7️⃣

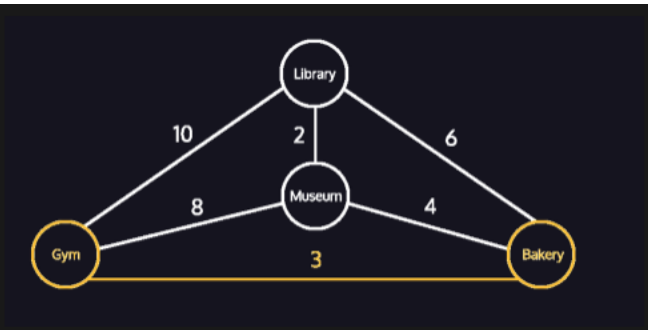
007 - Graphs

Graphs are the perfect data structure for modeling networks, which make them an indispensable piece of your data structure toolkit. They're composed of nodes, or *vertices*, which hold data, and *edges*, which are a connection between two vertices. A single node is a *vertex*.

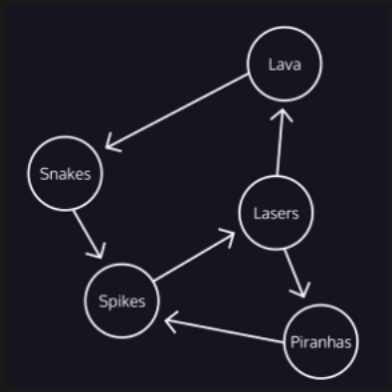
graph G = (V, E), where V is the set of vertices and E is the set of edges. Each edge in E defines a connection between two vertices (u, v), where u and v belong to the set of vertices V.

Definitions

- **Adjacent** - Given two nodes A and B. B is adjacent to A if there is a connection from A to B. In a digraph if B is adjacent to A, it doesn't mean that A is automatically adjacent to B.
- **Edge weight/edge cost** - A value associated with a connection between two nodes
- **Path** - A ordered sequence of vertices where a connection must exist between consecutive pairs in the sequence.
- **Simple path** - Every vertex in path is distinct
- **Path length** - The number of the edges in a path.
- **Cycle** - A path where the starting and ending node is the same
- **Strongly connected** - If there exists some path from every vertex to every other vertex, the graph is strongly connected.
- **Weakly connected** - If we take away the direction of the edges and there exists a path from every node to every other node, the digraph is weakly connected.
- **Weighted graph** - The edges have a number or cost associated with traveling between the vertices.



- Directed graph - The Edges restrict the direction of movement between vertices.



Representing Graphs

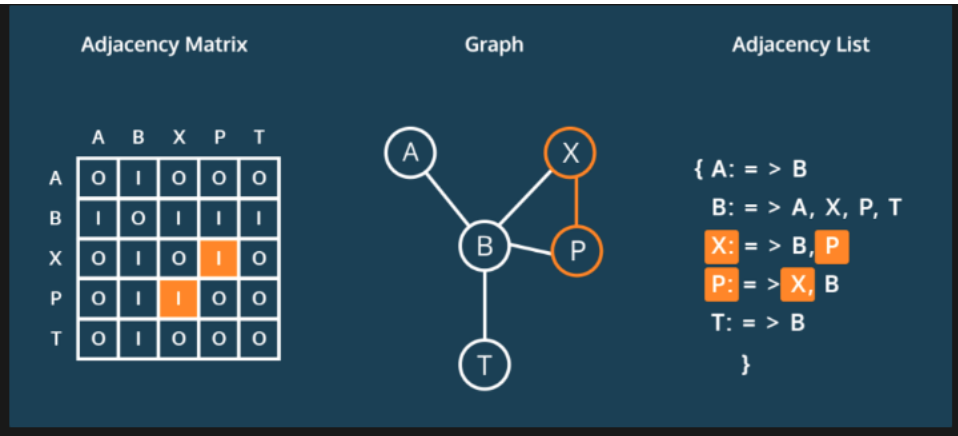
We typically represent the vertex-edge relationship of a graph in two ways: an adjacency list or an adjacency matrix.

An adjacency matrix is a table. Across the top, every vertex in the graph appears as a column. Down the side, every vertex appears again as a row. Edges can be bi-directional, so each vertex is listed twice.

