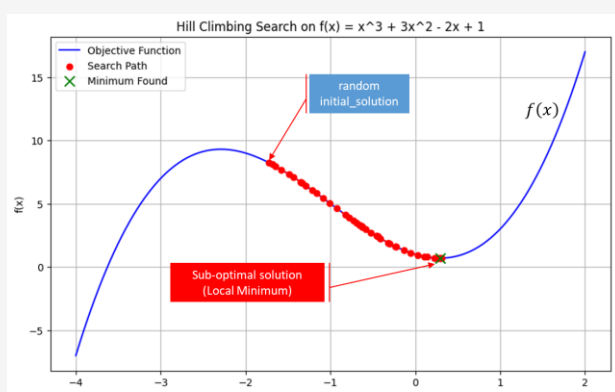
Simulated Annealing is a probabilistic algorithm that allows for "bad" moves with a certain probability.

Key points:

- **Temperature** :a measure of the algorithm's randomness or "energy."
- **Cooling schedule**: how the temperature decreases over time.
- **Acceptance probability**: The probability of accepting a "bad" move

Challenge

Local search algorithms **often get stuck in local optima**, preventing them from finding the global optimum.



107

Simulated Annealing

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb

Temperature :

Higher temperatures allow the algorithm to explore a wider range of solutions, including those that might initially appear suboptimal

Cooling schedule:

A well-designed cooling schedule ensures that the algorithm explores the solution space sufficiently at high temperatures while eventually converging to a near-optimal solution at low temperatures.

Acceptance probability: The probability of accepting a "bad" move

- The acceptance probability is determined by the temperature and the change in objective function value. At higher temperatures, the algorithm is more likely to accept "bad" moves, allowing it to escape local optima. As the temperature decreases, the probability of accepting "bad" moves also decreases, focusing the search on more promising solutions.

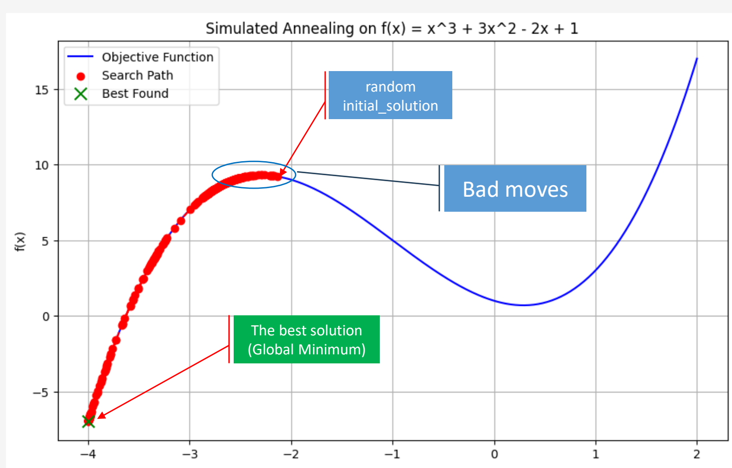
```
def simulated_annealing(initial_state, temperature, cooling_rate):
    current_state = initial_state
    while temperature > 0:
        neighbor = generate_neighbor(current_state)
        delta_energy = objective_function(neighbor) \
                        - objective_function(current_state)
        if delta_energy < 0 or \
                random.random() < math.exp(-delta_energy / temperature):
            current_state = neighbor
        temperature *= cooling_rate
    return current_state
```


108

Simulated Annealing

Temperature: 10.0
Cooling rate: 0.99

Exponential decay in temperature



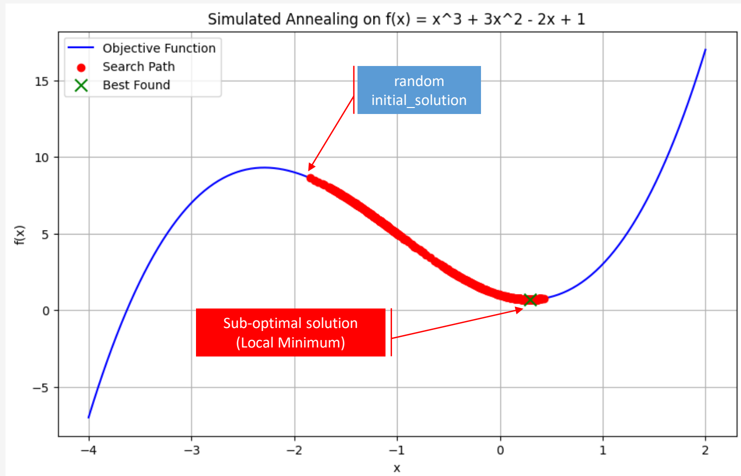
109

Simulated Annealing

Temperature: 10.0
Cooling rate: 0.99

Exponential decay in temperature

Exercise: Adjust temperature & cooling rate to avoid local minima



CENG-3511 Artificial Intelligence

110

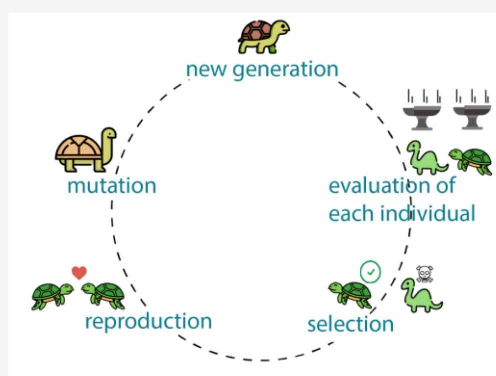
Genetic Algorithms

Key idea: **Mimic natural evolution**

- Genetic Algorithms are inspired by natural evolution and use genetic operators to create new solutions.

