Mahidol University

Faculty of Information and Communication Technology

Heuristic Search and Backtracking Algorithms

ITCS 503 – Design and Analysis of Algorithms

ปัญญาของแผ่นดิน Wisdom of the Land

Heuristic Search Algorithms

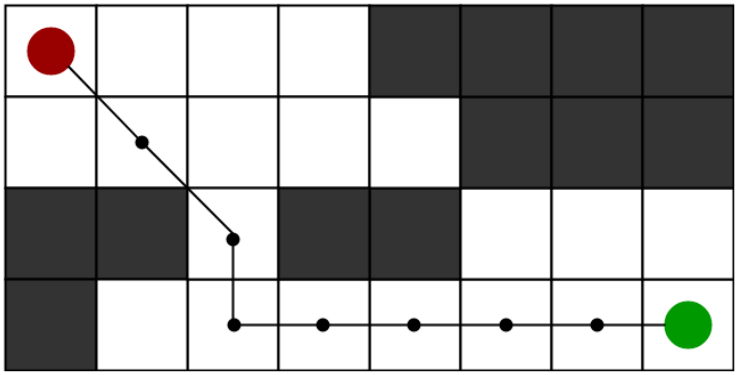
- In the context of heuristic search algorithm, heuristics refer to a set of criteria or rules of thumb or approaches that provide an estimate of the most viable solution.
- Heuristic search algorithms are often used for exploring different paths to obtain the optimal solution or a specified goal (target state).
- Heuristic search operates within the search space of a problem to find the best or near-optimal solution using systematic algorithms.
  - Unlike brute-force methods, which exhaustively evaluate all possible solutions, heuristic search leverages heuristic information to guide the search toward more promising paths.
  - By balancing exploration (searching new possibilities) and exploitation (refining known solutions), heuristic algorithms efficiently solve complex problems that would otherwise be computationally expensive.
- Heuristic search algorithms are often used in navigation systems, game playing, and optimization.
- The advantage of heuristic search techniques is their ability to efficiently navigate large search spaces.
  - By prioritizing the most promising paths, heuristics significantly reduce the number of possibilities that need to be explored.
  - This not only accelerates the search process but also allow us to solve complex problems that would be impractical for exact algorithms.

Components of Heuristic Search Algorithms

- State Space:
  - This implies that the totality of all possible states or settings, which is considered to be the solution for the given problem.
- Initial State:
  - The instance in the search tree of the highest level with no null values, serving as the initial state of the problem at hand.
- Goal Test:
  - The exploration phase ensures whether the present state is a terminal or consenting state in which the problem is solved.
- Successor Function:
  - This create a situation where individual states supplant the current state which represent the possible moves or solutions in the problem space.
- Heuristic Function:
  - The function of a heuristic is to estimate the value or distance from a given state to the target state. It helps to focus the process on regions or states that has prospect of achieving the goal.

A* Search Algorithm

- **Motivation**
  - To approximate the shortest path in real-life situations, e.g., in maps, games, mazes where there can be many obstacles or hindrances.
  - We can consider a 2D grid map having several obstacles. We would like to start from a source cell (colored red below) to reach towards a goal cell (colored green below) using the shortest path.

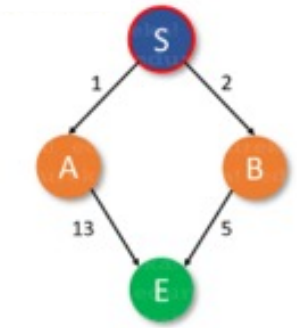


A* Search Algorithm

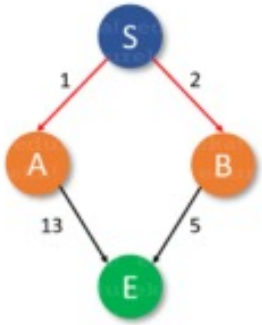
- **Explanation of approach**
  - Consider a square grid having many obstacles and we are given a starting cell (**s**) and a target cell (**d**).
  - We want to reach the target cell (if possible) from the starting cell as quickly as possible.
  - At each step, we pick the node according to a value-**'F'** which is a parameter equal to the sum of two other parameters – **F = G + H**
    - **'F'** : the sum of the other parameters G and H and is the least cost from one node to the next node. This parameter is responsible for helping us find the most optimal path from our source to destination.
    - **'G'** : the movement cost to move from **s** to a given square on the grid (**x**), following the path that has been generated to get there
    - **'H'** : the estimated movement cost to move from **x** to the **d**. This **'H'** is often referred to as the heuristic. This cost is not actual but is, in reality, a guess cost that we use to find which could be the most optimal path between the source and destination. which is nothing but a kind of smart guess.
  - Note that we really don't know the actual distance until we find the path, because all sorts of things can be in the way (obstacles such as walls, water, etc.). There can be many ways to calculate this **'h'** which are discussed in the later sections.

A* Search Algorithm

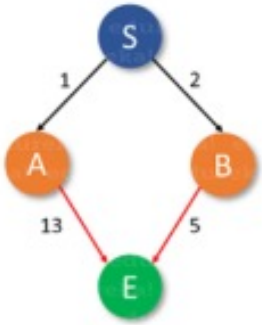
- **Simple example**



F = G + H
For source,
f = g + h = 0 + 5 = 5



For S-A, 1 + 4 = 5
For S-B, 2 + 5 = 7
Choose S-A



For S-A-E, (1+13) + 0 = 14
For S-B-E, (2+5) + 0 = 7
Choose S-B-E

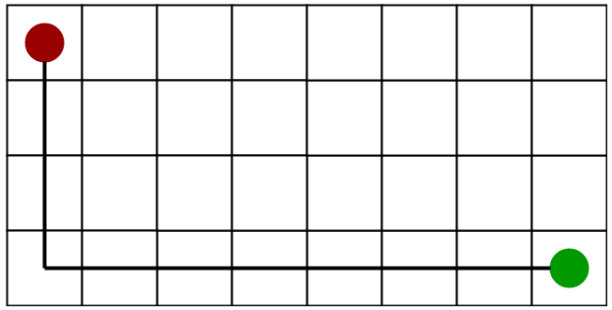
  - '**H**' can be
    - a random number (a guess).
    - an estimate distance between the current node and destination node.
  - After calculation, we have now found that B later has given us the least path. So, we change our least path to S-B-E and have reached our destination. That is how we use the formula to find out the most optimal path..

A* Search Algorithm

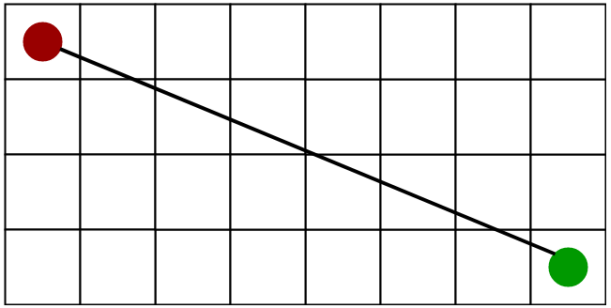
- Calculation of '**H**'
  - **Exact heuristic calculation**
    1. Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
    2. If there are no blocked cells/obstacles then we can just find the exact value of '**H**' without any pre-computation using the distance formula such as Manhattan or Euclidean Distance
       - Manhattan distance: the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively,
       - Euclidean distance:
  - Approximated heuristic calculation

A* Search Algorithm

- Calculation of '**H**'
  - Method 1: Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
  - Method 2: If there are no blocked cells/obstacles then we can just find the exact value of '**H**' without any pre-computation using the distance formula such as Manhattan or Euclidean Distance
    - Manhattan distance: the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively,
    - Euclidean distance: the geometric distance between the current cell and the goal cell



