Heuristics and Greedy Algorithms

Outline

- Greedy Heuristics
- Minimum Spanning Tree
    - Kruskal’s algorithm
    - Prim’s algorithm
- Shortest Path Problem
    - Dijkstra’s algorithm
- Knapsack Problem

Definitions

- **Algorithms** refer to a finite sequence of well-defined instructions typically used to **solve** (i.e. find the correct answer) a class of problems

- **Heuristics** are quick methods for finding an approximate solution, trading optimality, completeness, accuracy, or precision for speed

- **Greedy algorithms/heuristics** are those than make a locally optimal choice at each step (hoping to approximate a globally optimal solution)

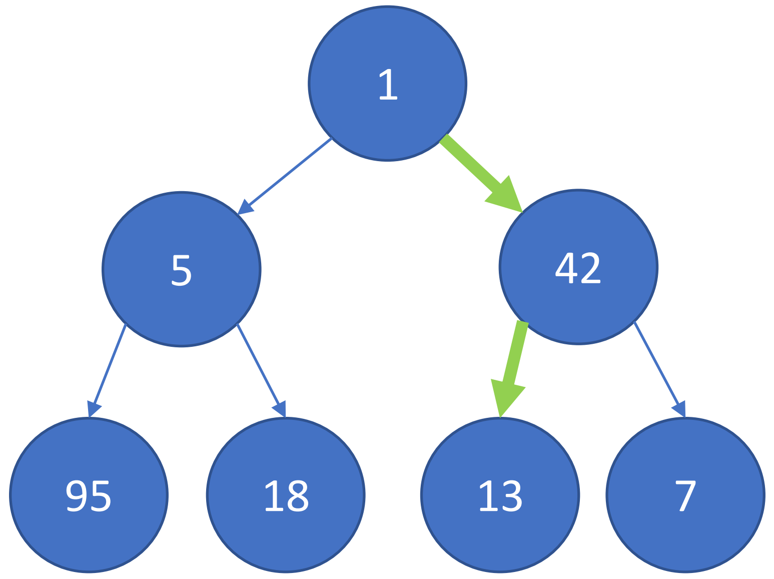
Evaluating a Heuristic

- **Expected performance**- given some large set of samples with known solutions, what is the expected difference (%) between the optimal and heuristic solution. Can be calculated on:
  - Randomly generated problems
  - Real-world/Benchmark problems
- **Worst-case performance**- given an adversarially constructed problem what is the worst performance possible
- **Nth Percentile** – when rank ordering performance vs optimal best to worst, what is the nth percentile
- Runtime, additional outputs, …

NP-hard Problems

- Recall that some problems are easy to check a solution, but hard to find the best solution
- This can make it difficult to know how well your solution is performing relative to what is possible
- Heuristics can help provide reasonable solutions that can be used as baselines when comparing more advanced solutions

Greedy

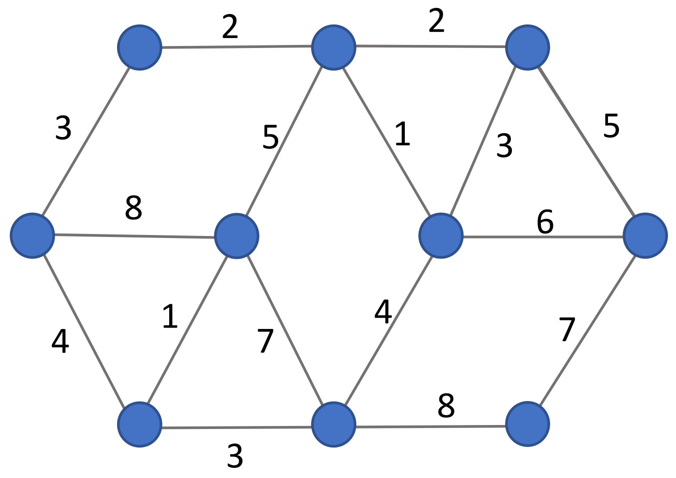


- Greedy algorithms make the choice that is best for now, ignoring future choices
- Greedy algorithms can perform arbitrarily poorly in general, but provide guaranteed optimality for some types of problems

Greedy Algorithms

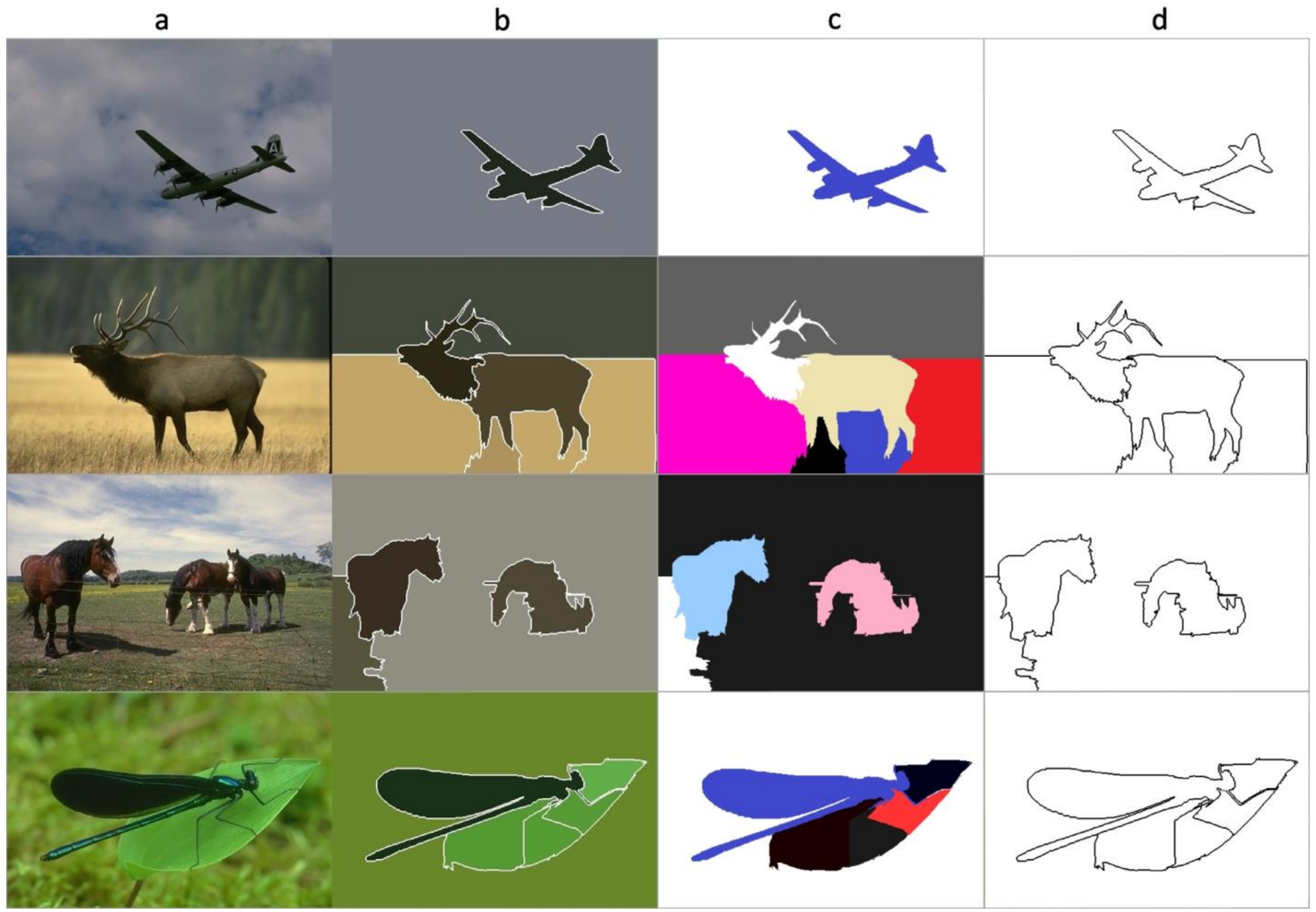
1. **Candidate set**, choices from which a solution is created
2. **Selection function**, chooses the best candidate to be added to the solution
3. **Feasibility test**, determines if a candidate move is feasible
4. **Objective function**, assigns a value to a partial or complete solution
5. **Stopping criteria,** indicates when we have discovered a complete solution

Minimum Spanning Tree



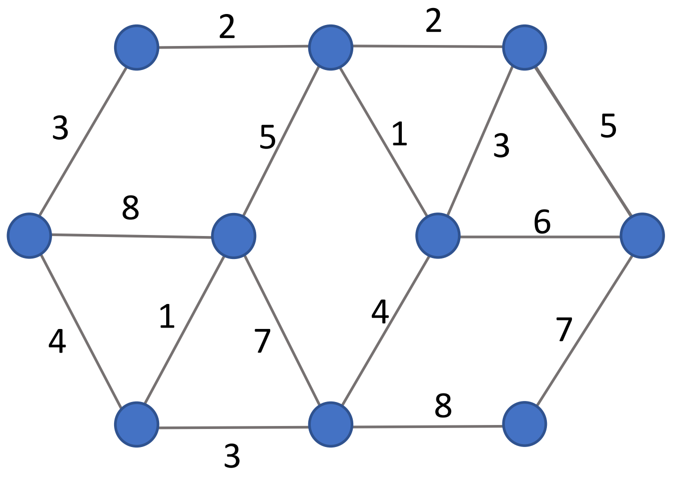
- A tree is an undirected graph in which any two vertices are connected by exactly one path (i.e. there are no cycles)
- The minimum spanning tree is the tree that connects every vertex with the minimum total of weighted arcs

