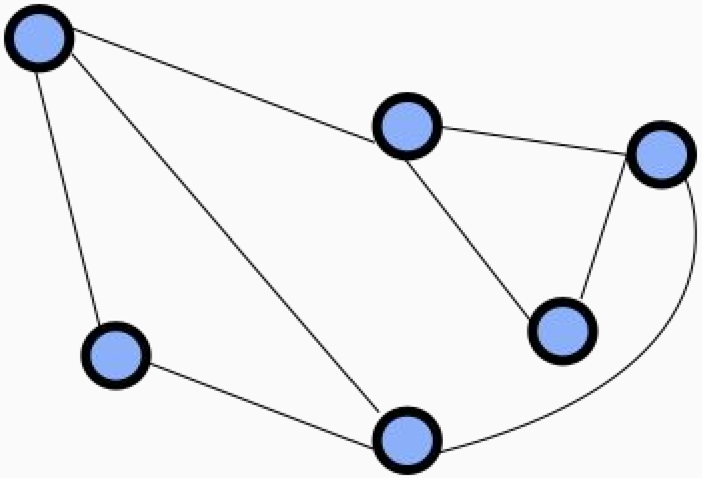
“Graphs and Path-finding”

Background

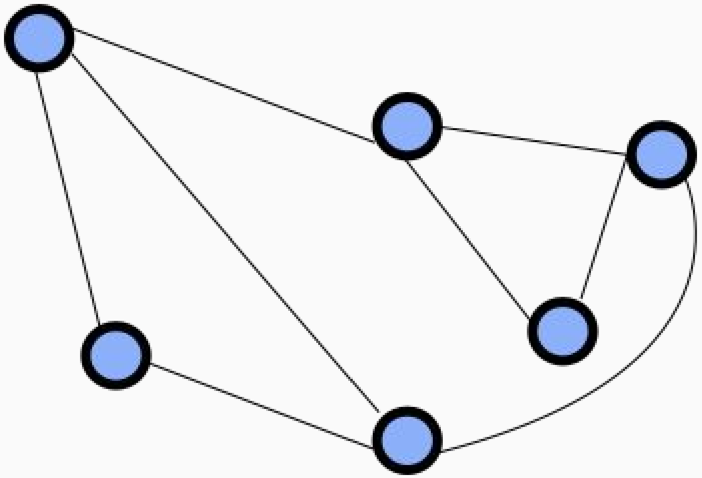
- Graphs
- Algorithms
- CS 3114?
- Math 3134?