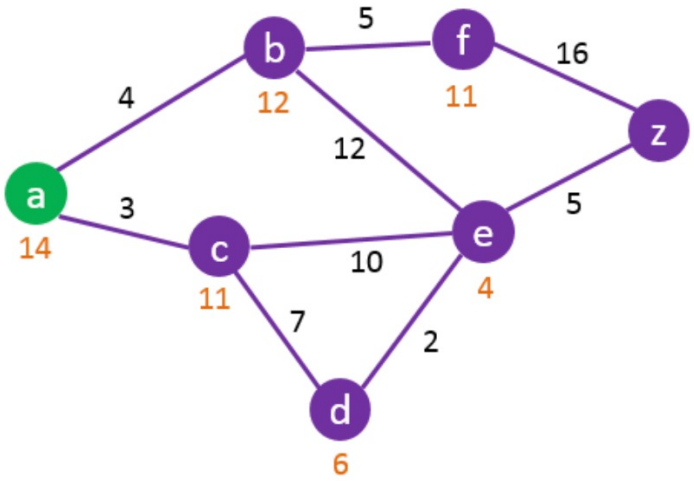
Manhattan distance



Euclidean distance

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



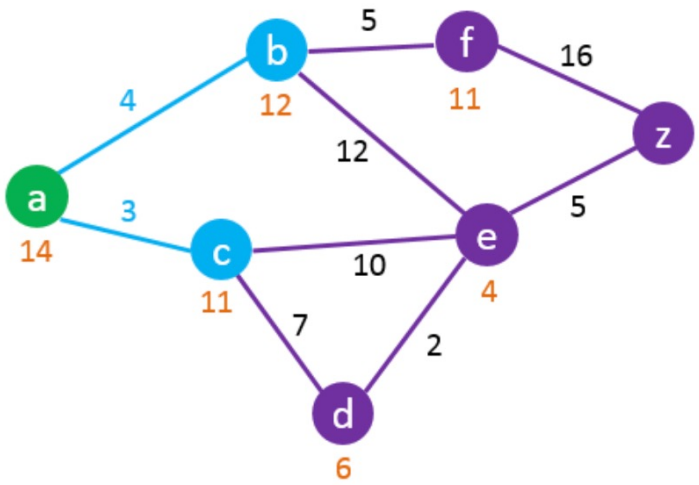
Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Current	0	14	14	
B		∞	12		
C		∞	11		
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Start by setting the starting node (A) as the current node.

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Current	0	14	14	
B		∞ 4	12	16	A
C		∞ 3	11	14	A
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

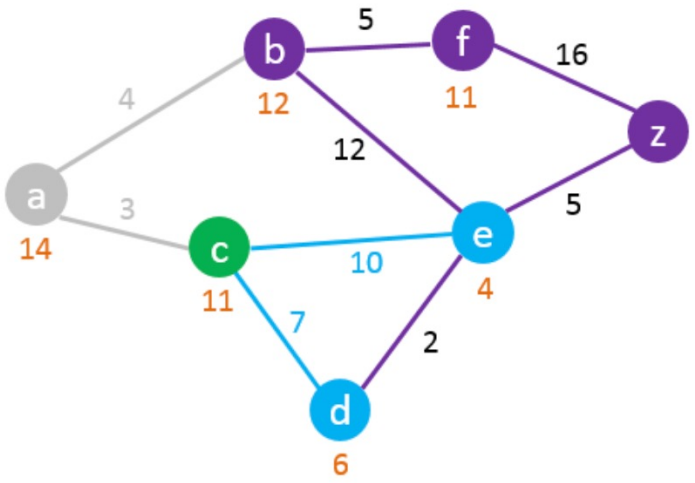
* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Check all the nodes connected to A and update their **"Shortest Distance from A"** and set their **"previous node"** to "A".

Update their total distance by adding the shortest distance from A and the heuristic distance to Z.

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B		4	12	16	A
C	Current	3	11	14	A
D		∞ 3+7=10	6	16	C
E		∞ 3+10=13	4	17	C
F		∞	11		
Z		∞	0		

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Set the current node (A) to **"visited"** and use the unvisited node with the smallest total distance as the **current node** (e.g. in this case: Node C).

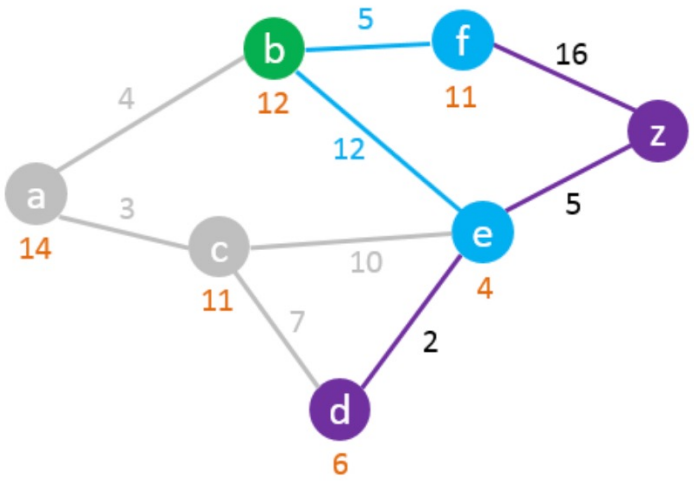
Check all unvisited nodes connected to the current node and add the distance from A to C to all distances from the connected nodes. Replace their values only if the new distance is lower than the previous one.

C -> D: 3 + 7 = 10 < ∞ – Change Node D
C -> E: 3 + 10 = 13 < ∞ – Change Node E

The next current node (unvisited node with the shortest total distance) could be either node B or node D. Let's use node B.

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B	Current	4	12	16	A
C	Visited	3	11	14	A
D		10	6	16	C
E		13 4+12=~~16~~	4	17	C
F		∞ 4+5=9	11	20	B
Z		∞	0		

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Check all unvisited nodes connected to the current node (B) and add the distance from A to B to all distances from the connected nodes.

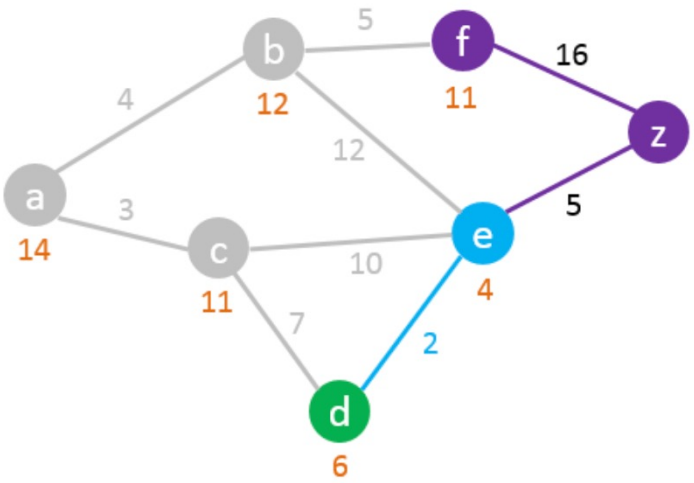
Replace their values only if the new distance is lower than the previous one.

B -> E: 4 + 12 = 16 > 13 – Do not change Node E
B -> F: 4 + 5 = 9 < ∞ – Change Node F

The next current node (unvisited node with the shortest total distance) is D.

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B	Visited	4	12	16	A
C	Visited	3	11	14	A
D	Current	10	6	16	C
E		~~13~~ 10+2=12	4	16	D
F		9	11	20	B
Z		∞	0		

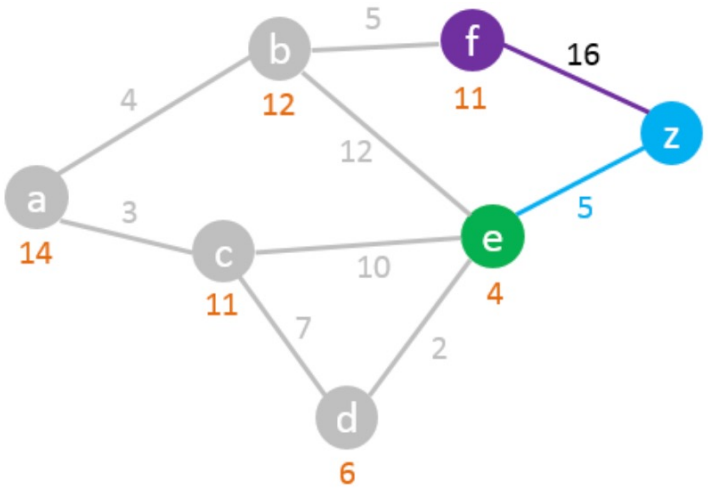
* Total Distance = Shortest Distance from A + Heuristic Distance to Z

Check all unvisited nodes connected to the current node (D) and add the distance from A to D to all distances from the connected nodes. Replace their values only if the new distance is lower than the previous one.

D -> E: 10 + 2 = 12 < 13 – Change Node E
The next current node (unvisited node with the shortest total distance) is E.