To find an edge between `B` and `P`, we would look for the `B` row and then trace across to the `P` column. The contents of this cell represent a possible edge.

Our diagram uses `1` to mark an edge, `0` for the absence of an edge. In a weighted graph, the cell contains the cost of that edge.

In an adjacency list, each vertex contains a list of the vertices where an edge exists. To find an edge, one looks through the list for the desired vertex.



Graphs Implementation

`Vertex`:

- Uses a dictionary as an adjacency list to store connected vertices.
- Connected vertex names are keys and the edge weights are values.
- Has methods to add edges and return a list of connected vertices.

```
class Vertex:
  def __init__(self, value):
    self.value = value
    self.edges = {}

  def add_edge(self, vertex, weight = 0):
    self.edges[vertex] = weight

  def get_edges(self):
    return list(self.edges.keys())
```

`Graph`:

- Can be initialized as a directed graph, where edges are set in one direction.
- Stores every vertex inside a dictionary
  - Vertex data is the key and the vertex instance is the value.
- Has methods to add vertices, edges between vertices, and determine if a path exists between two vertices.

```python
class Graph:
  def __init__(self, directed = False):
    self.graph_dict = {}
    self.directed = directed

  def add_vertex(self, vertex):
    self.graph_dict[vertex.value] = vertex

  def add_edge(self, from_vertex, to_vertex, weight = 0):
    self.graph_dict[from_vertex.value].add_edge(to_vertex.value, weight)
    if not self.directed:
      self.graph_dict[to_vertex.value].add_edge(from_vertex.value, weight)

  # Finding the path using breadth first search O(n) time complexity
  def find_path(self, start_vertex, end_vertex):
    start = [start_vertex]
    seen = {}
    while len(start) > 0:
      current_vertex = start.pop(0)
      seen[current_vertex] = True
      print("Visiting " + current_vertex)
      if current_vertex == end_vertex:
        return True
      else:
        vertices_to_visit = set(self.graph_dict[current_vertex].edges.keys())
        start += [vertex for vertex in vertices_to_visit if vertex not in seen]
    return False
```

Breadth First Search

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the vertices of a graph or tree level by level. It starts at a given source vertex (or node) and systematically explores the vertices in a breadthward motion, visiting all the neighbors of a vertex before moving to their neighbors.

Key Concepts:

1. **Queue:** BFS uses a queue data structure to keep track of vertices that need to be visited. The source vertex is enqueued initially.
2. **Level by Level:** BFS explores the graph level by level, ensuring that all vertices at a certain distance from the source are visited before moving to vertices farther away.
3. **Shortest Path:** In an unweighted graph, BFS guarantees finding the shortest path between the source vertex and any reachable vertex.
4. **Visited Nodes:** BFS maintains a list of visited vertices to avoid revisiting the same vertex and to prevent infinite loops in the case of cycles.

Algorithm Steps:

1. Enqueue the source vertex into the queue and mark it as visited.
2. While the queue is not empty:
   a. Dequeue a vertex from the front of the queue.
   b. Visit the dequeued vertex and process it.
   c. Enqueue all unvisited neighbors of the dequeued vertex into the queue and mark them as visited.

Applications:

1. Finding shortest paths in an unweighted graph.
2. Shortest path problems like finding the minimum number of moves to solve puzzles (e.g., sliding puzzle).
3. Traversing and searching trees and graphs.
4. Connectivity and reachability analysis in networks.

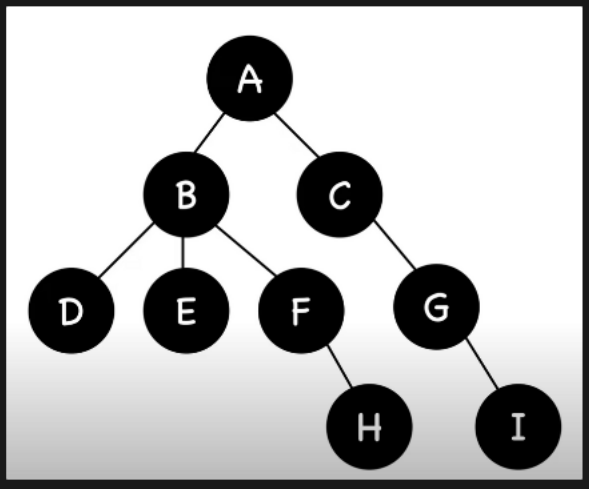
Advantages:

1. Guarantees the shortest path in an unweighted graph.
2. Visits all vertices reachable from the source vertex.
3. Useful for finding the shortest path or level-based exploration.