Graphs



Graphs

- Nodes (vertexes)
- Edges

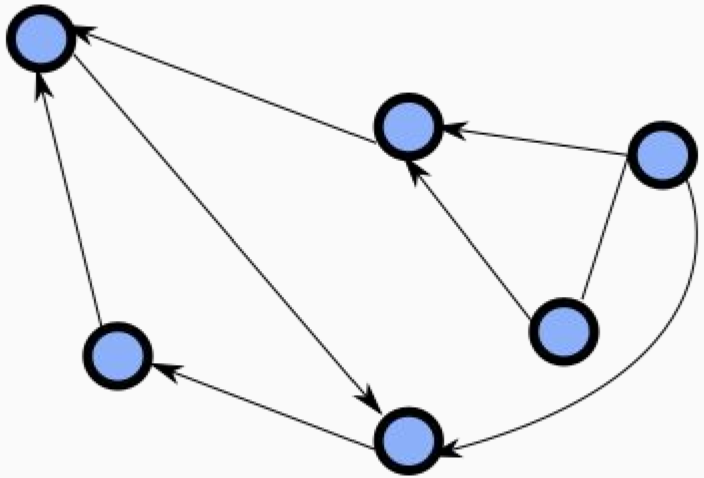


Graphs

- Types of graphs

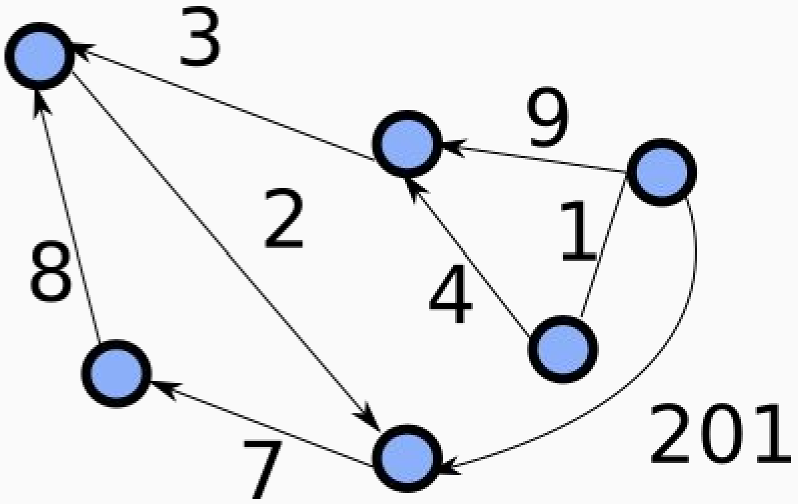
Graphs

- Directed



Graphs

- Weighted



Algorithms

- What can we do we this?

Algorithms

- Traversals/Searches/”Tree-growing”
    - BFS, DFS
    - Dijkstra’s

Dijkstra’s and BFS

Breadth-first search

- Starting at a single node, search all neighbors
- Then, search nodes a distance 2 away
- Repeat until goal is found
- Doing this, you can find the shortest path to the goal

Dijkstra’s Algorithm

- Single source, all destinations shortest path algorithm
- Given a node, find the shortest path to any or all other nodes
- Frequently used for a single target/goal
- Very similar in structure to BFS, but for weighted graphs

```java
static State bfs(State start) {
	Queue<State> queue = new ArrayDeque<State>();
	Set<State> visited = new HashSet<State>();

	queue.offer(start);
	visited.add(start);
	while (!queue.isEmpty()) {
		State current = queue.poll();
		if (current.goal()) {
			return current;
		}

		for (State adj : current.adj()) {
			if (!visited.contains(adj)) {
				queue.offer(adj);
				visited.add(adj);
			}
		}
	}
}
```

Basic code for BFS in Java

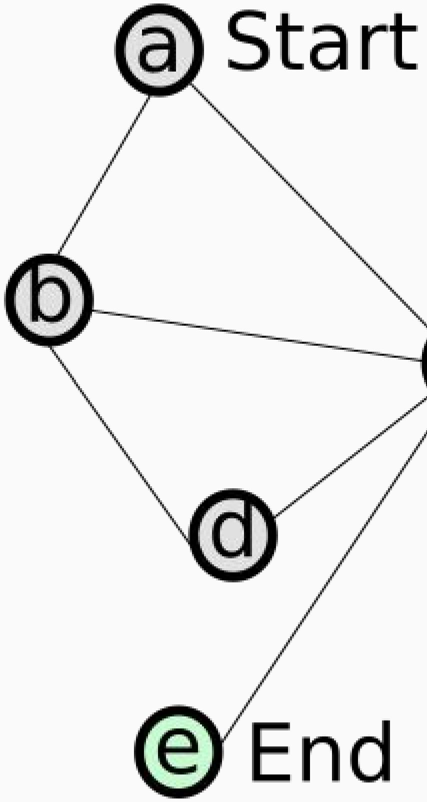
```java
static State dijkstras(State start) {
    Queue<State> queue = new PriorityQueue<State>();
    Map<State, Double> distances = new HashMap<State, Double>();

    queue.offer(start);
    distances.put(start, start.dist);
    while (!queue.isEmpty()) {
        State current = queue.poll();
        if (current.goal()) {
            return current;
        }
        if (distances.get(current) < current.dist) {
            continue;
        }

        for (State adj : current.adj()) {
            Double best = distances.get(adj);
            if (best == null || best > adj.dist) {
                queue.offer(adj);
                distances.put(adj, adj.dist);
            }
        }
    }
    return null;
}
```