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B	Visited	4	12	16	A
C	Visited	3	11	14	A
D	Visited	10	6	16	C
E	Current	12	4	16	D
F		9	11	20	B
Z		∞ 12+5=17	0	17	E

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

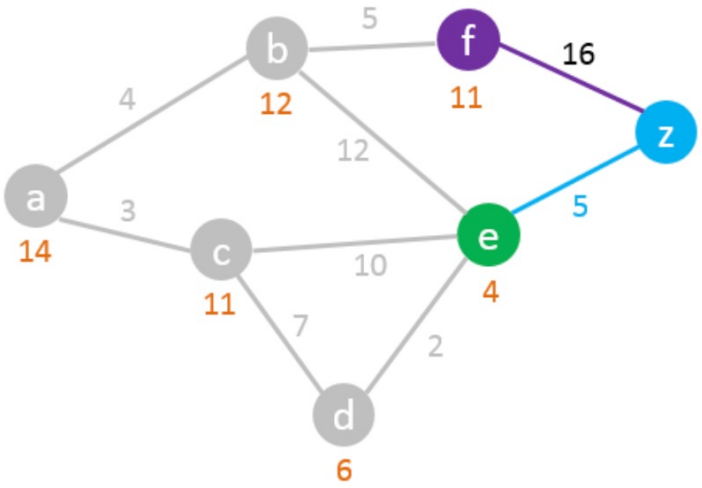
Check all unvisited nodes connected to the current node (E) and add the distance from A to E to all distances from the connected nodes.

Replace their values only if the new distance is lower than the previous one.

E -> Z: 12 + 5 = 17 < ∞ – Change Node Z

A* Search Algorithm

- Example: what is the shortest distance from **a** to **z**?



Node	Status	Shortest Distance From A	Heuristic Distance to Z	Total Distance*	Previous Node
A	Visited	0	14	14	
B	Visited	4	12	16	A
C	Visited	3	11	14	A
D	Visited	10	6	16	C
E	Visited	12	4	16	D
F		9	11	20	B
Z	Current	17	0	17	E

* Total Distance = Shortest Distance from A + Heuristic Distance to Z

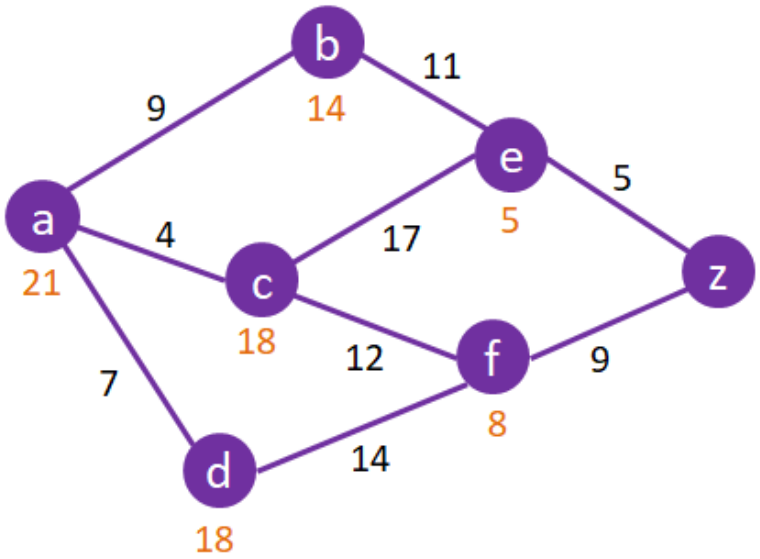
We found a path from A to Z, but is it the shortest one?

Check all unvisited nodes. In this example, there is only one unvisited node (F). However its total distance (20) is already greater than the distance we have from A to Z (17) so there is no need to visit node F as it should not lead to a shorter path.

We found the shortest path from A to Z.
Read the path from Z to A using the previous node column:
Z > E > D > C > A
So the Shortest Path is:
A – C – D – E – Z with a length of 17

Homework 1.1

- Solve for the shortest path from A to Z using A* search algorithm by filling in the table below
  https://www.101computing.net/a-star-search-algorithm/



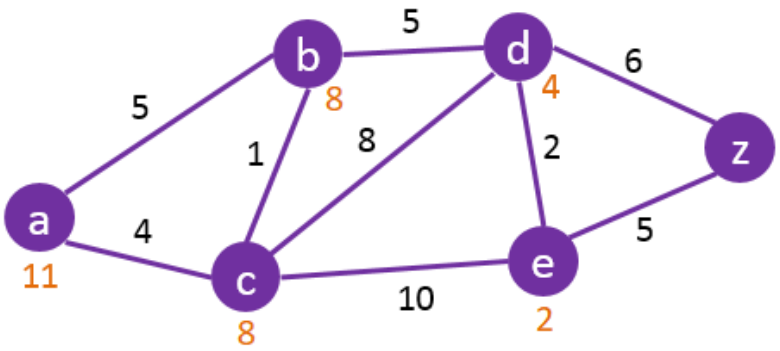
Shortest Path?

Length?

Node	Status	Shortest Distance from A	Heurisitic Distance to Z	Total Distance	Previous Node
A	Current	0	21	21	
B		∞	14		
C		∞	18		
D		∞	18		
E		∞	5		
F		∞	8		
Z		∞	0		

Homework 1.2

- Solve for the shortest path from A to Z using A* search algorithm
  https://www.101computing.net/a-star-search-algorithm/



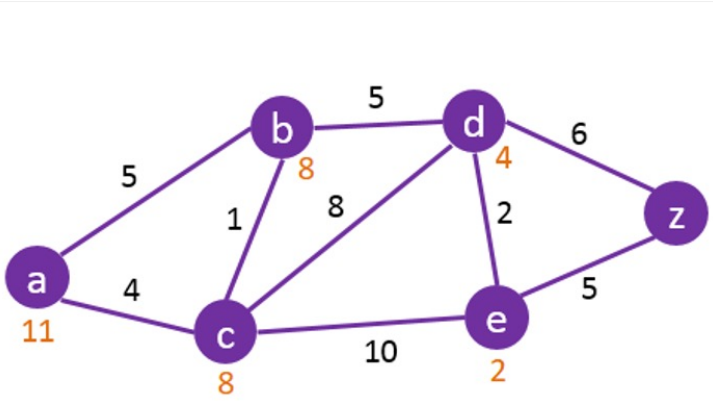
Shortest Path?

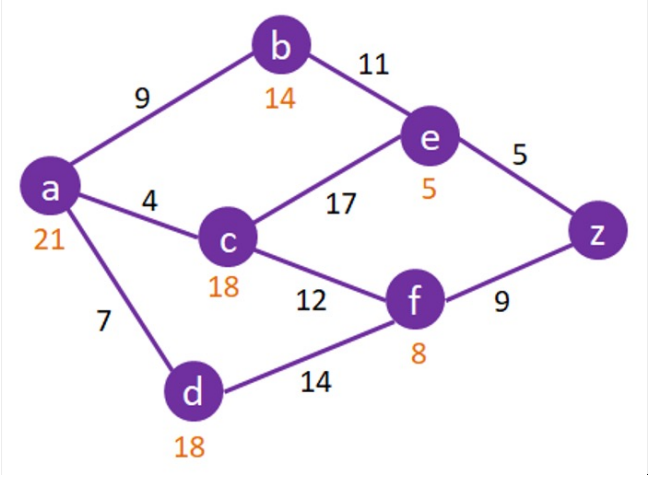
Length?

Node	Status	Shortest Distance from A	Heurisitic Distance to Z	Total Distance	Previous Node
A	Current	0	11	21	
B		∞	8		
C		∞	8		
D		∞	4		
E		∞	2		
Z		∞	0		

Homework 1.3

- Write a A* search algorithm as a function to create the routing table in Python. Test your function with homework 1.1 and 1.2. You need to use Graph data structure.

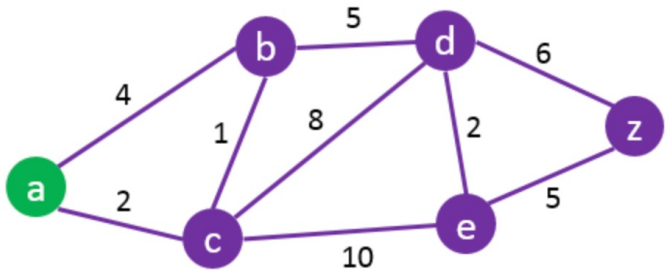




https://www.101computing.net/dijkstras-shortest-path-algorithm/

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.