Disadvantages:

1. Can be memory-intensive, especially for graphs with many vertices and levels.
2. Might not be efficient for certain types of graphs or problems.

Consider the following to be the graph we are searching (it is a tree):



Here, we will traverse the tree in the following order:

bfs_queue: [A], Visited: (A)

bfs_queue: [], Visited: (A)

bfs_queue: [C, B], Visited: (A)

bfs_queue: [C], Visited: (A, B)

bfs_queue: [F, E, D, C], Visited: (A, B)

bfs_queue: [F, E, D], Visited: (A, B, C)

bfs_queue: [G, F, E, D], Visited: (A, B, C)

bfs_queue: [G, F, E], Visited: (A, B, C, D)

bfs_queue: [G, F], Visited: (A, B, C, D, E)

bfs_queue: [G], Visited: (A, B, C, D, E, F)

bfs_queue: [H, G], Visited: (A, B, C, D, E, F)

bfs_queue: [H], Visited: (A, B, C, D, E, F, G)

bfs_queue: [I, H], Visited: (A, B, C, D, E, F, G)

bfs_queue: [I], Visited: (A, B, C, D, E, F, G, H)

bfs_queue: [], Visited: (A, B, C, D, E, F, G, H, I)

```
def bfs(graph, start_vertex, target_value):
  path = [start_vertex]
  vertex_and_path = [start_vertex, path]
  bfs_queue = [vertex_and_path]
  visited = set()
  while bfs_queue:
    current_vertex, path = bfs_queue.pop(0)
    visited.add(current_vertex)
    for neighbor in graph[current_vertex]:
      if neighbor not in visited:
        if neighbor is target_value:
          return path + [neighbor]
        else:
          bfs_queue.append([neighbor, path + [neighbor]])
```

Depth First Search

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It starts at a given source vertex (or node) and explores as deeply as possible along each branch before moving to another branch.

Key Concepts:

1. **Stack or Recursion:** DFS can be implemented using either a stack data structure or recursion. The stack stores nodes to be visited, while recursion uses the call stack to keep track of the traversal.
2. **Exploration Depth:** DFS goes deep into a branch before exploring neighboring branches, which can lead to deeper exploration compared to Breadth-First Search (BFS).
3. **Visited Nodes:** DFS maintains a list of visited vertices to avoid revisiting the same vertex and to prevent infinite loops in the case of cycles.

Algorithm Steps:

1. Start at the source vertex and mark it as visited.
2. Explore an unvisited neighbor of the current vertex, mark it as visited, and move to it.
3. Repeat step 2 for the newly visited vertex until you reach a dead end.
4. Backtrack to the most recent vertex with unexplored neighbors and continue exploration.
5. Repeat steps 2-4 until all vertices are visited.

Applications:

1. Topological sorting of directed acyclic graphs (DAGs).
2. Solving puzzles and games like mazes, Sudoku, and the 8-puzzle.
3. Finding connected components and strongly connected components in a graph.
4. Pathfinding in certain cases.

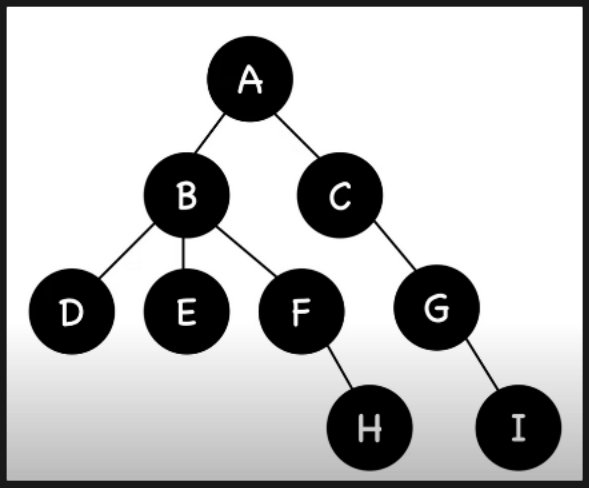
Advantages:

1. Memory-efficient as it requires minimal memory to store the path to the current vertex.
2. Can explore deeper levels quickly and might find solutions faster if the goal is deeper in the graph.