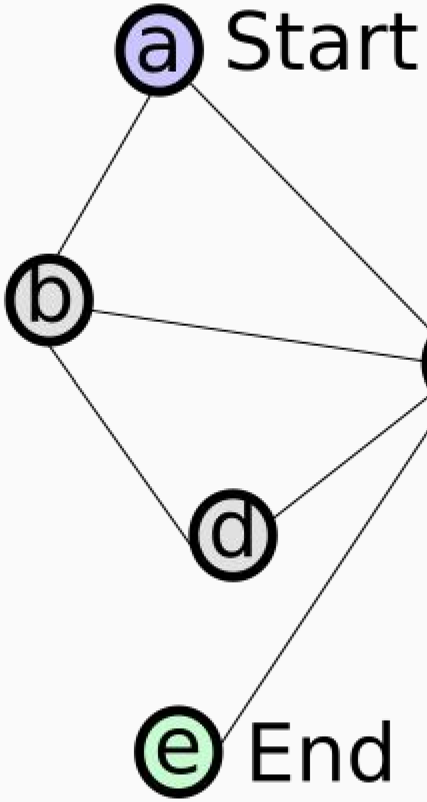
Basic code for Dijkstra’s in Java

Queue



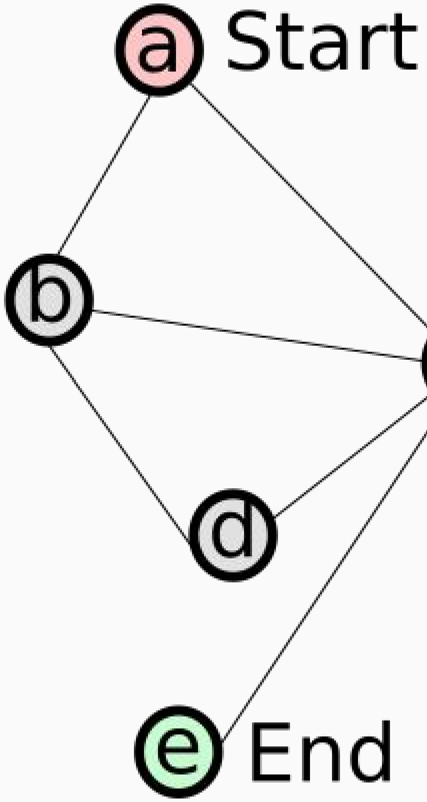
Queue

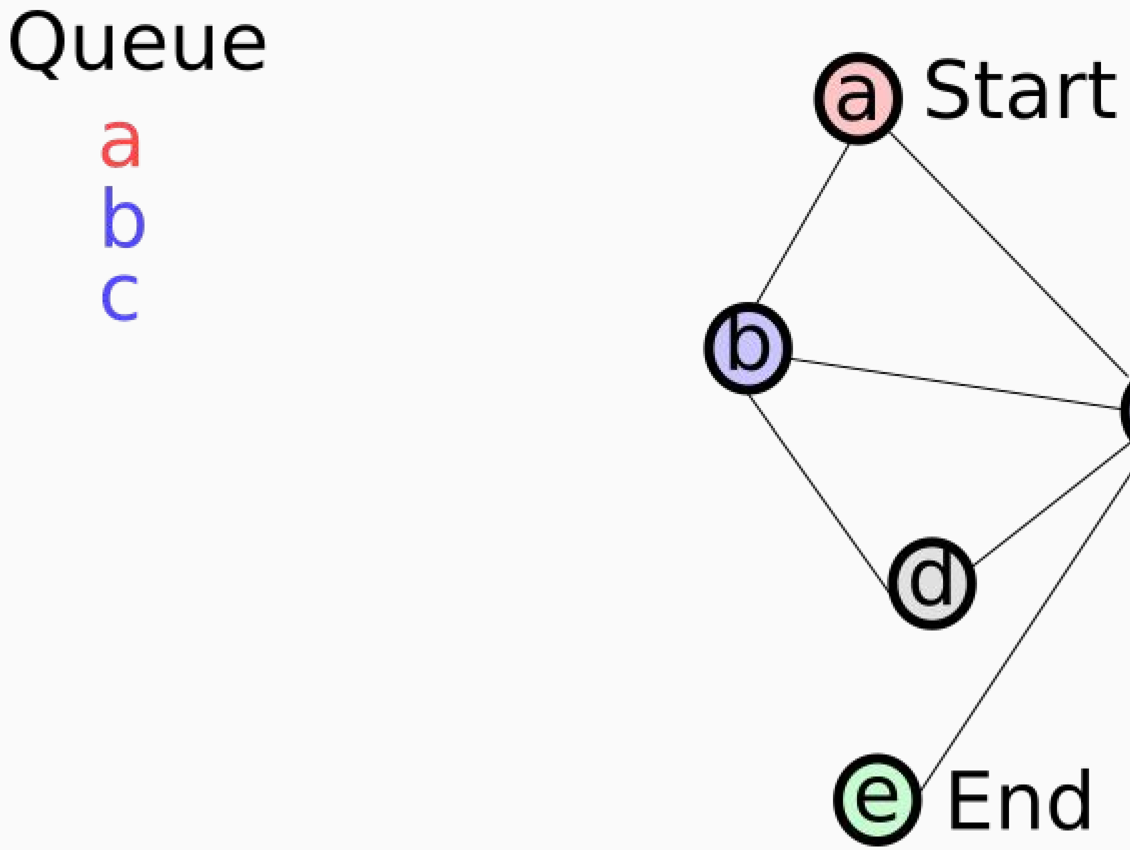
a



Queue