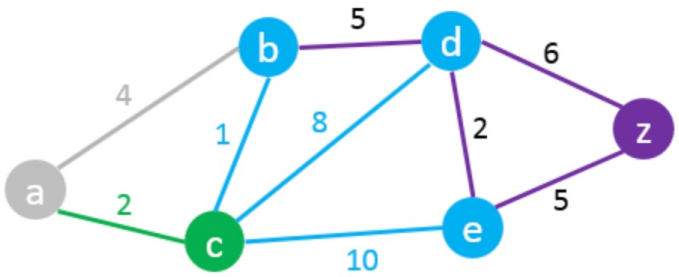


Node	Status	Shortest Distance From A	Previous Node
A	Current Node	0	
B		∞	
C		∞	
D		∞	
E		∞	
Z		∞	

Start by setting the starting node (**A**) as the current node.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



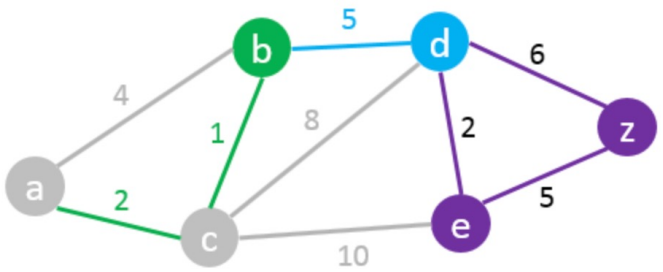
Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B		4 2+1=3	C
C	Current Node	2	A
D		∞ 2+8=10	C
E		∞ 2+10=12	C
Z		∞	

Check all unvisited nodes connected to the current node and add the distance from **A** to **C** to all distances from the connected nodes. Replace their values only if the new distance is lower than the previous one.

Through C we have
C -> B: 2 + 1 = 3 < 4 – Update how to reach Node B from A
C -> D: 2 + 8 = 10 < ∞ – Update how to reach Node D from A
C -> E: 2 + 10 = 12 < ∞ – Update how to reach Node E from A

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Current Node	3	C
C	Visited Node	2	A
D		10	C
E		12	C
Z		∞	

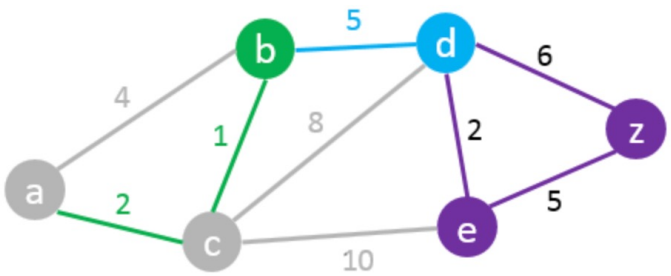
Set the current node **C** status to Visited since it has the smallest distance from A.

We then repeat the same process always picking the closest unvisited node to **A** as the current node. (Horizontal Traversal)

In this case node **B** becomes the current node.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



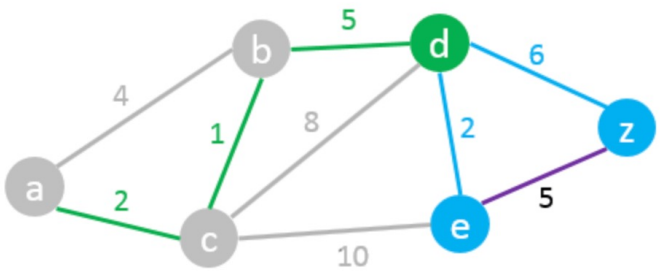
Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Current Node	3	C
C	Visited Node	2	A
D		~~10~~ 3+5=8	B
E		12	C
Z		∞	

In this case node **B** becomes the current node.

Determine where B can reach. Through B, we have
B -> D: 3 + 5 = 8 < 10 – Update how to reach node D from A
B -> C: Cannot be done (loop) since A now can reach B from C. Poison reverse.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Current Node	3	C
C	Visited Node	2	A
D		~~10~~ 3+5=8	B
E		12	C
Z		∞	

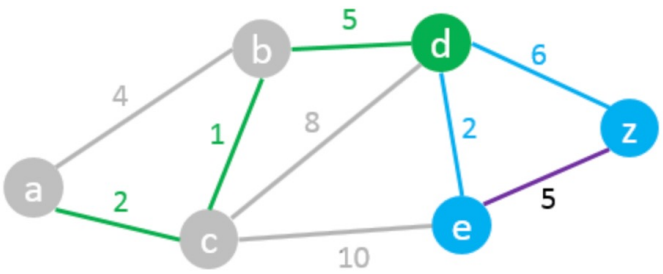
In this case node **B** becomes the current node.

Determine where B can reach. Through B, we have
B -> D: 3 + 5 = 8 < 10 – Update how to reach node D from A
B -> C: Cannot be done (loop) since A now can reach B from C. Poison reverse.

Next "Current Node" will be **D** since it has the shortest distance from A amongst all unvisited nodes.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Visited Node	3	C
C	Visited Node	2	A
D	Current Node	8	B
E		~~12~~ 8 + 2 = 10	D
Z		~~∞~~ 8 + 6 = 14	D

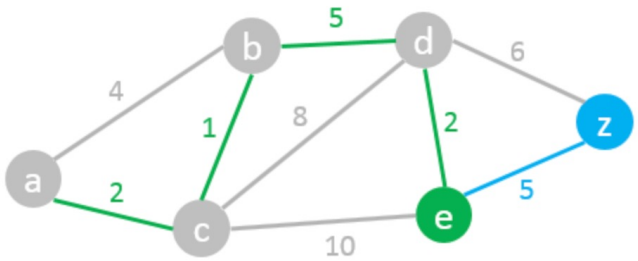
The "Current Node" is **D** since it has the shortest distance from A amongst all unvisited nodes. From current node **D**, we have

D -> E : 8+2 = 10 < 12 – Update how to reach Node E
D -> Z : 8+6 = 14 < ∞ – update how to reach Node Z

We found a path from A to Z but it may not be the shortest one yet. So we need to carry on the process to visit all the unvisited nodes.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



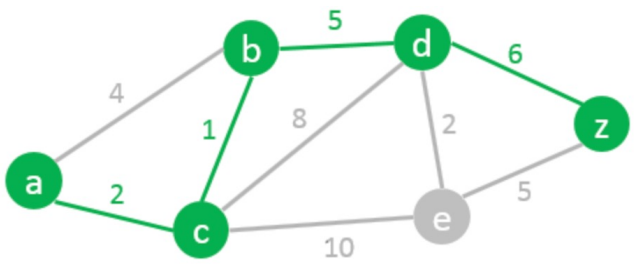
Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Visited Node	3	C
C	Visited Node	2	A
D	Visited Node	8	B
E	Current Node	10	D
Z		14 10 +5 =~~15~~	D

The "Current Node" is **E** since it has the shortest distance from A amongst all unvisited nodes. From current node **E**, we have

E -> Z : 10+5 = 15 > 14 – We do not update how to reach node Z.

Dijkstra's Shortest Path Algorithm

- Dijkstra's Shortest Path Algorithm is a special case of A* algorithm. It is used to find the shortest path between any two nodes on a weighted graph.



Node	Status	Shortest Distance From A	Previous Node
A	Visited Node	0	
B	Visited Node	3	C
C	Visited Node	2	A
D	Visited Node	8	B
E	Visited Node	10	D
Z	Current Node	14	D

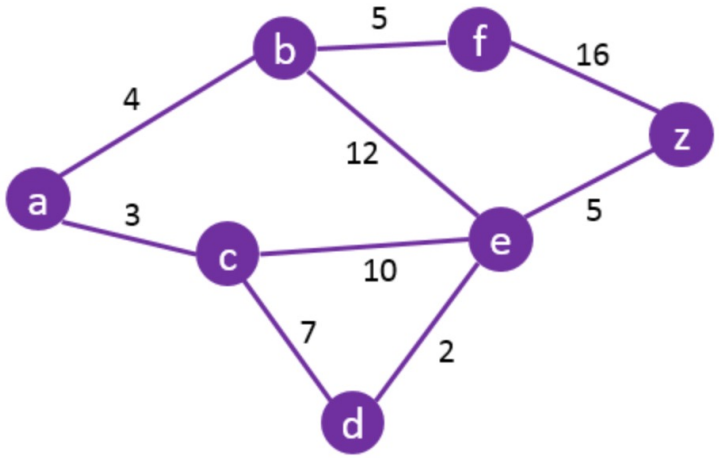
We found the shortest path from **A** to **Z**.
Read the path from Z to A using the previous node column:
Z > D > B > C > A

So the Shortest Path is:
A – C – B – D – Z with a length of 14

In addition, we also find the shortest path from A to B, C, D, and E as well.

Homework 2.1

- Update the routing table in the view of A using Dijkstar's shortest path algorithm.