Disadvantages:

1. Might not guarantee the shortest path in an unweighted graph.
2. Can get stuck in infinite loops if cycles are present and not handled properly.

Consider the following to be the graph we are searching (it is a tree):



Here, we will traverse the tree in the following order:

dfs_stack: [A], Visited: (A)

dfs_stack: [], Visited: (A)

dfs_stack: [C, B], Visited: (A)

dfs_stack: [B], Visited: (A, C)

dfs_stack: [G, B], Visited: (A, C)

dfs_stack: [B], Visited: (A, C, G)

dfs_stack: [I, B], Visited: (A, C, G)

dfs_stack: [B], Visited: (A, C, G, I)

dfs_stack: [], Visited: (A, C, G, I, B)

dfs_stack: [F, E, D], Visited: (A, C, G, I, B)

dfs_stack: [E, D], Visited: (A, C, G, I, B, F)

dfs_stack: [H, E, D], Visited: (A, C, G, I, B, F)

dfs_stack: [E, D], Visited: (A, C, G, I, B, F, H)

dfs_stack: [D], Visited: (A, C, G, I, B, F, H, E)

dfs_stack: [], Visited: (A, C, G, I, B, F, H, E, D)

```python
def dfs(graph, current_vertex, target_value, visited = None):
  if visited is None:
    visited = []
  visited.append(current_vertex)
  if current_vertex is target_value:
    return visited

  for neighbor in graph[current_vertex]:
    if neighbor not in visited:
      path = dfs(graph, neighbor, target_value, visited)
      if path:
        return path

some_important_graph = {
    'lava': set(['sharks', 'piranhas']),
    'sharks': set(['lava', 'bees', 'lasers']),
    'piranhas': set(['lava', 'crocodiles']),
    'bees': set(['sharks']),
    'lasers': set(['sharks', 'crocodiles']),
    'crocodiles': set(['piranhas', 'lasers'])
  }

print(dfs(some_important_graph, 'lava', 'lasers'))
```

Both algorithms have a time complexity of O(Vertices + Edges)

Greedy Algorithms

A greedy algorithm is an algorithm used in *optimization problems*, which aim to find the *optimal* solution (either the maximum or the minimum) among all feasible solutions. For example, finding the shortest path between two locations on the map is an optimization problem that aims to minimize the result.

A greedy algorithm does **NOT** always achieve the global optimal solution. This is because greedy algorithms make decisions purely based on the best choice at the time, without regard to the overall problem.

A problem that can be correctly solved by a greedy algorithm must satisfy these two properties:

- Optimal substructure property - the optimal solution for the problem contains optimal solutions to the sub-problems.
- Greedy property - the global optimal solution can be reached by making locally optimal choices.

Greedy algorithms have the following advantages:

- Easy to understand and implement.
- Time and space complexities are easy to analyze.
- Perform better than other paradigms like divide-and-conquer.

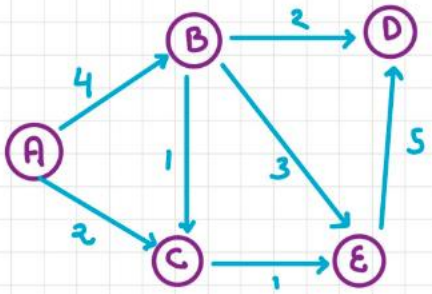
However, there are some crucial disadvantages:

- Most greedy algorithms fail to find the global optimal solution.
- Hard to prove the correctness of a greedy algorithm.