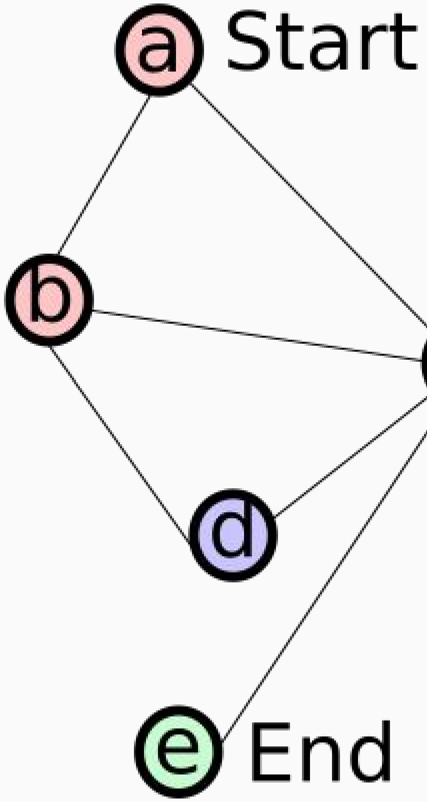
a




Queue
a
b
c
a Start
b
c
d
e End

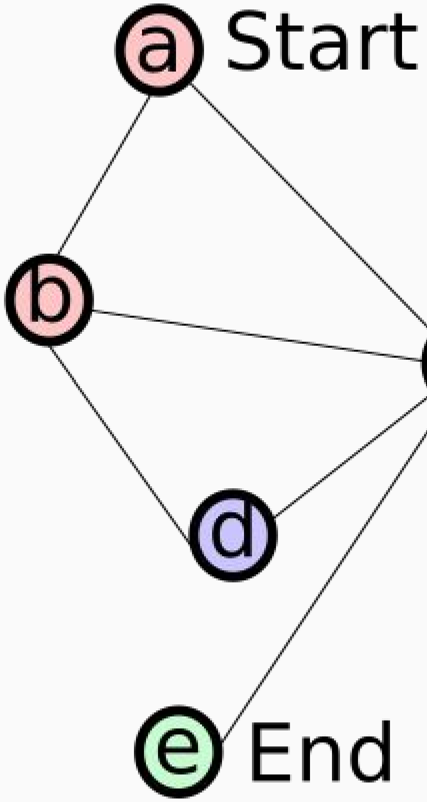
Queue

a
b
c
d



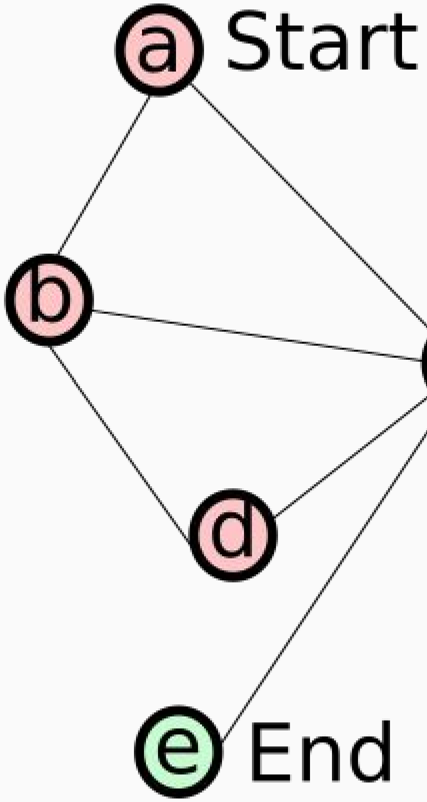