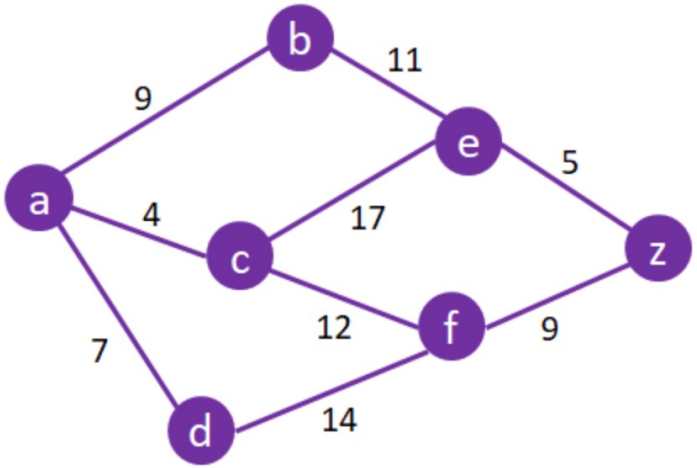


Node	Status	Shortest Distance from A	Previous Node
A	Current	0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
Z		∞	

https://www.101computing.net/dijkstras-shortest-path-algorithm/

Homework 2.2

- Update the routing table in the view of A using Dijkstar's shortest path algorithm.

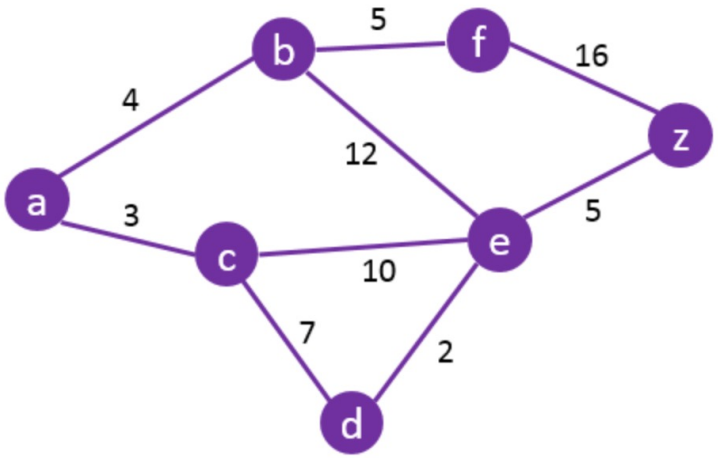


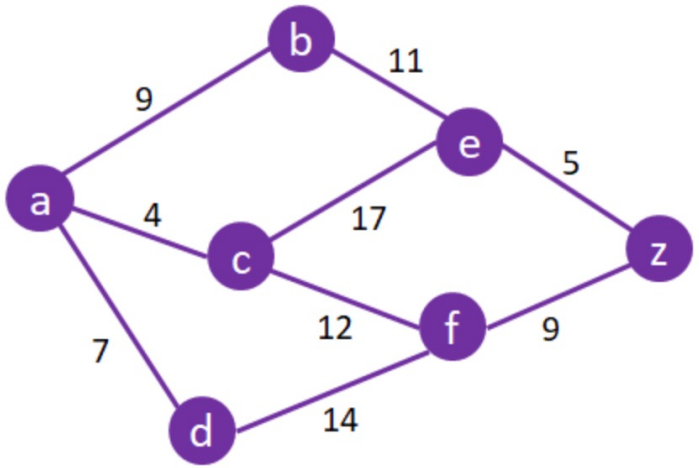
Node	Status	Shortest Distance from A	Previous Node
A	Current	0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
Z		∞	

https://www.101computing.net/dijkstras-shortest-path-algorithm/

Homework 2.3

- Write a Dijkstar's shortest path algorithm as a function to create the routing table in Python. Test your function with homework 2.1 and 2.2. You need to use Graph data structure.
- Compare your results against the results from A* search algorithm and report.





https://www.101computing.net/dijkstras-shortest-path-algorithm/

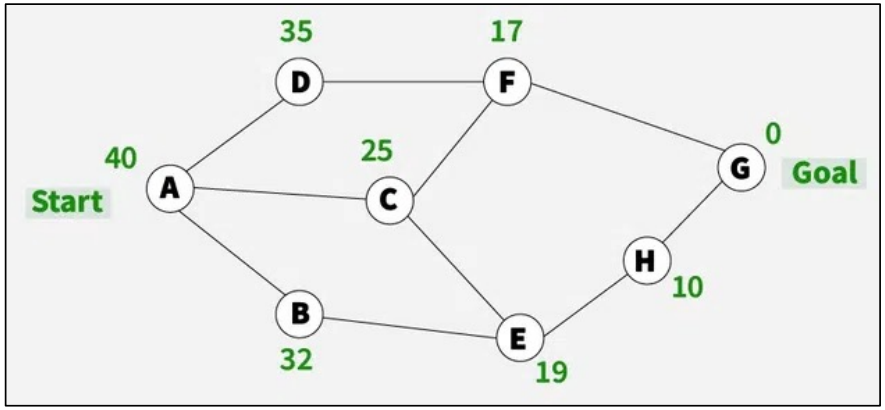
Greedy Best First Search Algorithm

- Greedy Best-First Search algorithm attempts to find the most promising path from a given starting point to a goal.
- It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path (hence, greedy).

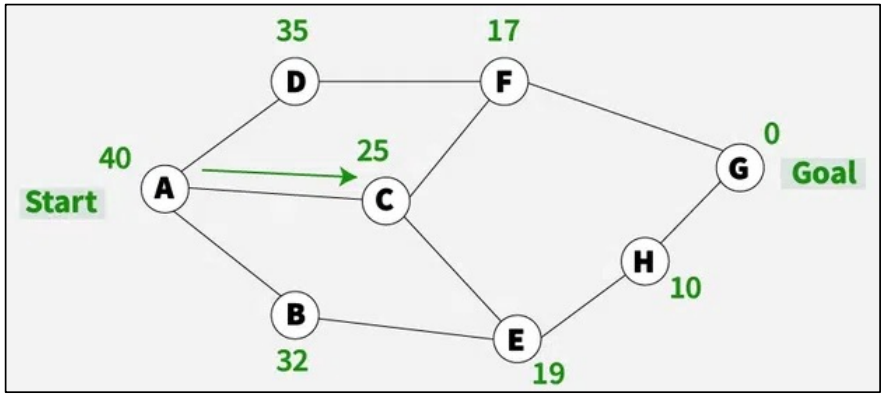
- **Approach**:
  - Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
  - The algorithm uses a heuristic function to determine which path is the most promising.
  - The heuristic function considers the cost of the current path and the estimated cost of the remaining paths.
  - If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

Greedy Best First Search Algorithm

- Example: We want to travel from A to G.



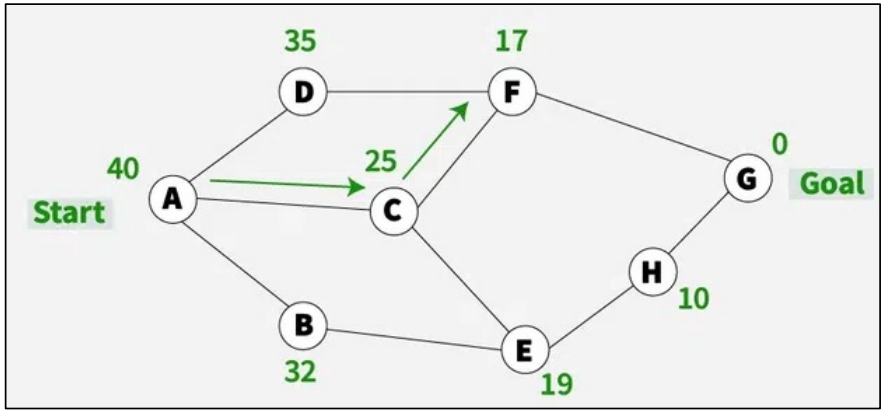
1) We are starting from A , so from A there are direct path to node B(with heuristics value of 32) , from A to C (with heuristics value of 25) and from A to D(with heuristics value of 35) .