Key points:

- **Selection**: Natural selection (who lives, who dies)
- **Crossover**: Reproduction of offsprings (child)
- **Mutation**: Random alternation of genetic material



CENG-3511 Artificial Intelligence

111

Genetic Algorithms

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb

Population

Objective function

$$f(x) = x^3 + 3x^2 - 2x + 1$$

randomly select a set of x values from -4 to 2	{-3.0, 1.0, -2.0}
Evaluate f(x) values for the current generation	{7.0, 3.0, 8.0}
Choose a fix # of x values, say two, with smallest f(x): others will die	{-3.0, 1.0}
Produce two offsprings from the selected parents: (-3 * 0.6 + 1.0 * 0.4) and (-3 * 0.5 + 1.0 * 0.5)	{-1.4, -1.0}
	{-1.4, -1.1}
Add offsprings to the population and repeat	{-3.0, 1.0, -1.4, -1.1}

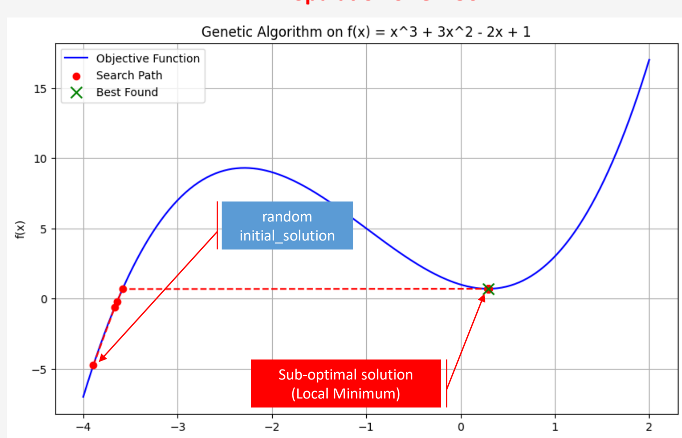
```python
def genetic_algorithm(objective_function, population_size,
                      num_generations, crossover_rate, mutation_rate):
    # ... (initialize population)
    for generation in range(num_generations):
        # ... (evaluate fitness)
        # ... (select parents)
        # ... (crossover)
        # ... (mutation)
    # ... (return best solution)
```


Genetic Algorithms

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb


```python
# Parameters for the genetic algorithm
pop_size = 50  # Population size
generations = 100  # Number of generations
crossover_rate = 0.8  # Crossover rate
mutation_rate = 0.1  # Mutation rate
bounds = (-4, 2)  # Search bounds
```

Population Size = 50

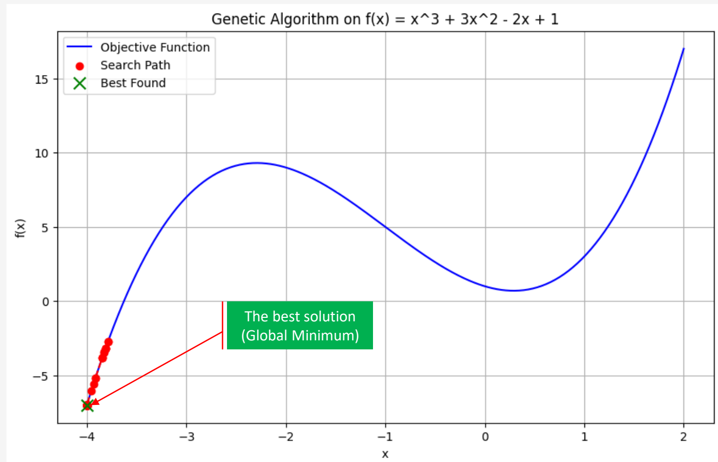


Genetic Algorithms

CENG3511-AI-Lab2-LocalSearchStrategies.ipynb

Population Size = 100

```
# Parameters for the genetic algorithm
pop_size = 100  # Population size
generations = 100  # Number of generations
crossover_rate = 0.8  # Crossover rate
mutation_rate = 0.1  # Mutation rate
bounds = (-4, 2)  # Search bounds
```



Real-world Applications

Optimization of machine learning methods

- Finding the optimal weights and biases, hyper-parameter values, etc.
- We'll revisit Local Search Algorithms in
  - Week 5: Optimization and Metaheuristics
  - Weeks 9-12: Machine Learning

Scheduling problems

- Allocating resources to tasks efficiently: Timetabling, Traveling salesman

Protein structure prediction

- Determining the 3D structure of a protein.

Many other areas: Engineering, finance, biology, and more.

Summary

Local Search Strategies
- Hill Climbing
- Simulated Annealing
- Genetic Algorithms

and others,
- Tabu Search
- Variable Neighborhood
- ...

Uninformed Search
- Breadth-first
- Depth-first
- DLS, IDS
- Uniform Cost Search

Informed Search
- Greedy Best-First
- A* Search

Cheaper
(~ linear complexity)

Sub-optimal solutions

Simulated Annealing < Uniform Cost < A* Search

Expensive
(~ exponential complexity)

Optimal solution