Minimum Spanning Tree



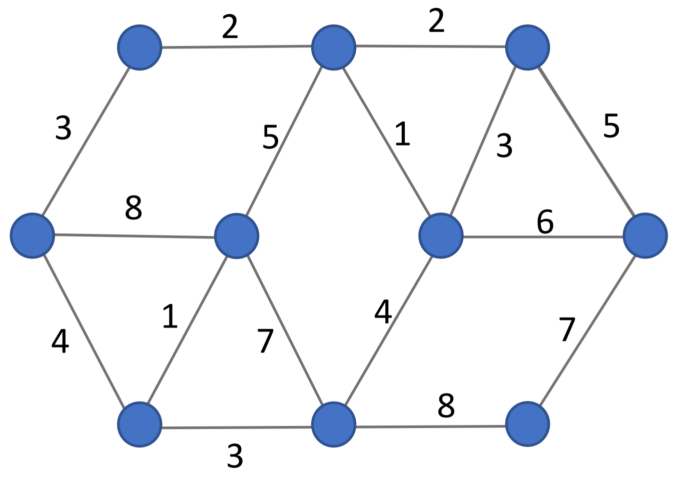
- Min spanning trees have applications in network design (computer, telecommunication, etc.)
- Appear as a sub problem for heuristics for NP-hard problems (including the traveling salesman problem)
- Image segmentation
- Cluster analysis

Greedy Algorithm 1



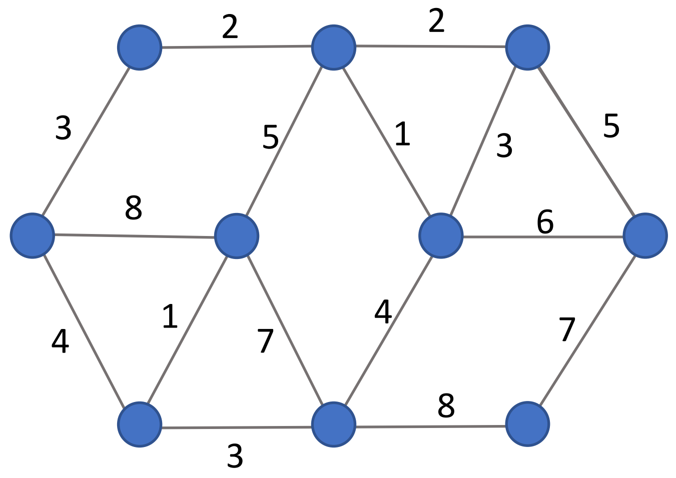
- Start from an arbitrary node and build a spanning tree from there
  - **Candidate Set**: Arcs connected to nodes in the current solution
  - **Selection Function**: Arc with minimum cost
  - **Feasibility Test**: Does it connect new node (otherwise creates cycle)
  - **Objective Function**: Total weights of arcs
  - **Stopping Criteria**: Connect all Nodes

Greedy Algorithm 1



- Start from an arbitrary node and build a spanning tree from there
  - **Candidate Set**: Arcs connected to *exactly one node* in the current solution
  - **Selection Function**: Arc with minimum cost
  - **Feasibility Test**: All trees generated in this way are feasible
  - **Objective Function**: Total weights of arcs
  - **Stopping Criteria**: Connect all Nodes