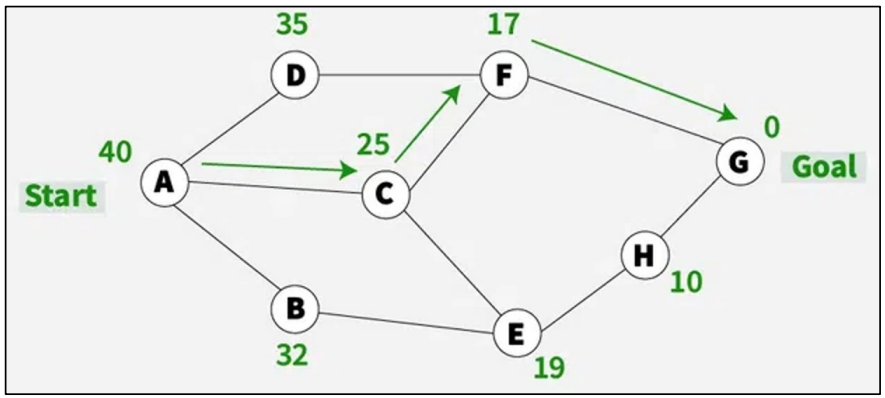
2) So as per best first search algorithm, we choose the path with lowest heuristics value , currently C has lowest value among above node . So we will go from A to C.

Greedy Best First Search Algorithm

- Example: We want to travel from A to G.



3) Now from C we have direct paths as C to F(with heuristics value of 17) and C to E(with heuristics value of 19) , so we will go from C to F.



4) Now from F we have direct path to go to the goal node G (with heuristics value of 0) , so we will go from F to G.

So now the goal node G has been reached and the path we will follow is A->C->F->G.

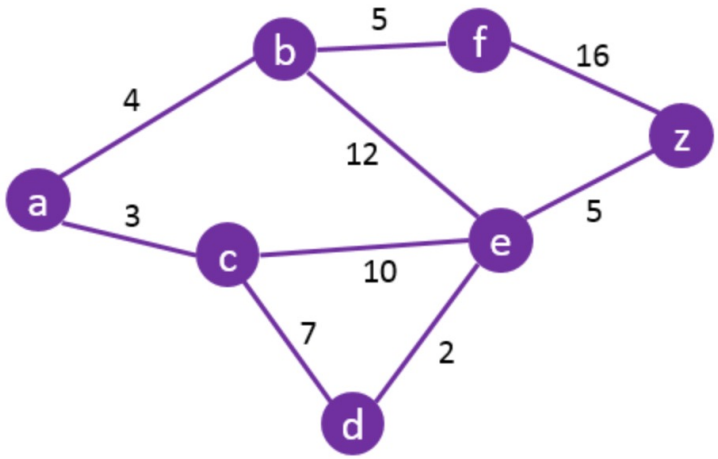
Greedy Best First Search Algorithm

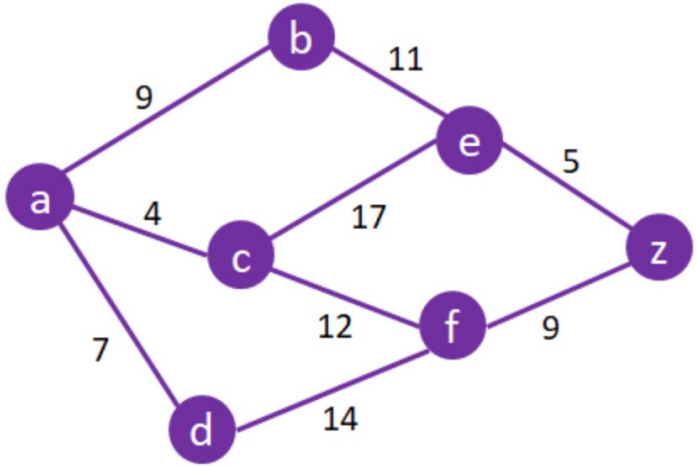
- Advantages of Greedy Best-First Search
  - Simple and Easy to Implement
  - Fast and Efficient
  - Low Memory Requirements
  - Flexible.

- Disadvantages of Greedy Best-First Search
  - Inaccurate Results
  - Local Optima
  - Heuristic Function
  - Lack of Completeness

Homework 3

- Implement the Greedy Best First Seach on the graph used in homework 2.1 and 2.2. Report the results against the results of A* search algorithm and Dijkstar's algorithm.





https://www.geeksforgeeks.org/dsa/greedy-best-first-search-algorithm/

Hill Climbing Algorithm

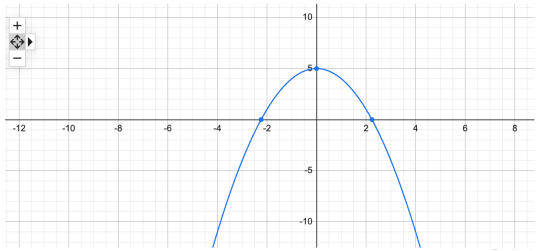
- **Terminology**
  - **Local Maximum**: A local maximum is a state better than its neighbors but not the best overall. While its objective function value is higher than nearby states, a global maximum may still exist.
  - **Global Maximum**: The global maximum is the best state in the state-space diagram where the objective function achieves its highest value. This is the optimal solution the algorithm seeks.
  - **Plateau/Flat Local Maximum**: A plateau is a flat region where neighboring states have the same objective function value, making it difficult for the algorithm to decide on the best direction to move.
  - **Ridge**: A ridge is a higher region with a slope which can look like a peak. This may cause the algorithm to stop prematurely, missing better solutions nearby.
  - **Current State**: The current state refers to the algorithm's position in the state-space diagram during its search for the optimal solution.
  - **Shoulder**: A shoulder is a plateau with an uphill edge allowing the algorithm to move toward better solutions if it continues searching beyond the plateau.

Hill Climbing Algorithm

- Example: Say we would like to solve for the maximum of a mathematical function $f(x) = -x^2 + 5$. We would like to find x and $f(x)$ where $f(x)$ is maximum.

```
Import numpy as np

def objective(x):
    return -x[0] ** 2 + 5

def generate_neighbors(x, step_size=0.1):
    return [np.array([x[0] + step_size]), np.array([x[0] - step_size])]

def hill_climbing(objective, initial, n_iterations=100, step_size=0.1):
    current = np.array([initial])
    current_eval = objective(current)
    for i in range(n_iterations):
        neighbors = generate_neighbors(current, step_size)
        neighbor_evals = [objective(n) for n in neighbors]

        best_idx = np.argmax(neighbor_evals)
        if neighbor_evals[best_idx] > current_eval:
            current = neighbors[best_idx]
            current_eval = neighbor_evals[best_idx]
            print(f"Step {i+1}: x = {current[0]:.4f}, f(x) = {current_eval:.4f}")
        else:
            print("No better neighbors found. Algorithm converged.")
            break
    return current, current_eval
```

Graph for $-x^2 + 5$




```
Step 1: x = 1.9000, f(x) = 1.3900
Step 2: x = 1.8000, f(x) = 1.7600
Step 3: x = 1.7000, f(x) = 2.1100
Step 4: x = 1.6000, f(x) = 2.4400
Step 5: x = 1.5000, f(x) = 2.7500
Step 6: x = 1.4000, f(x) = 3.0400
Step 7: x = 1.3000, f(x) = 3.3100
Step 8: x = 1.2000, f(x) = 3.5600
Step 9: x = 1.1000, f(x) = 3.7900
Step 10: x = 1.0000, f(x) = 4.0000
Step 11: x = 0.9000, f(x) = 4.1900
Step 12: x = 0.8000, f(x) = 4.3600
Step 13: x = 0.7000, f(x) = 4.5100
Step 14: x = 0.6000, f(x) = 4.6400
Step 15: x = 0.5000, f(x) = 4.7500
Step 16: x = 0.4000, f(x) = 4.8400
Step 17: x = 0.3000, f(x) = 4.9100
Step 18: x = 0.2000, f(x) = 4.9600
Step 19: x = 0.1000, f(x) = 4.9900
Step 20: x = -0.0000, f(x) = 5.0000
No better neighbors found. Algorithm converged.

Best solution x = -0.0000, f(x) = 5.0000
```

Hill Climbing Algorithm

- **Advantages of hill climbing algorithm**
  - Simplicity and Ease of Implementation
  - Versatility
  - Efficiency in Finding Local Optima
  - Customizability