Dijkstra's Algorithm

Dijkstra's Algorithm works as following:

1. Instantiate a dictionary that will eventually map vertices to their distance from the start vertex
2. Assign the start vertex a distance of 0 in a min heap
3. Assign every other vertex a distance of infinity in a min heap
4. Remove the vertex with the smallest distance from the min heap and set that to the current vertex
5. For the current vertex, consider all of its adjacent vertices and calculate the distance to them as **(distance to the current vertex) + (edge weight of current vertex to adjacent vertex)**.
6. If this new distance is less than the current distance, replace the current distance.
7. Repeat 4 and 5 until the heap is empty
8. After the heap is empty, return the distances



Vertex	Dist.	Prev.	Visited
A	0		True
B	∞		False
C	∞		False
D	∞		False
E	∞		False

Vertex	Dist.	Prev.	Visited
A	0		True
B	4		False
C	2	A	True
D	∞		False
E	∞		False

Vertex	Dist.	Prev.	Visited
A	0		True
B	4		False
C	2	A	True
D	∞		False
E	3	C	True

Vertex	Dist.	Prev.	Visited
A	0		True
B	4		False
C	2	A	True
D	8	E	True
E	3	C	True

Vertex	Dist.	Prev.	Visited
A	0		True
B	4	A	True
C	2	A	True
D	6	B	True
E	3	C	True

1) A to B
Shortest Dist. → 4
Shortest Path. → A → B

2) A to C
Shortest Dist. → 2
Shortest Path → A → C

3) A to D
Shortest Dist. → 6
Shortest Path → A → B → D

4) A to E
Shortest Dist. → 3
Shortest Path → A → C → E

```python
from heapq import heappop, heappush
from math import inf

graph = {
        'A': [('B', 10), ('C', 3)],
        'C': [('D', 2)],
        'D': [('E', 10)],
        'E': [('A', 7)],
        'B': [('C', 3), ('D', 2)]
    }


def dijkstras(graph, start):
  distances = {}

  for vertex in graph:
    distances[vertex] = inf

  distances[start] = 0
  vertices_to_explore = [(0, start)]

  while vertices_to_explore:
    current_distance, current_vertex = heappop(vertices_to_explore)

    for neighbor, edge_weight in graph[current_vertex]:
      new_distance = current_distance + edge_weight

      if new_distance < distances[neighbor]:
        distances[neighbor] = new_distance
        heappush(vertices_to_explore, (new_distance, neighbor))

  return distances

distances_from_d = dijkstras(graph, 'D')
print("\n\nShortest Distances: {0}".format(distances_from_d))
```

Just like breadth-first search and depth-first search, to search through an entire graph, in the worst case, we would go through all of the edges and all of the vertices resulting in a runtime of O(E + V).

For Dijkstra's, we use a min-heap to keep track of all the distances. Searching through and updating a min-heap with V nodes takes O(log V) because in each layer of the min-heap, we reduce the number of nodes we are looking at by a factor of 2.

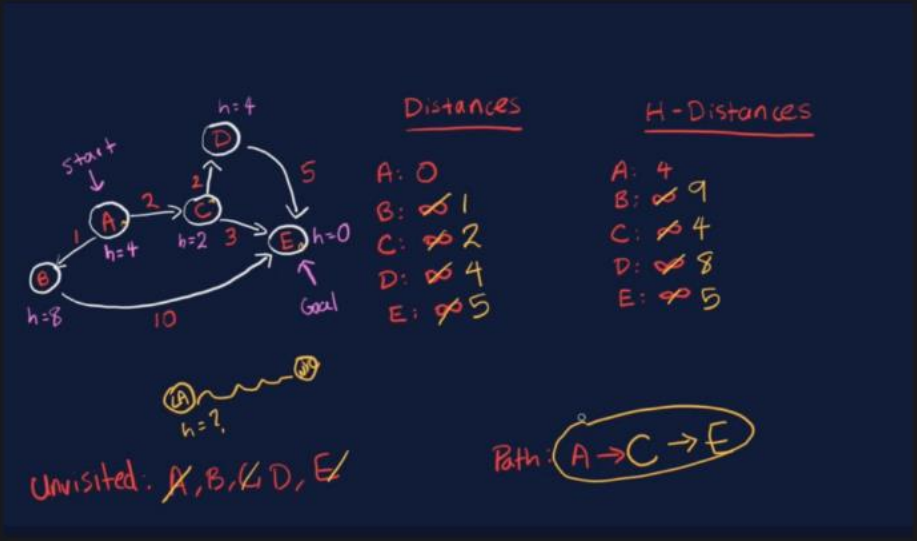
> **In the worst case, we would update the min-heap every iteration. Since there are at most E + V iterations of Dijkstra's and it takes log V to update a min-heap in the worst case, then the runtime of Dijkstra's is O((E+V)log V).**

Pathfinding Algorithms (A* Algorithm)

Pathfinding is the algorithmic concept of finding the shortest path between point A and point B on a graph. We can find pathfinding algorithms implemented in many location-based applications.