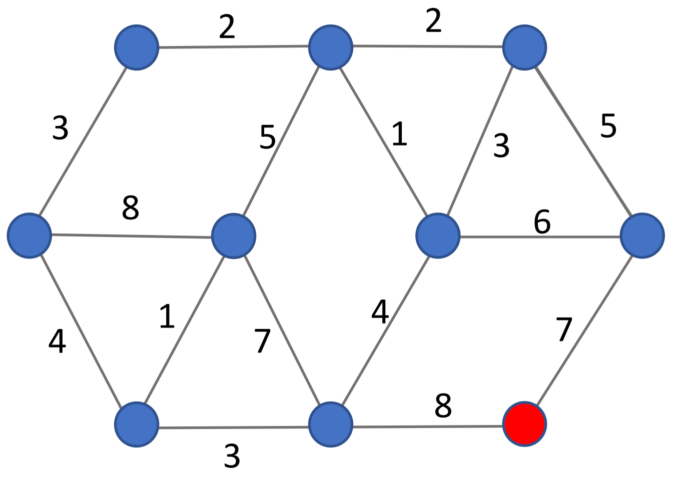
Prim’s Algorithm



- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

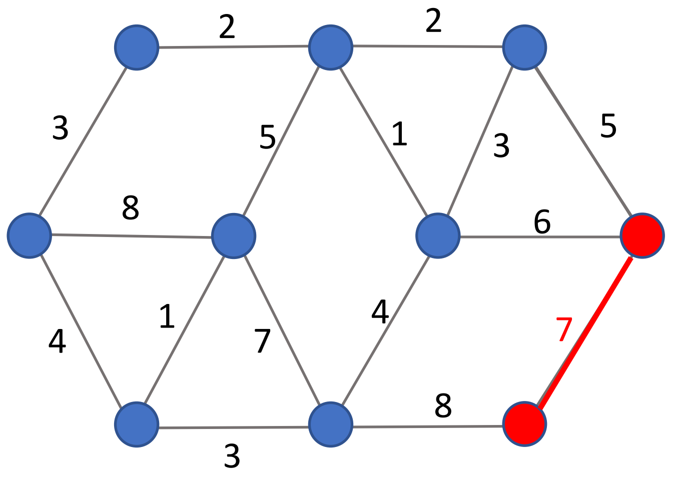
Prim’s Algorithm

- **Step 1**: Choose arbitrary node



Objective Value: 0

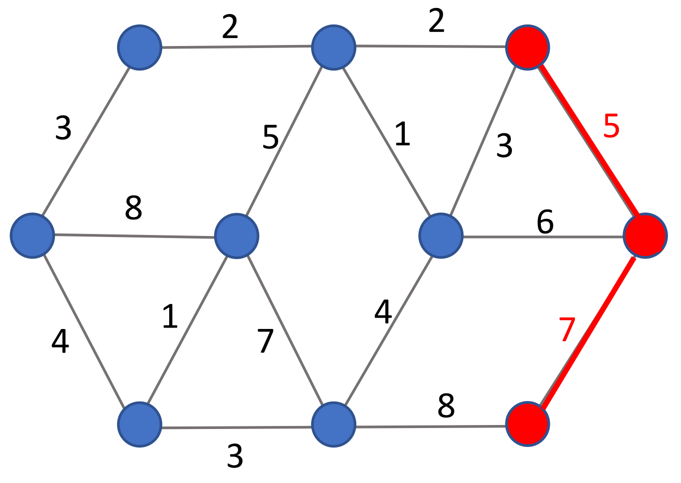
Prim’s Algorithm



Objective Value: 7

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree

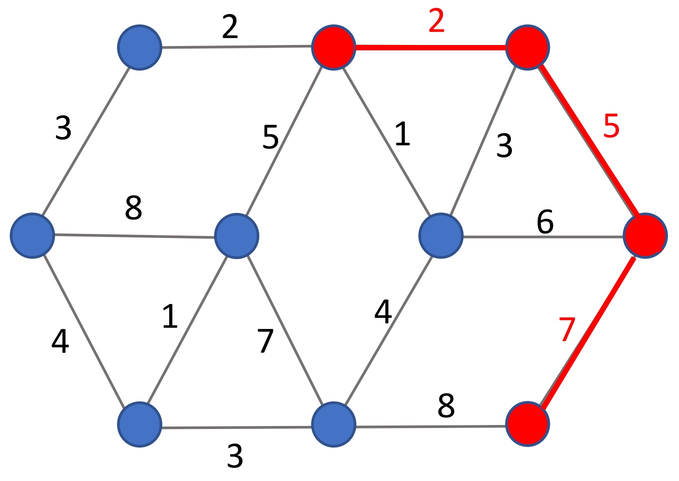
Prim's Algorithm



Objective Value: 12

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

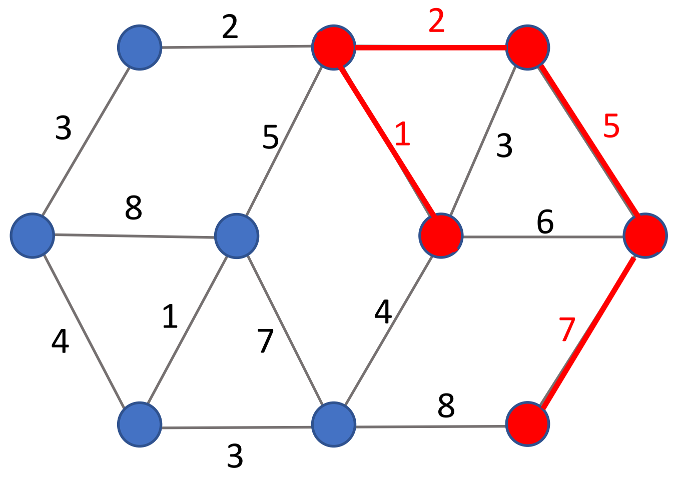
Prim's Algorithm



Objective Value: 14

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

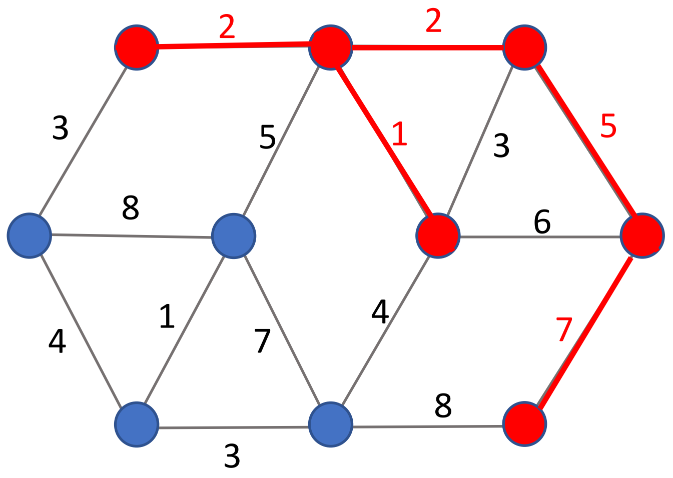
Prim’s Algorithm



Objective Value: 15

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

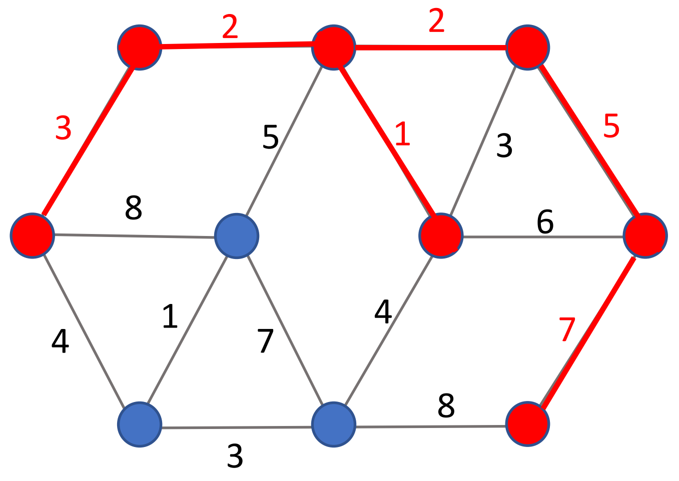
Prim’s Algorithm



Objective Value: 17

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

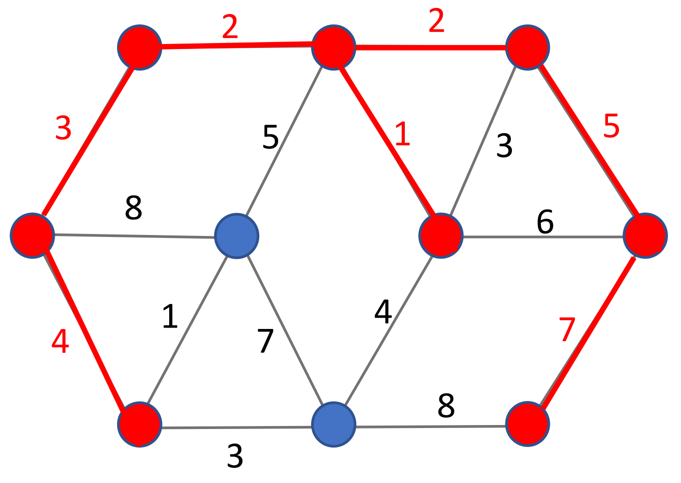
Prim’s Algorithm

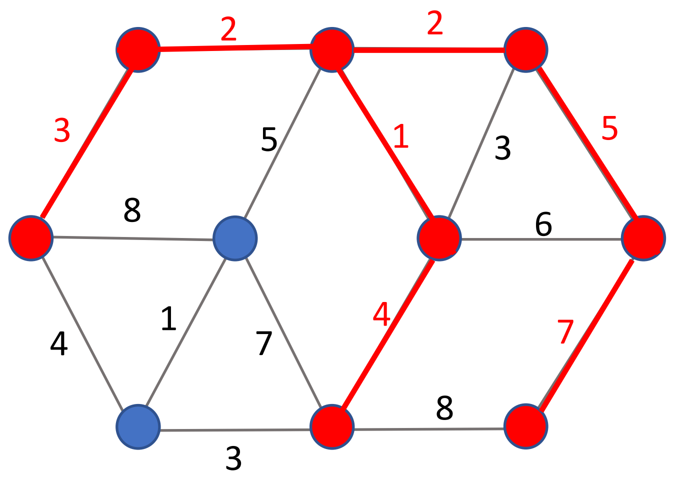


Objective Value: 20

- **Step 1**: Choose arbitrary node
- **Step 2**: Add the min weighted arc to the tree that connects from a node in the tree to a node not in the tree
- **Step 3**: Repeat step 2 until all nodes are in the tree