Backtracking Algorithm

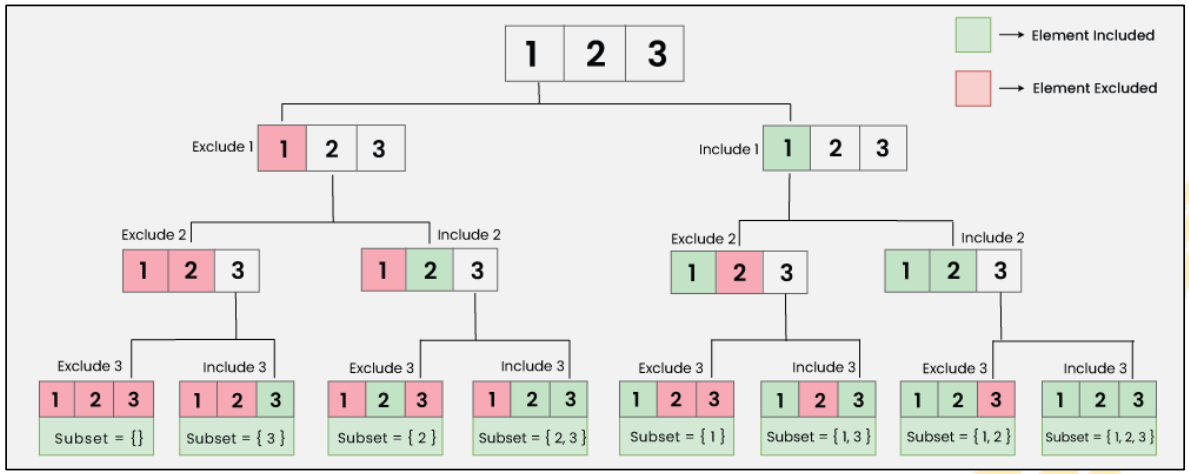
- Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly.

- Approach:
  1. Choose an initial solution.
  2. Explore all possible extensions of the current solution.
  3. If an extension leads to a solution, return that solution.
  4. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
  5. Repeat steps 2-4 until all possible solutions have been explored.

Backtracking Algorithm

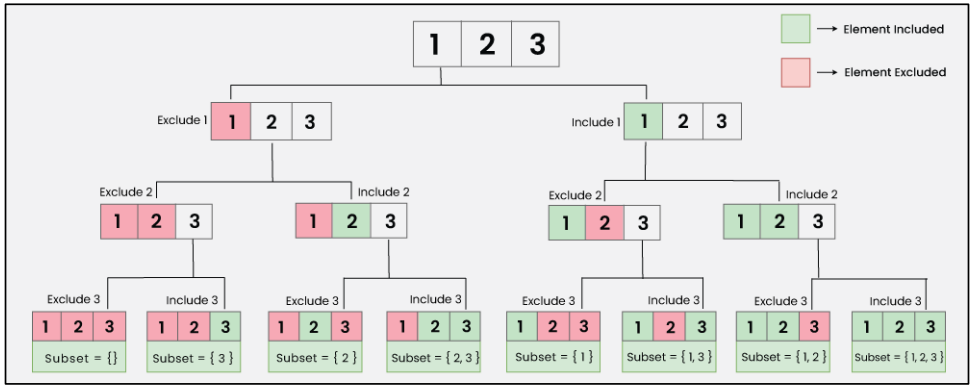
- Example: **All subsets of a given Array**
  - Given an integer array `arr[]`, find all the subsets of the array.
  - A subset is any selection of elements from an array, where the order does not matter, and no element appears more than once. It can include any number of elements, from none (the empty subset) to all the elements of the array.
    - Example: Input: arr[] = [1, 2, 3] , Output: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

- Backtracking algorithm
  - The idea is to use backtracking to explore all possible choices one by one recursively. For each element, there two options, either include it into subset or exclude it.



Backtracking Algorithm

- Example: all subsets of a given array



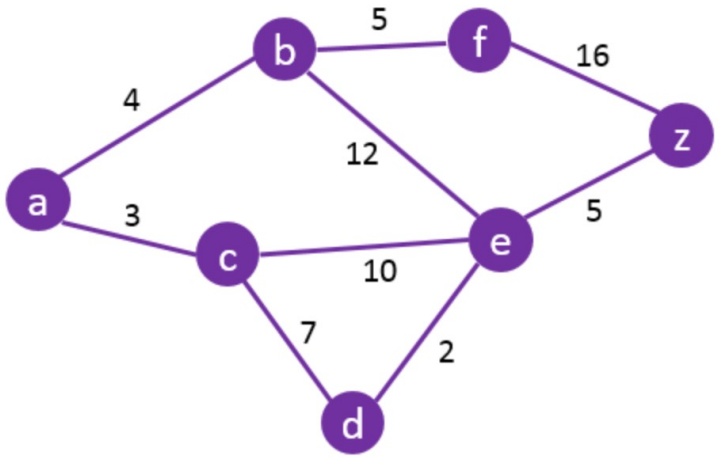
Output

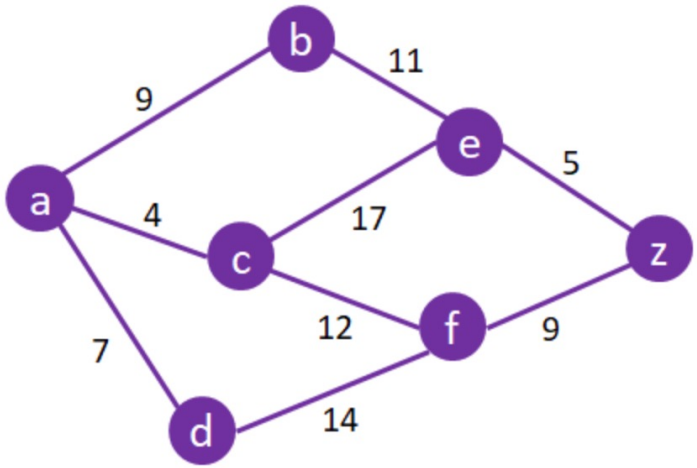
```
[1, 2, 3]
[1, 2]
[1, 3]
[1]
[2, 3]
[2]
[3]
[]
```

```python
def subsetRecur(i, arr, res, subset):

    # add subset at end of array
    if i == len(arr):
        res.append(list(subset))
        return

    # include the current value and recursively find all subsets
    subset.append(arr[i])
    subsetRecur(i + 1, arr, res, subset)

    # exclude the current value and recursively find all subsets
    subset.pop()
    subsetRecur(i + 1, arr, res, subset)

def subsets(arr):
    subset = []
    res = []

    # finding subset recursively
    subsetRecur(0, arr, res, subset)
    return res

# driver
if __name__ == "__main__":
    arr = [1, 2, 3]
    res = subsets(arr)

    for subset in res:
        print("[", end="")
        print(", ".join(str(num) for num in subset), end="")
        print("]")
```

Homework 4

- Use backtracking algorithm to print all possible paths from a source to destination using backtracking algorithm (include and exclude a node for printing). Also calculate the cost of the path. Apply your code to the graph in homework 2.1 and 2.2.
- Hint: Depth-First Search (DFS) is a form of backtracking. See https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/