The A* algorithm is a modification of Dijkstra's algorithm. *A** works by having both a starting vertex and a target vertex. The algorithm computes an estimated distance, or heuristic, for all possible paths between the starting vertex and the goal vertex and then selects the shortest one.

Dijkstra's algorithm is great for finding the shortest distance from a start vertex to all other vertices in the graph. However, it is not the best when we are just looking for the shortest distance from a single start vertex to a single end vertex.



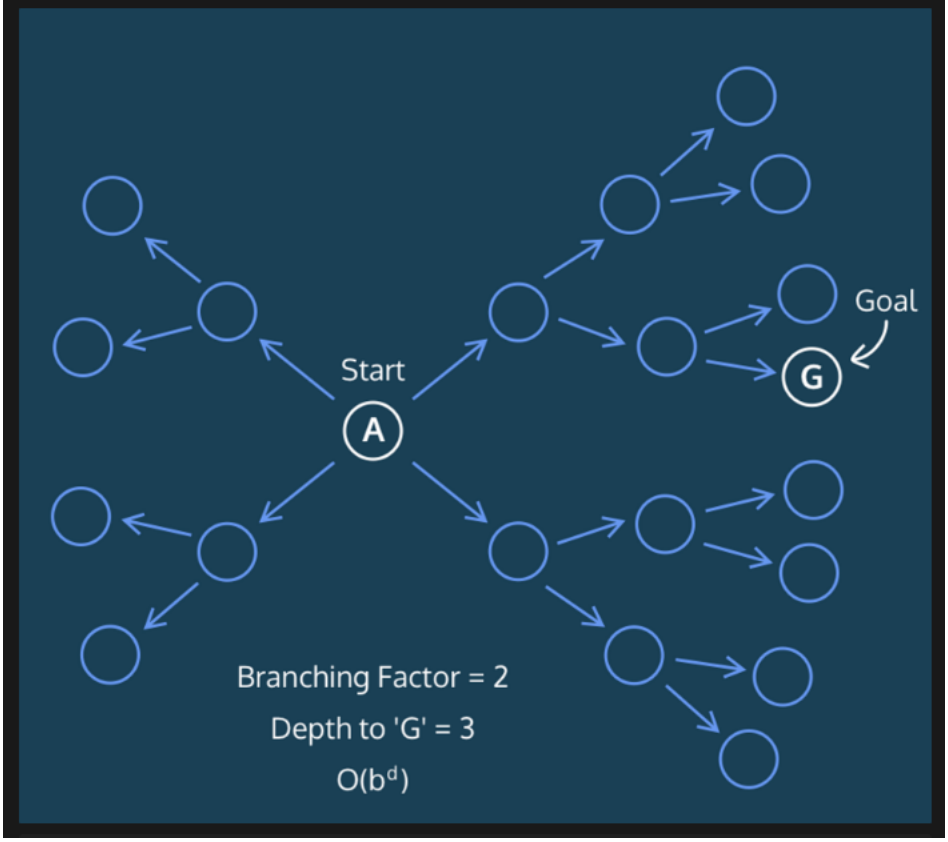
Because of heuristic values, we don't pop off places that might not be worth visiting. This is the reason A* is considered to be better than dijkstras.

Runtime of A*

In A*, we have a goal vertex. Thus, the algorithm is optimized such that in most graphs, every vertex will **NOT** be searched. Remember that Dijkstra's has a runtime of `O((V+E) log V)` because in the worst case you search every vertex and go through all the edges while storing data in a min-heap.

For A*, let's try describing the runtime in better terms. Using the above graph, let's call `b` the branching factor of the graph. The branching factor is the *average* number of edges per vertex in the graph. In this case, **on average**, every vertex has 2 edges, and thus, the branching factor is equal to 2. Let's call `d` the depth of the goal vertex from the start vertex. Because the goal vertex is 3 edges away from the start vertex, `d` is equal to 3.