Prim’s Algorithm

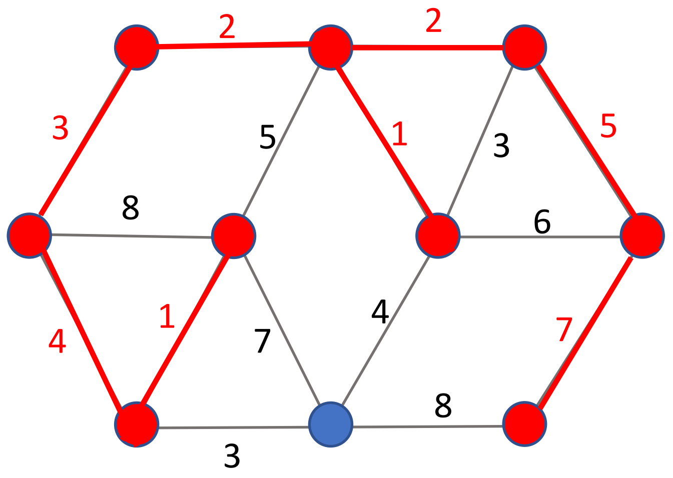




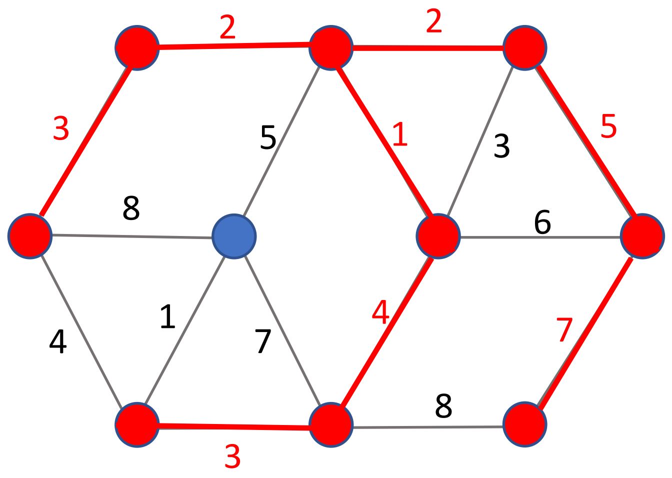
Objective Value: 24

Objective Value: 24

Prim’s Algorithm

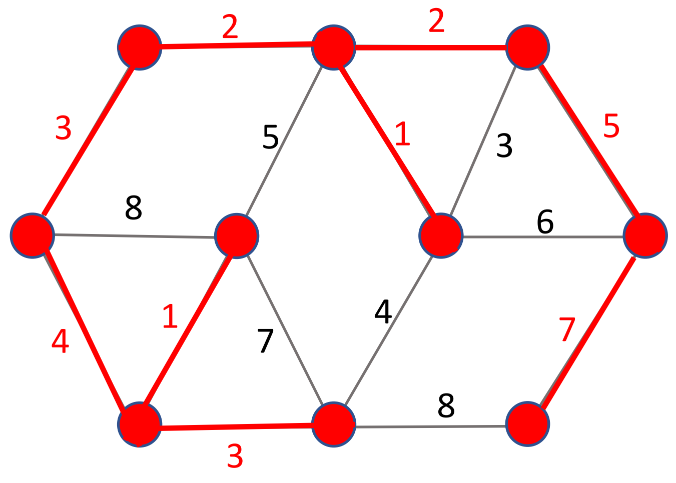


Objective Value: 25

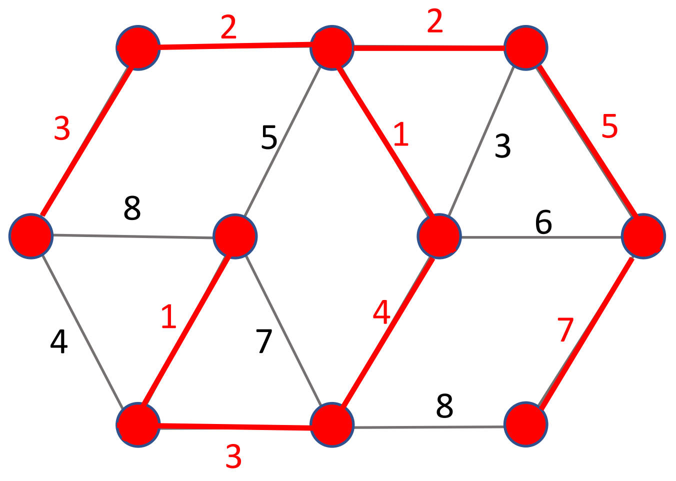


Objective Value: 27

Prim’s Algorithm

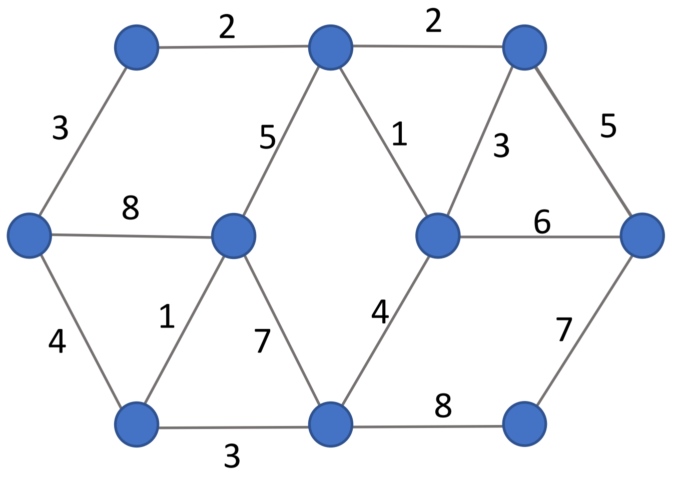


Objective Value: 28



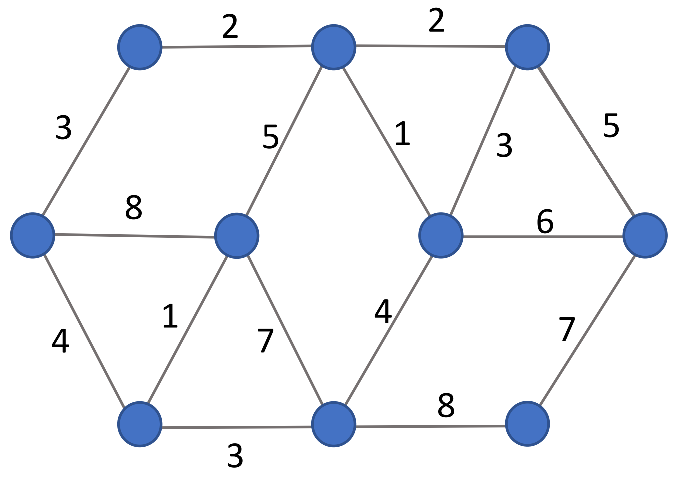
Objective Value: 28

Greedy Algorithm 2



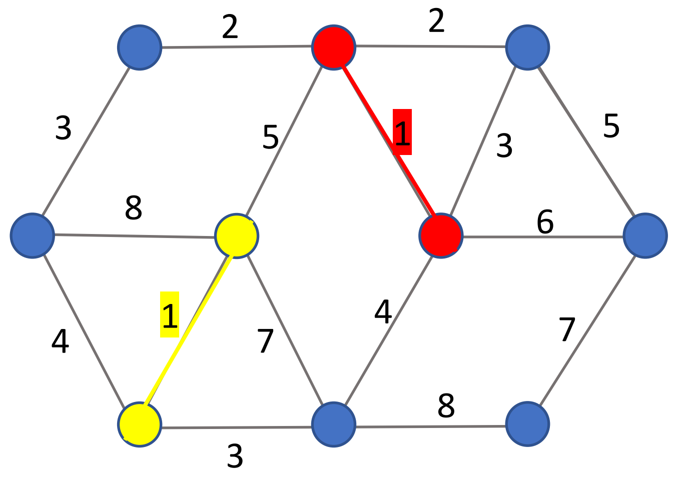
- Start from an arbitrary node and build a spanning tree from there
  - **Candidate Set**: Arcs not yet in a spanning tree
  - **Selection Function**: Arc with minimum cost
  - **Feasibility Test**: Does it create a cycle
  - **Objective Function**: Total weights of arcs in all spanning trees
  - **Stopping Criteria**: Connect all Nodes in single spanning tree

Kruskal’s Algorithm



- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
    - remove an edge with minimum weight from *S*
    - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

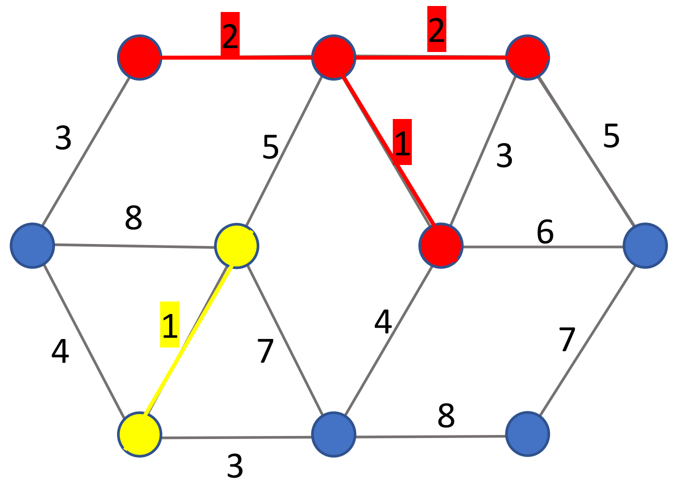
Kruskal’s Algorithm



Objective Value: 2

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
  - remove an edge with minimum weight from *S*
  - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

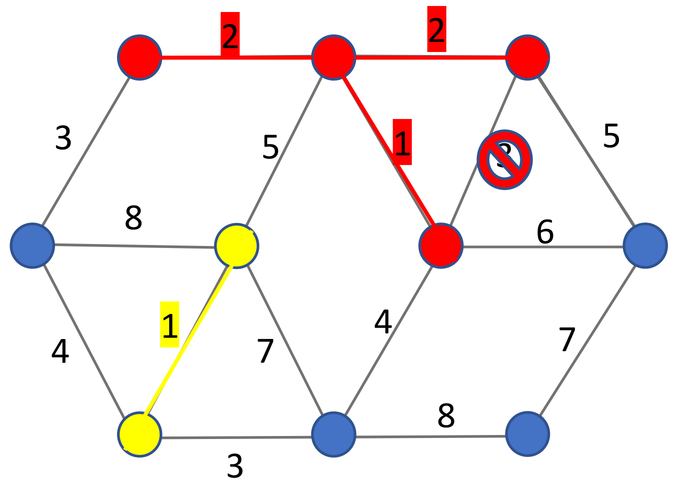
Kruskal’s Algorithm



Objective Value: 6

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
    - remove an edge with minimum weight from *S*
    - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

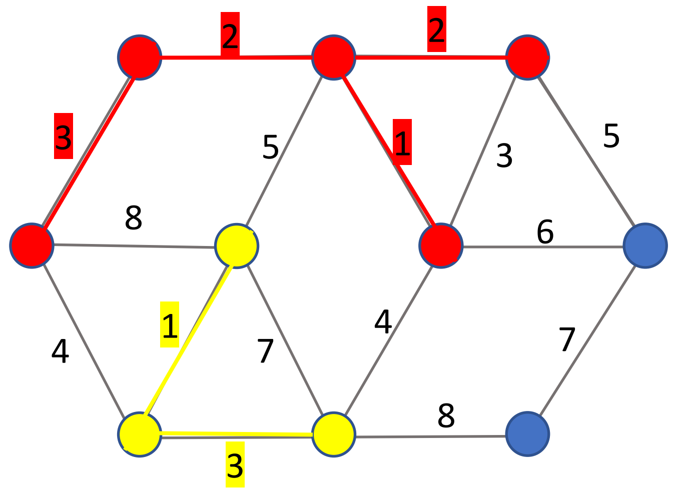
Kruskal's Algorithm



Objective Value: 6

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
  - remove an edge with minimum weight from *S*
  - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

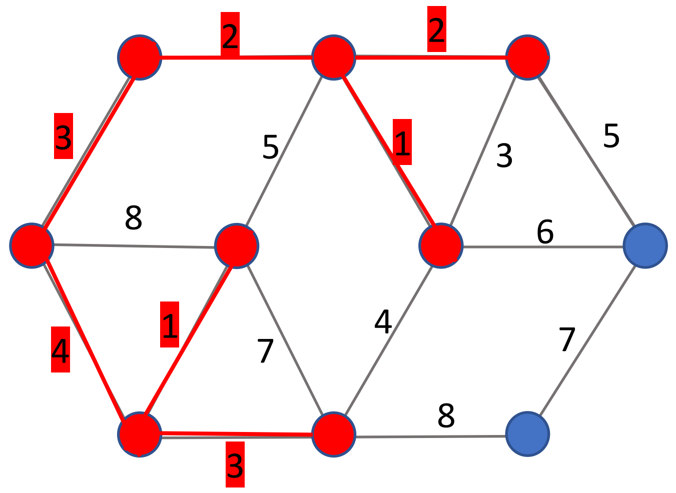
Kruskal’s Algorithm



Objective Value: 12

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
  - remove an edge with minimum weight from *S*
  - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

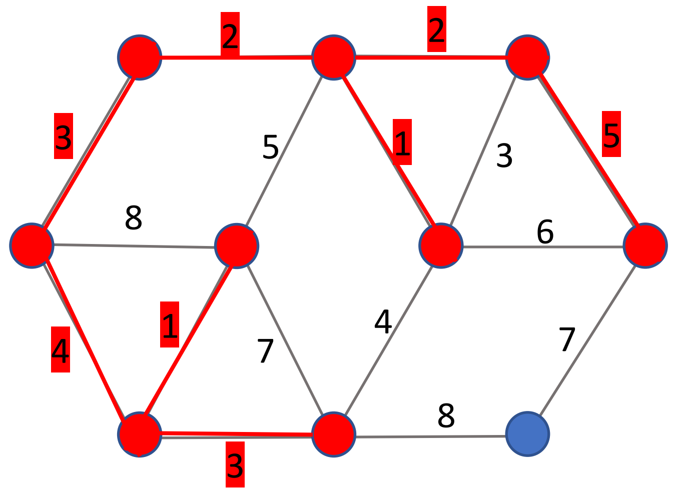
Kruskal’s Algorithm



Objective Value: 16

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
    - remove an edge with minimum weight from *S*
    - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