Example: Sudoku Solver

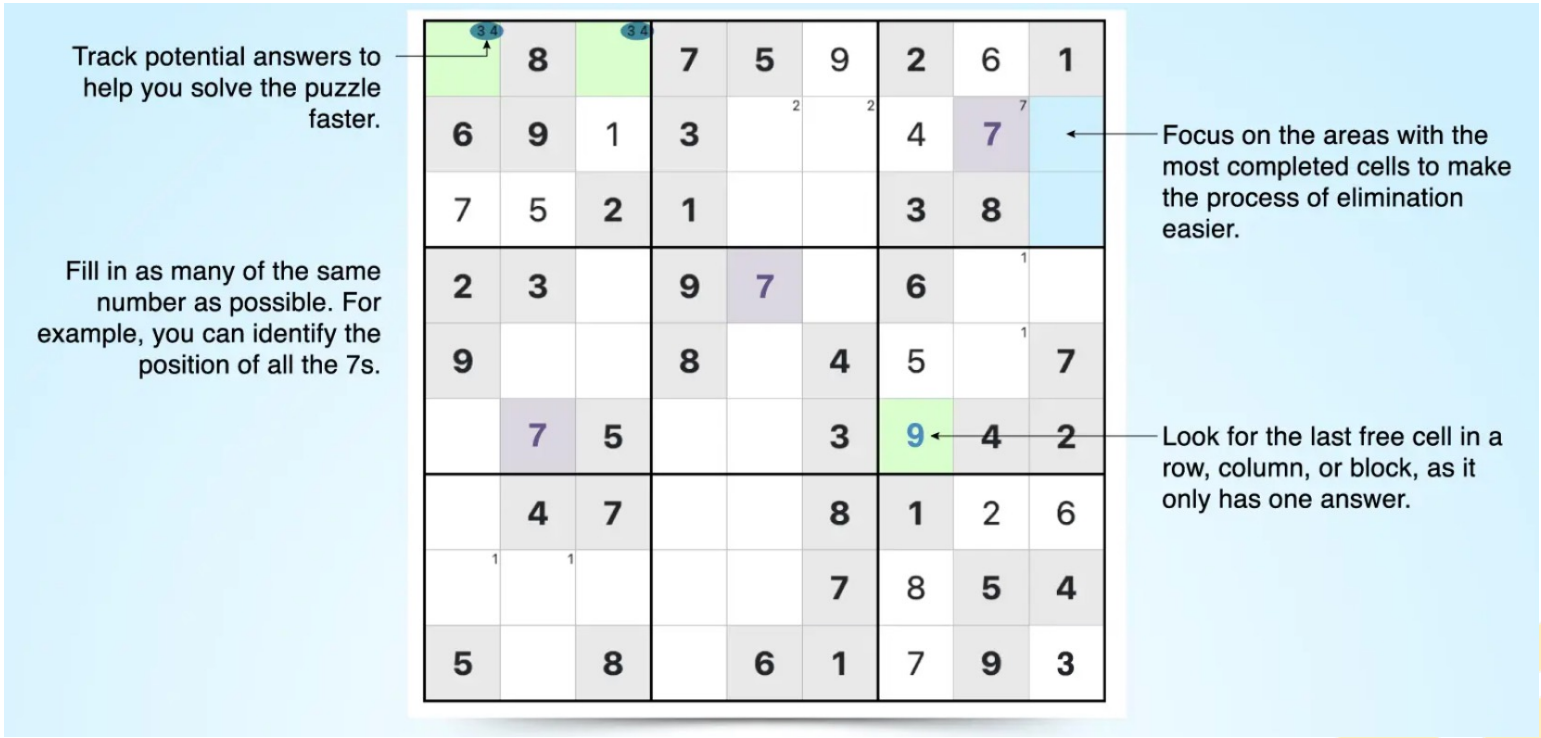
- A Sudoku is a logic-based number puzzle played on a 9x9 grid divided into nine 3x3 subgrids (boxes).
- The goal is to fill the grid so that each row, each column, and each 3x3 box contains all the numbers from 1 to 9 exactly once. Some cells are pre-filled with numbers, which serve as clues to help the player deduce the values for the empty cells.
- Play at https://sudokubliss.com/

	7			**1**	8		4	9
	3		7			6		8
		9	5					**1**
	2	**1**		5				7
		4	8		7	5		
3				4		9	8	
6					2	8		
7		5			9		2	
2	9		**1**	6			3	

5	7	6	2	**1**	8	3	4	9
1	3	2	7	9	4	6	5	8
4	8	9	5	3	6	2	7	**1**
8	2	**1**	9	5	3	4	6	7
9	6	4	8	2	7	5	**1**	3
3	5	7	6	4	**1**	9	8	2
6	**1**	3	4	7	2	8	9	5
7	4	5	3	8	9	**1**	2	6
2	9	8	**1**	6	5	7	3	4

Example: Sudoku Solver

- How to play Sudoku
  - https://sudokubliss.com/guides/how-to-play
  - https://sudoku.com/how-to-play/sudoku-rules-for-complete-beginners/
  - https://www.nytimes.com/2023/03/02/crosswords/how-to-solve-sudoku.html
  - https://seasonsretirement.com/how-to-play-sudoku/



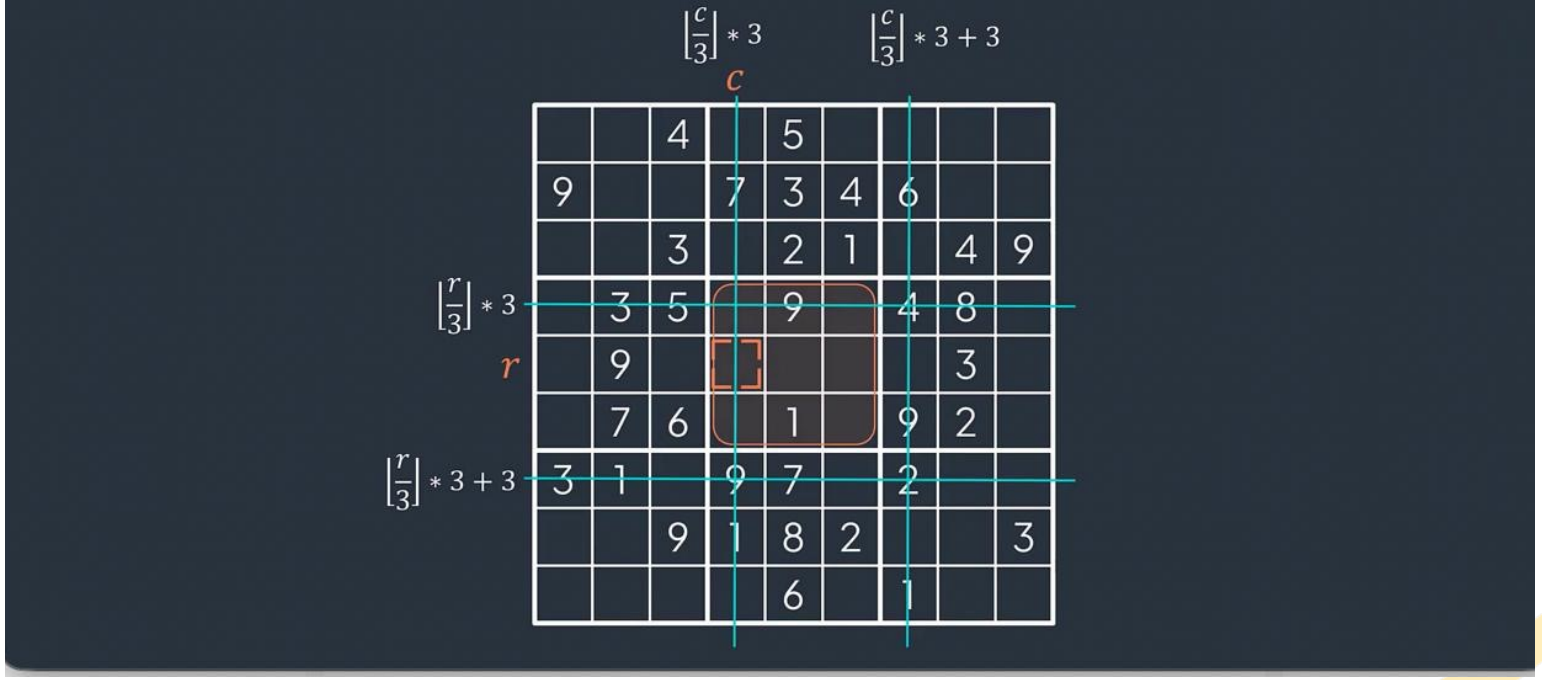
Example: Sudoku Solver

- **Backtracking heuristic** to solve a Sudoku puzzle
  - Starting from row 0 and column 0 (0,0)
  - Try to input a number from the set {1, 2, …, 9} into the cell
  - See if the number creates any conflict (is the number valid in the cell position?)
  - If the number is valid in the cell position, then move on to the next cell to the right. If the current cell is in the last column, move to the next row. If the end of the row is reached, then finish the operation.
  - In programming, we need to represent the puzzle as a 2D array.

Example: Sudoku Solver

- Is the number that we try valid for the current cell?

```
def is_valid(grid, r, c, k):
    not_in_row = k not in grid[r]
    not_in_column = k not in [grid[i][c] for i in range(9)]
    not_in_box = k not in [grid[i][j] for i in range(r//3*3, r//3*3+3) for j in range(c//3*3, c//3*3+3)]
```



Example: Sudoku Solver

- Solving the puzzle

```
def solve(grid, r=0, c=0):
    if r == 9:
        return True
    elif c == 9:
        return solve(grid, r+1, 0)
    elif grid[r][c] != 0:
        return solve(grid, r, c+1)
    else:
        for k in range(1, 10):
            if is_valid(grid, r, c, k):
                grid[r][c] = k
                if solve(grid, r, c+1):
                    return True
                grid[r][c] = 0
        return False
```

Move on if there is a number (k) that we can place in the current position.

Backtracking if there is no number (k) that we can place in the next position.

Homework 5: Sudoku Solver

- See animation at https://youtu.be/tapjBw9Mab0

- Code a backtracking algorithm that solves a sudoku puzzle (tested with at least 10 examples from https://sudokubliss.com/ with easy, medium, hard, and expert level).
- Check your results against the solutions from the website.
- Plot the histogram of the time it takes for each difficulty level and calculate the average solving time.