Queue

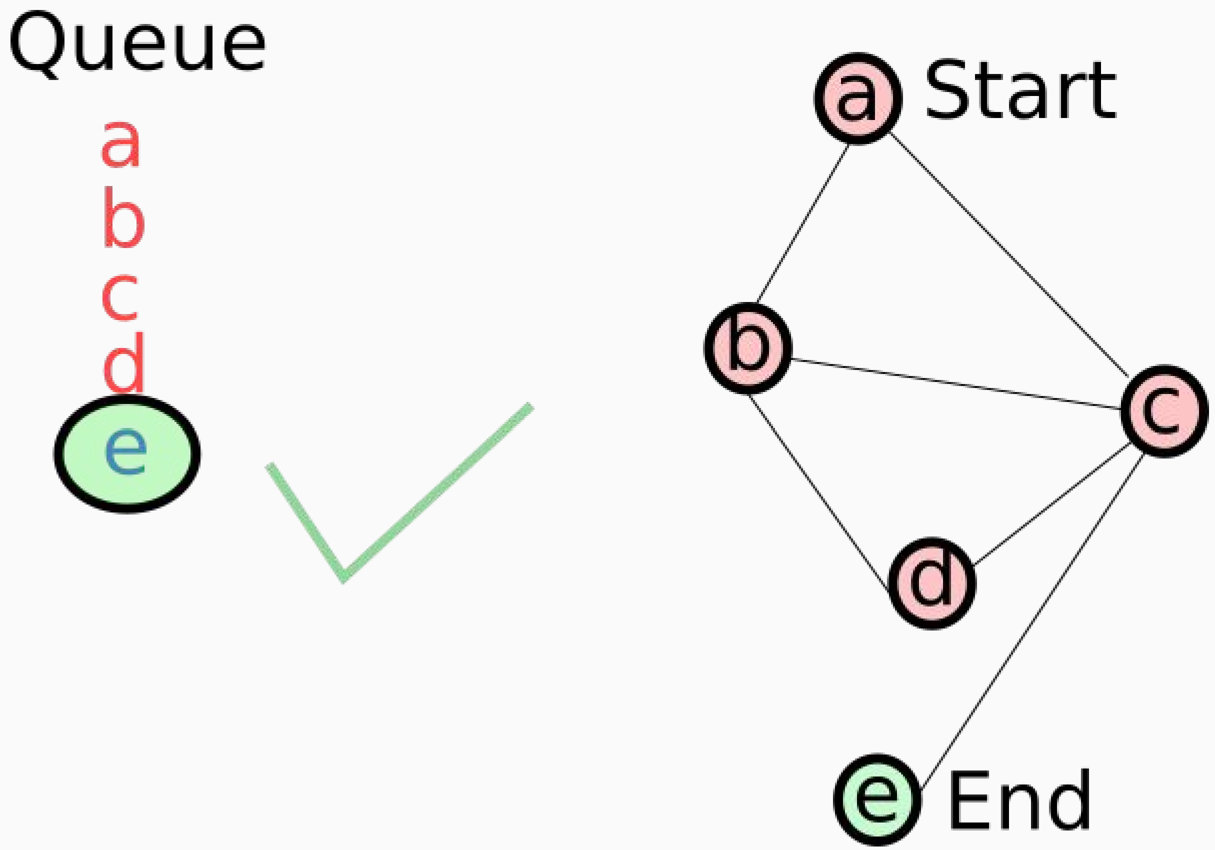
a
b
c
d
e



Queue

a
b
c
d
e



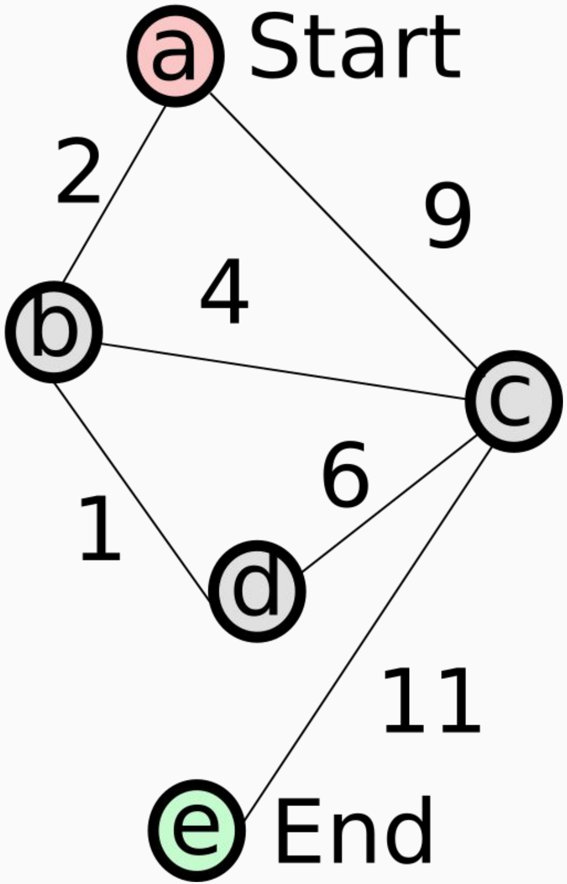

Queue
a
b
c
d
e
a Start
b
c
d
e End

Dijkstra’s vs. BFS

- Works on weighted graphs
- Still processes nodes in increasing distance from start
- Keeps track of current distances to each target
- Greedily picks the target with the closest distance