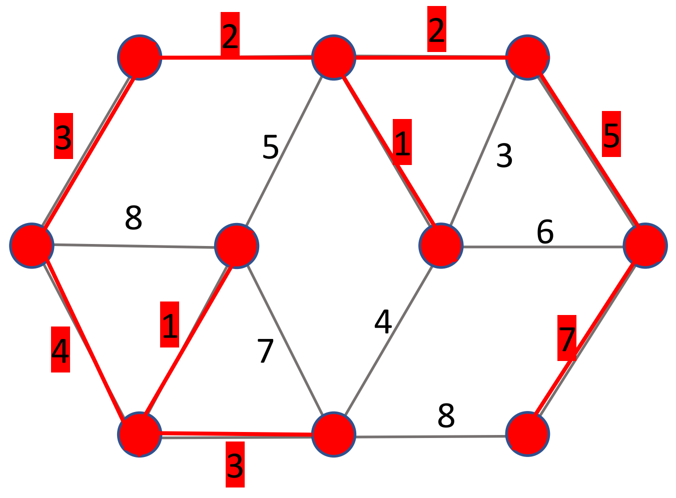
Kruskal’s Algorithm



Objective Value: 21

- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
  - remove an edge with minimum weight from *S*
  - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

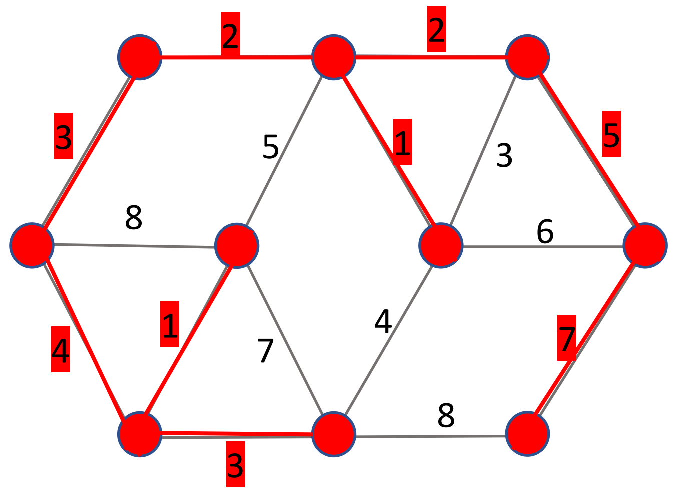
Kruskal’s Algorithm



Objective Value: 28

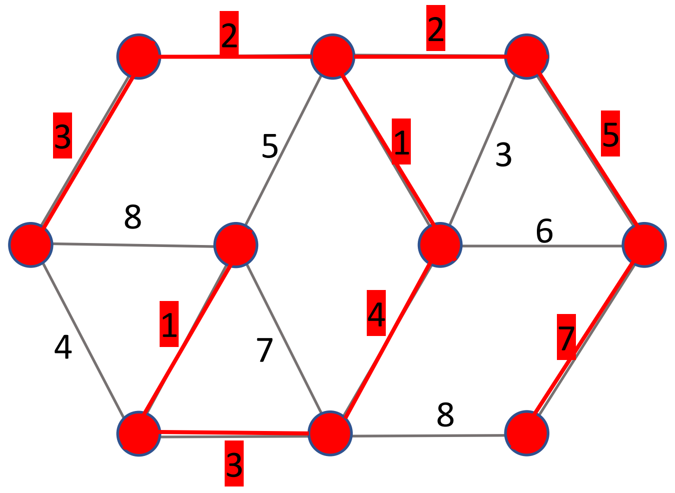
- Create a forest *F* (a set of trees), where each vertex in the graph is its own tree
- Create a set *S* of all the edges in the graph
- **while** *F* is not yet spanning (i.e. contains more than one tree)
    - remove an edge with minimum weight from *S*
    - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

Kruskal’s Algorithm



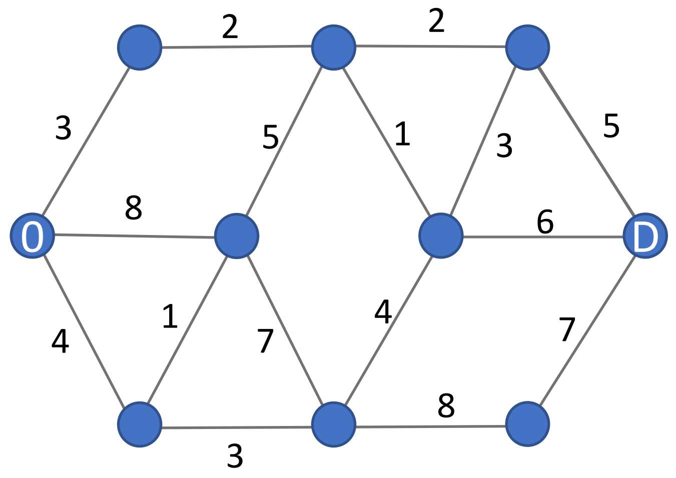
Objective Value: 28

Alternate Solution:



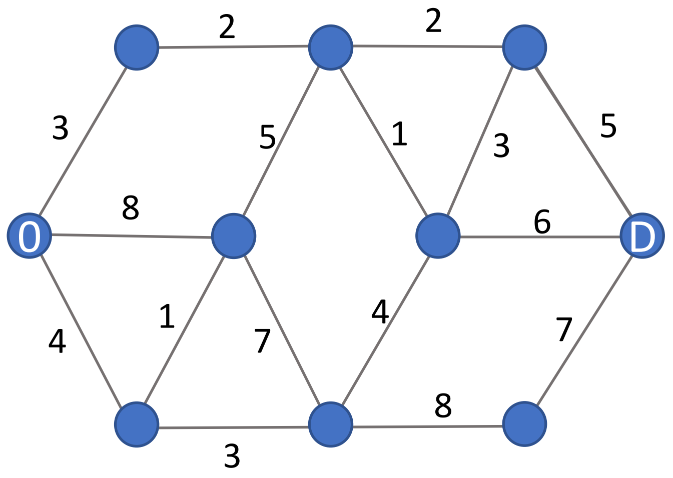
Objective Value: 28

Shortest Path Problem



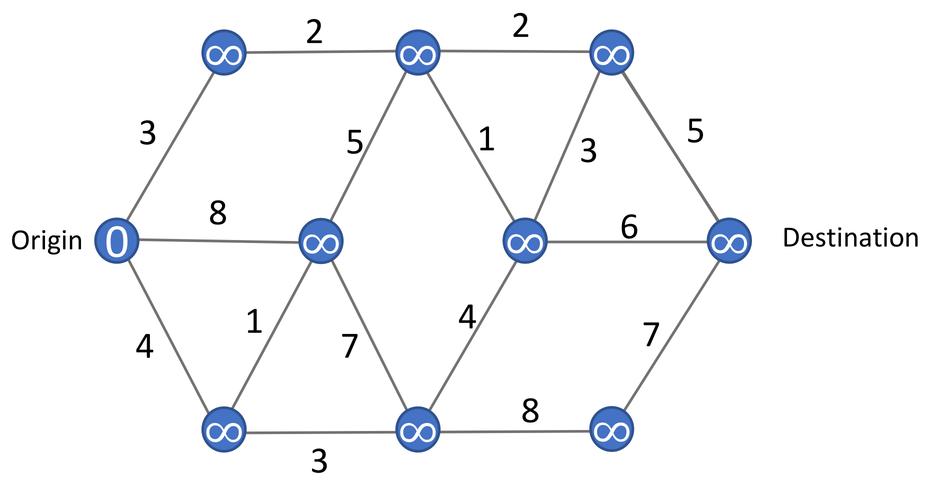
- Is there a faster way to solve the shortest path problem than to solve a linear program?
- What is a Greedy Algorithm solution going to look like?

Dijkstra’s Algorithm



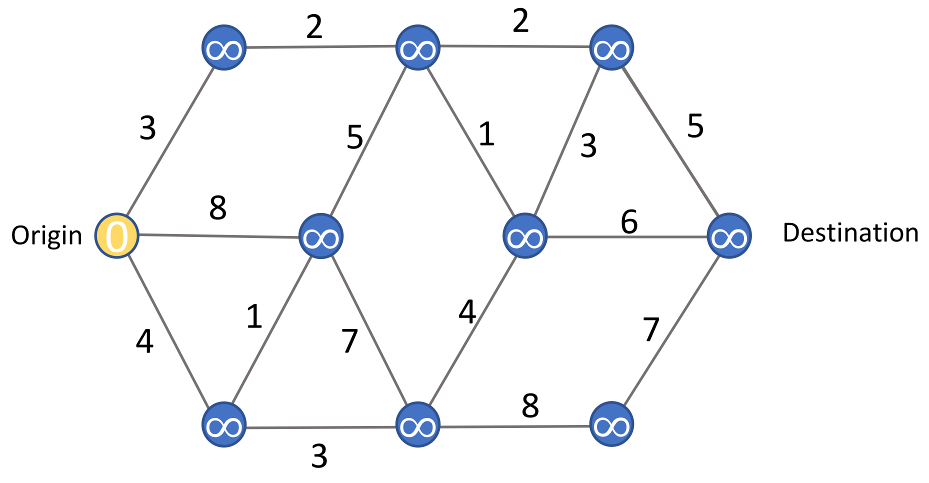
1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all nodes to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

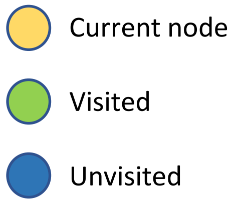
Dijkstra’s Algorithm



1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all nodes to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