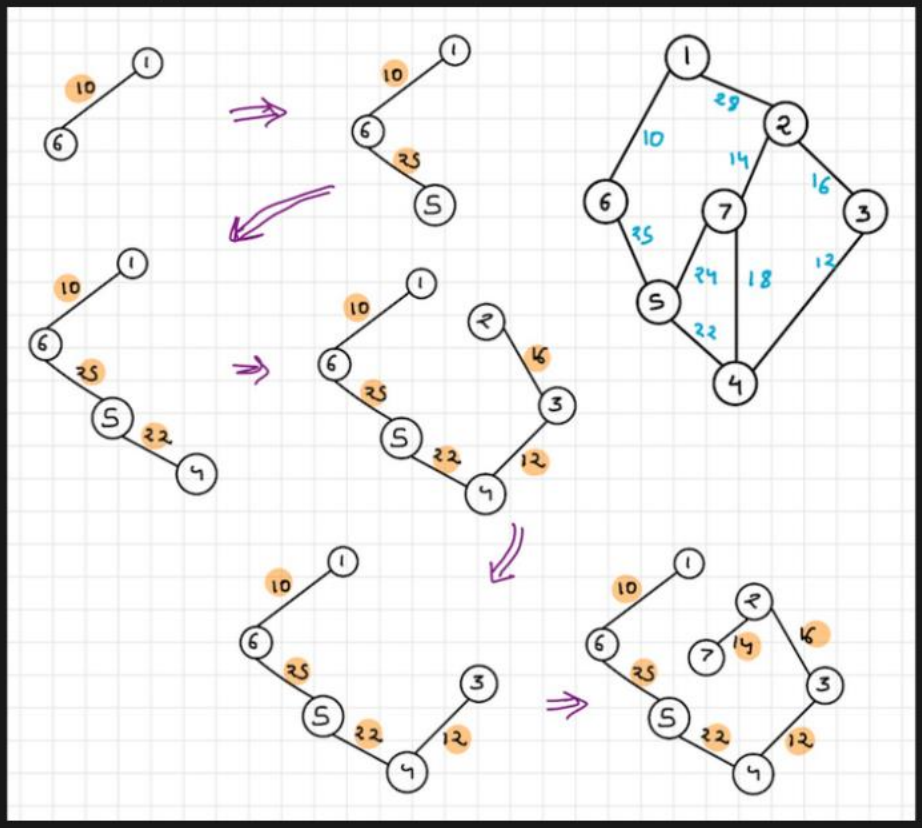
In the worst case, we would look at all of the edges in the direction of the goal vertex until we reach the goal vertex. We would look at `b` edges for every vertex in our search for close to `d` iterations. Thus, the runtime is **O(b^d)**.



```
def a_star(graph, start, target):
  print("Starting A* algorithm!")
  count = 0
  paths_and_distances = {}
  for vertex in graph:
    paths_and_distances[vertex] = [inf, [start.name]]

  paths_and_distances[start][0] = 0
  vertices_to_explore = [(0, start)]
  while vertices_to_explore and paths_and_distances[target][0] == inf:
    current_distance, current_vertex = heappop(vertices_to_explore)
    for neighbor, edge_weight in graph[current_vertex]:
      new_distance = current_distance + edge_weight + heuristic(neighbor, tar
get)
      new_path = paths_and_distances[current_vertex][1] + [neighbor.name]

      if new_distance < paths_and_distances[neighbor][0]:
        paths_and_distances[neighbor][0] = new_distance
        paths_and_distances[neighbor][1] = new_path
        heappush(vertices_to_explore, (new_distance, neighbor))
        count += 1
        print("\nAt " + vertices_to_explore[0][1].name)

  print("Found a path from {0} to {1} in {2} steps: ".format(start.name, targ
et.name, count), paths_and_distances[target][1])

  return paths_and_distances[target][1]
```

Prim's Algorithm

We pick a vertex to be the "root" of the MST. After that we simply grow the tree by joining isolated vertices one at a time. An isolated vertex is any vertex that isn't part of the MST yet picking the smallest edge weight. To support this, we will use a MinHeap. We queue into this heap edges that will connect an isolated vertex with the current MST. We use infinity if there is no direct edge yet to any vertex in the MST.



Kruskal's Algorithm

Kruskal's algorithm starts by sorting edges according to weights. It then picks the smallest edge out and adds it to T only if the end points are in different connected components. If we use something like a BFS or DFS to find connected components, it will be very slow. Instead, what we can do is use a disjoint set.