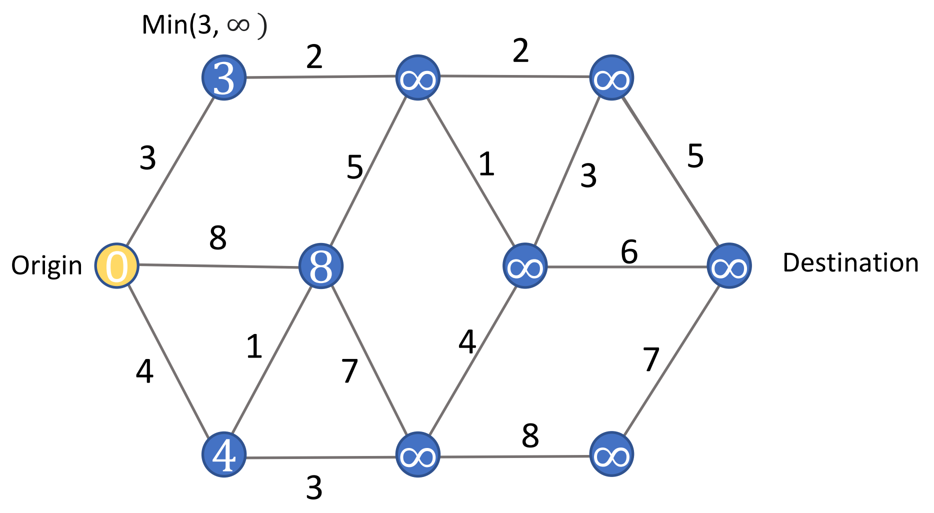
Dijkstra's Algorithm





1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all nodes to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra's Algorithm



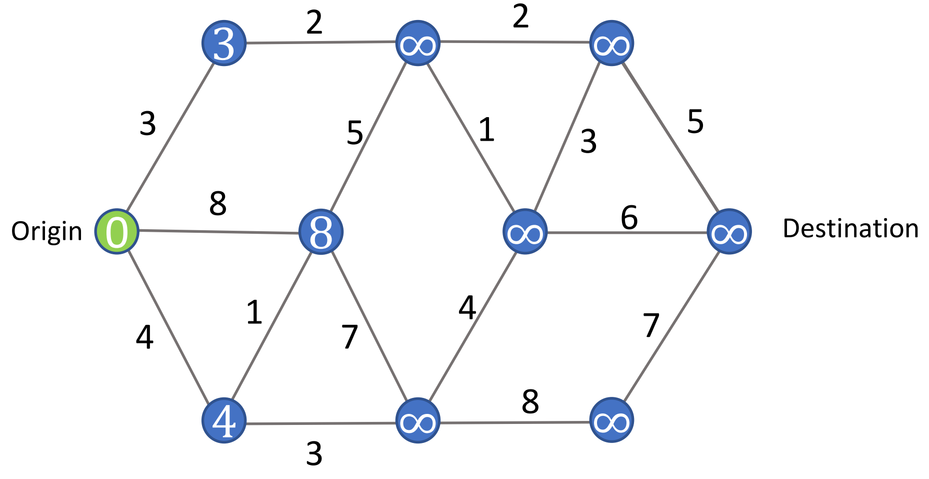
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all nodes to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



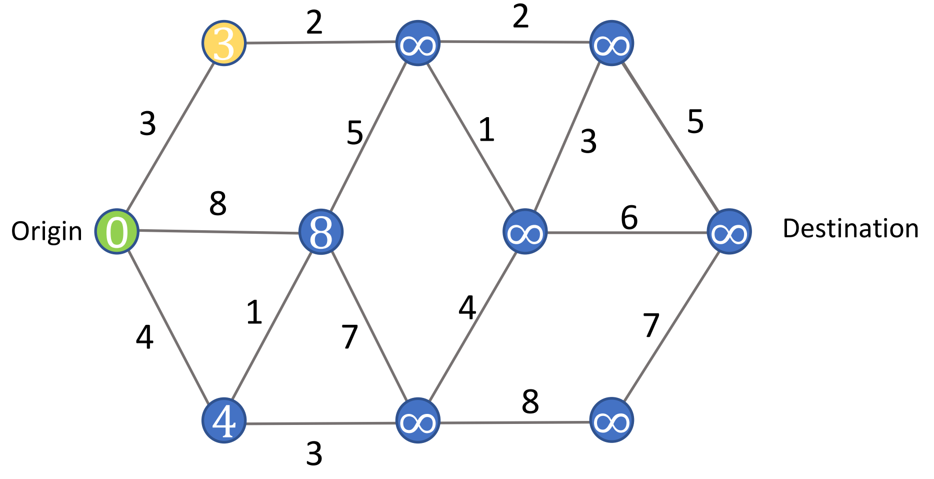
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. ➡ Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



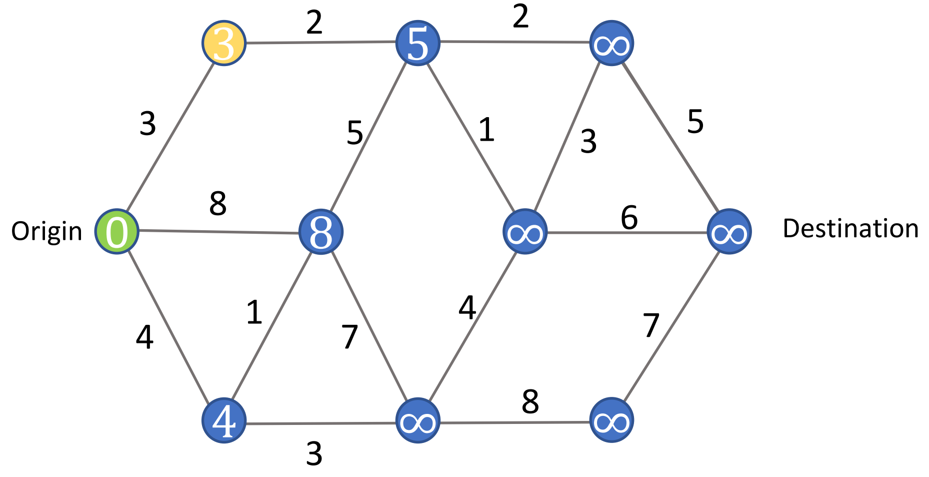
● Current node

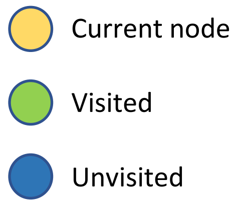
● Visited

● Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

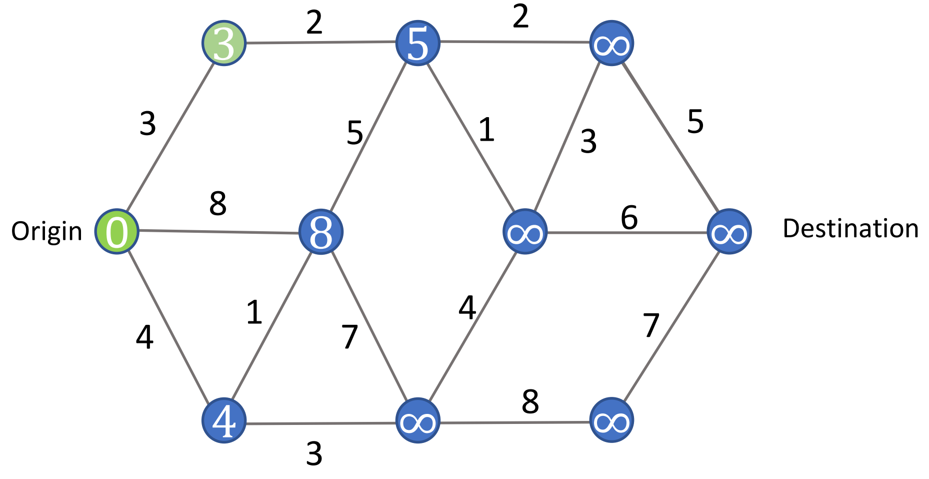
Dijkstra’s Algorithm





1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



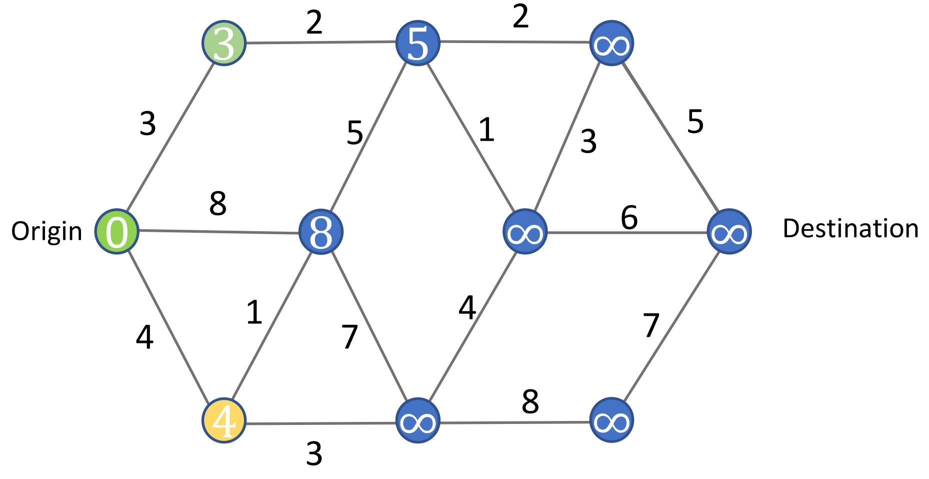
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



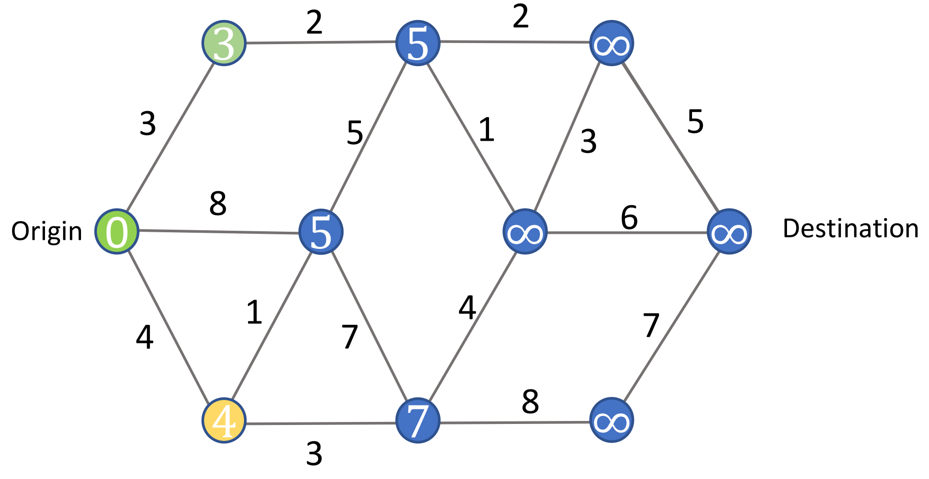
● Current node

● Visited

● Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



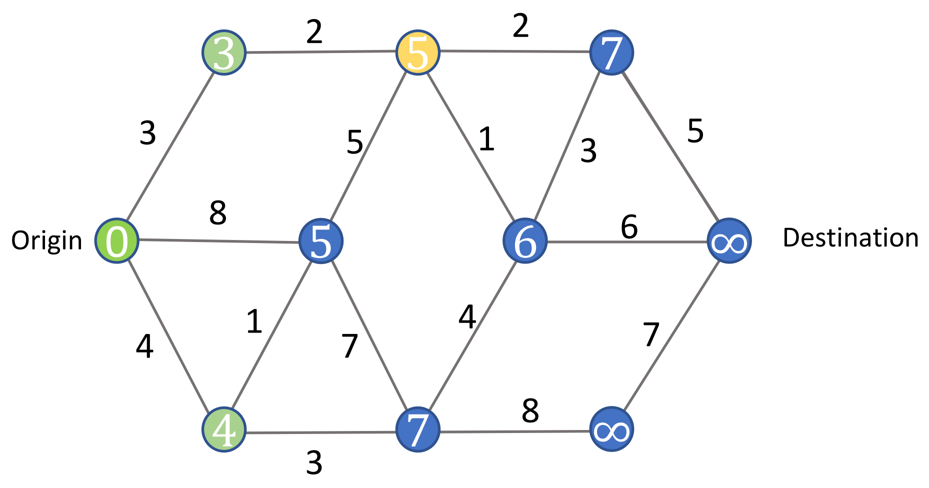
● Current node

● Visited

● Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



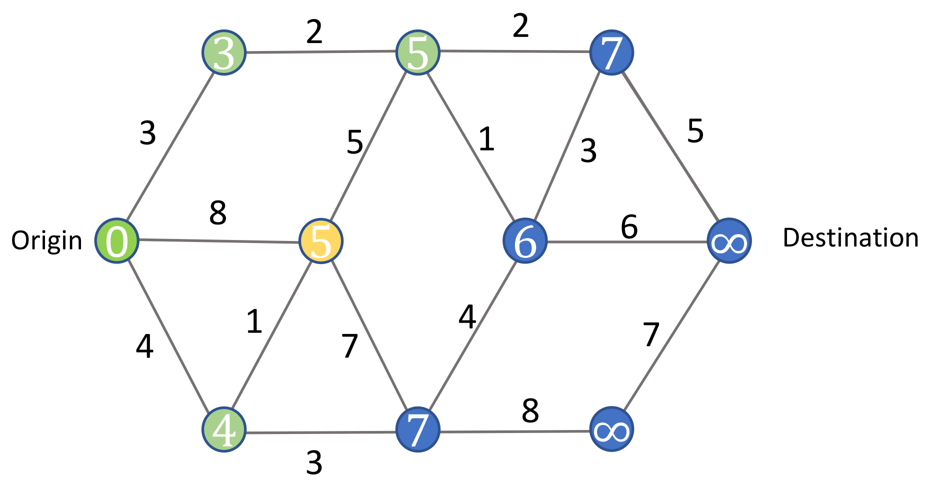
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



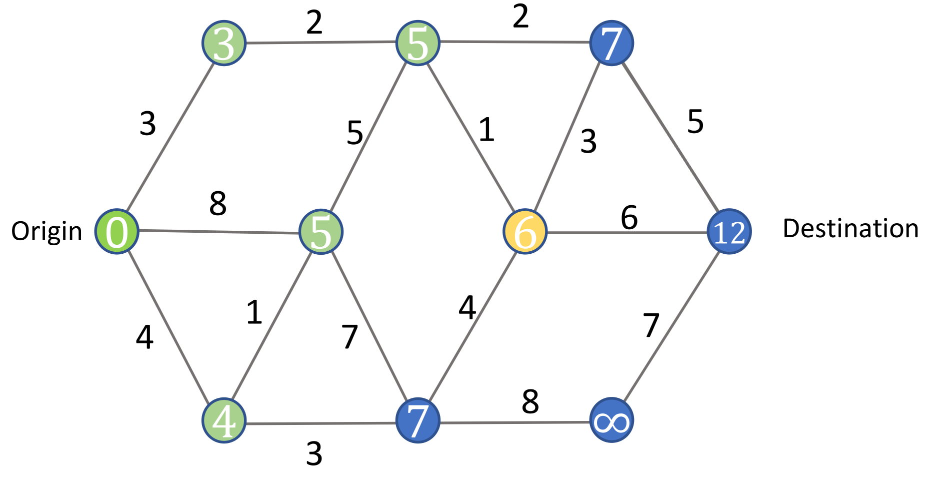
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



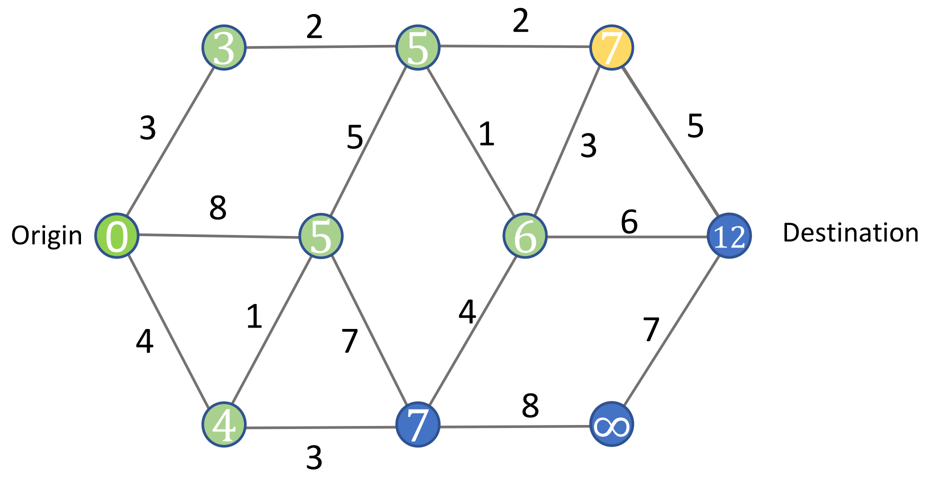
🟡 Current node

🟢 Visited

🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra’s Algorithm



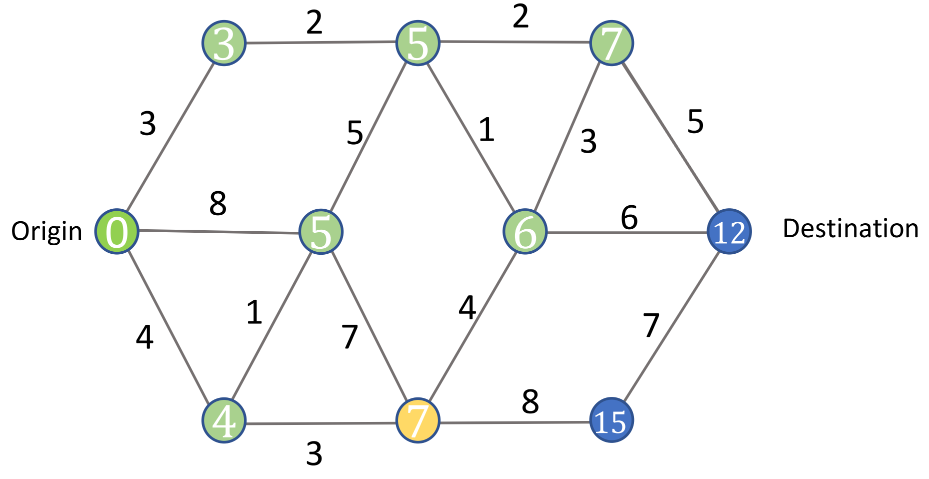
● Current node

● Visited

● Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra's Algorithm



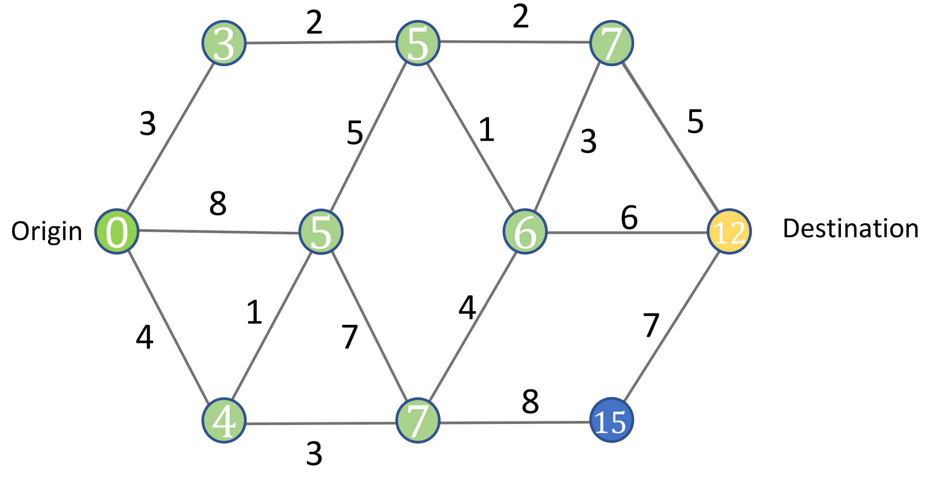
● Current node

● Visited

● Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Dijkstra's Algorithm



🟡 Current node

🟢 Visited

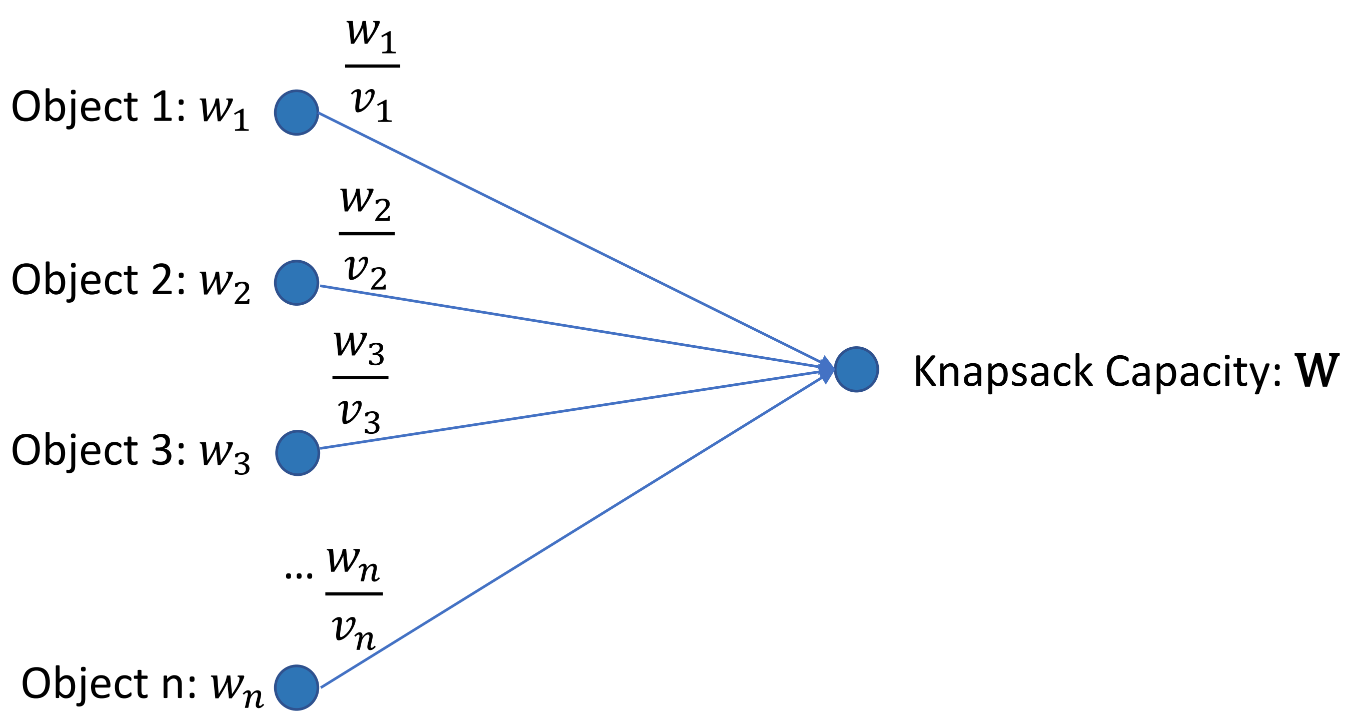
🔵 Unvisited

1. Give every node a temporary distance value:
   * 0 for the Origin
   * ∞ for all other nodes
2. Add all to an Unvisited List and set the Origin as the current node.
3. For the current node, consider all unvisited neighbors and calculate a temporary distances through the current node. Compare the newly calculated distance to its current assigned temporary distance and assign the minimum.
4. Mark the current node as visited and remove it from the Unvisited List.
5. If the destination node has been marked visited or if the smallest temporary distance among the unvisited nodes is ∞, then stop. Else, select the unvisited node that has the minimum temporary distance, set it as the new current node, and go to step 3.

Knapsack Problem

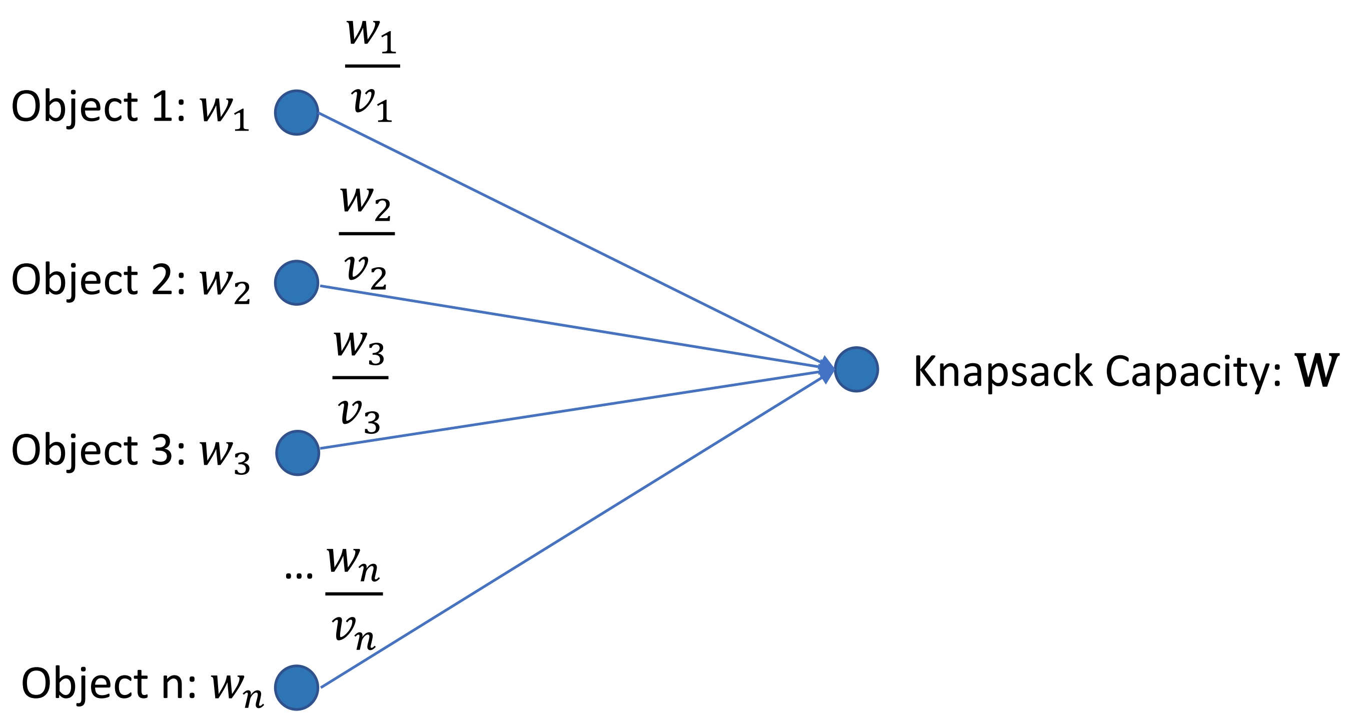
- The knapsack problem is a combinatorial optimization problem where:
  - the objective is to maximize the amount of valuable items you can fit in a knapsack
  - each item has a weight and value
  - the knapsack has a limited weight capacity

Knapsack Problem



- The knapsack problem can be thought of as a transportation problem:
  - Demand = Weight Capacity
  - Supply = Object Weights
  - Cost = Object Value/Weight
  - Objective = Maximize Cost

Knapsack Problem



- In the continuous knapsack problem you simply choose the items with the highest value “density” (i.e. Value/Weight)
- However, this only works if you can divide items

Knapsack Problem

- Knapsack capacity: 100 lbs
- Item 1:
  - 51 lbs
  - \$52
- Item 2:
  - 50 lbs
  - \$50
- Item 3:
  - 50 lbs
  - \$49

- Continuous solution:
  - 51 lbs Item 1 + 49 lbs Item 2
  - $\$52 + \$50 \times \frac{49}{50} = \$101$
- Discrete Greedy solution:
  - 51 lbs Item 1
  - $\$52$
- Discrete Optimal solution:
  - 50 lbs Item 2 + 50 lbs Item 3
  - $\$50 + \$49 = \$99$

Knapsack Worst Case Scenario

- Knapsack capacity: 100 lbs
- Item 1:
  - ε lbs
  - $ ε
- Item 2:
  - 100 lbs
  - $100 - ε

- Item 1 has a value “density” of 1, while Item 2 < 1
- The ratio between the optimal and greedy solutions can be arbitrarily large

- $\lim_{\varepsilon \to 0} \frac{100-\varepsilon}{\varepsilon} = \infty$