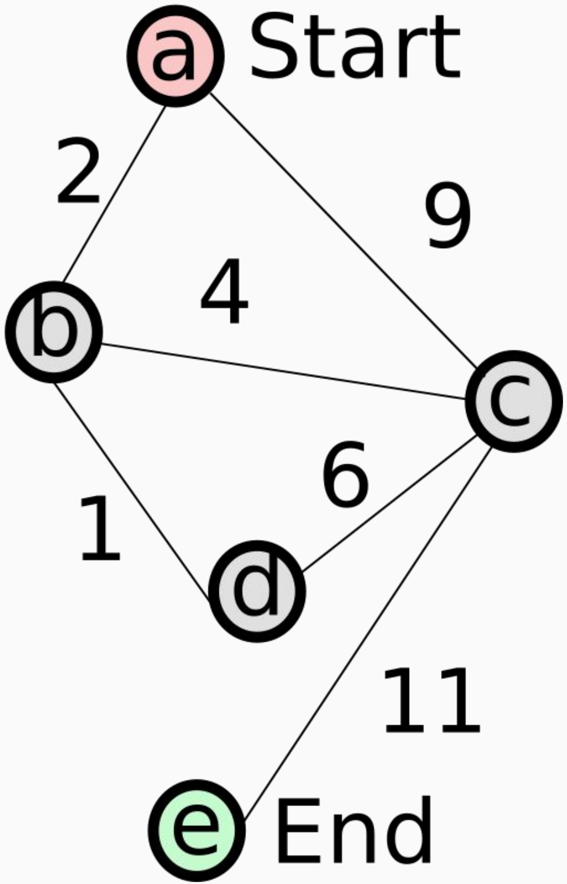
Distances

a: 0
b: ∞
c: ∞
d: ∞
e: ∞



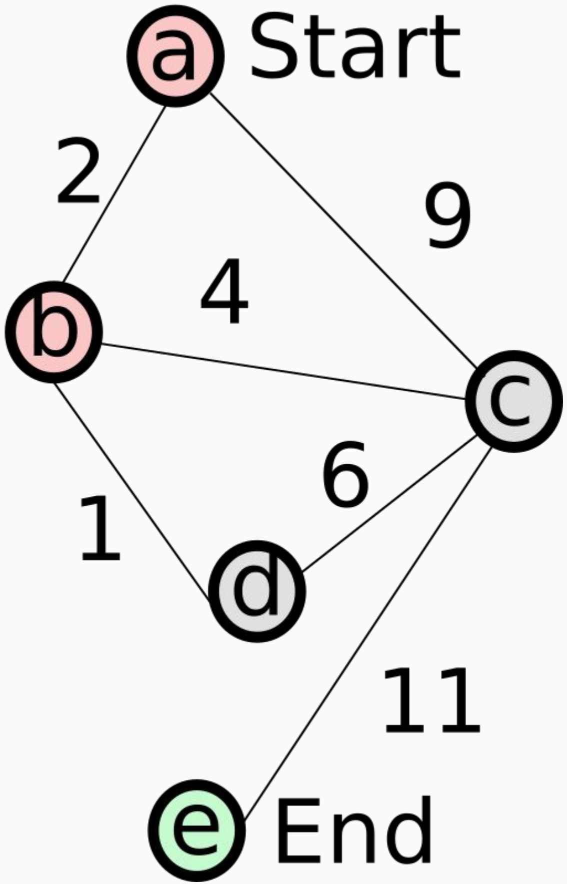
Distances

a: 0
b: 2
c: 9
d: ∞
e: ∞



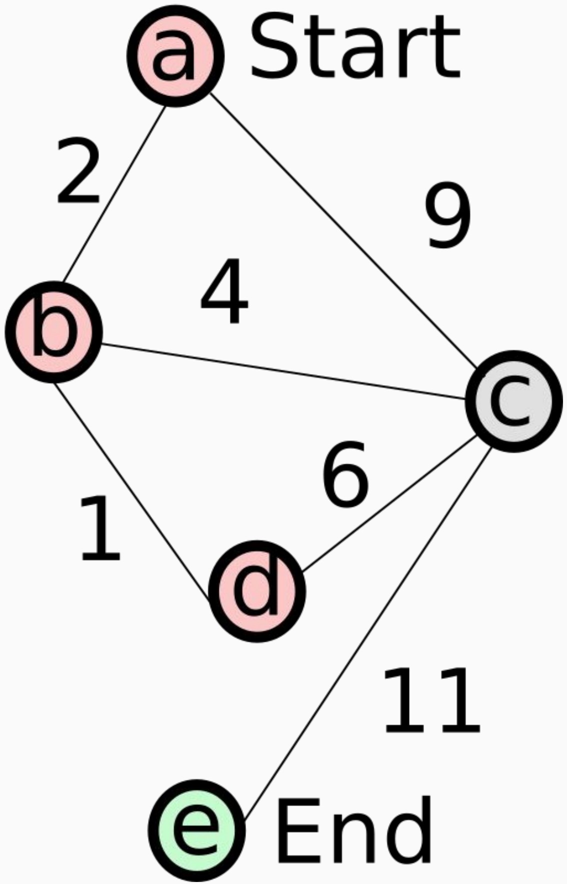
Distances

a: 0
b: 2
c: ~~9~~ 6
d: 3
e: ∞



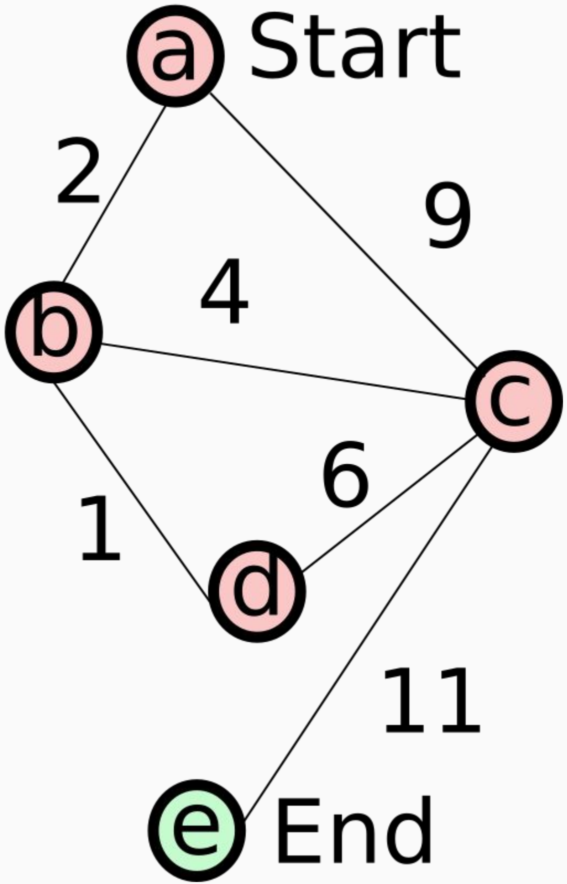
Distances

a: 0
b: 2
c: ~~9~~ 6
d: 3
e: ∞



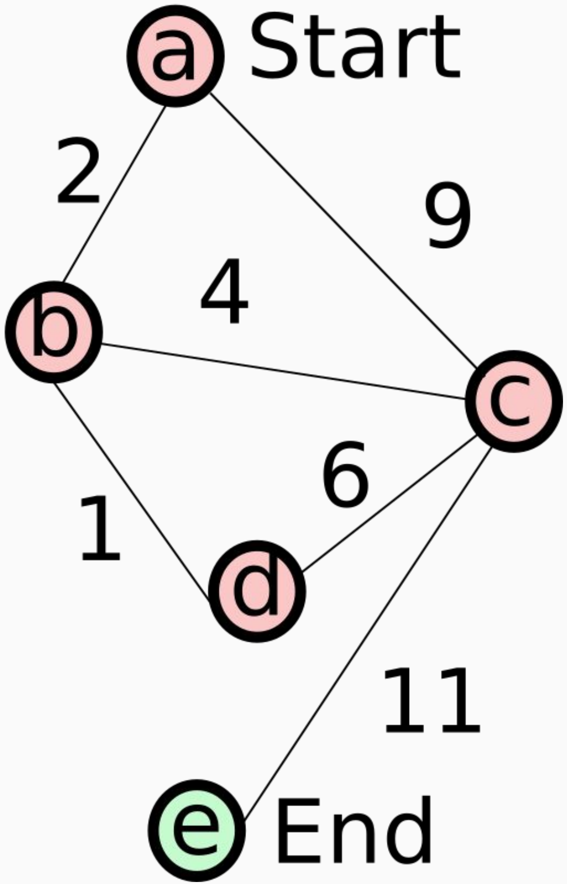
Distances

a: 0
b: 2
c: ~~9~~ 6
d: 3
e: 17



Distances

a: 0
b: 2
c: ~~9~~ 6
d: 3
e: 17



Visual of Dijkstra's

Implementation

Storing a graph

- List of edges
    - Edge[], ArrayList<Edge>
- Adjacency list
    - List<Node>[], List<List<Node>>, etc
- Adjacency matrix

Storing a graph

- An adjacency list is preferably in almost every case, unless your algorithm depends on a different data structure
  - eg: Floyd-Warshall, Kruskal's
- Easiest way is like so:
  - ```
    ArrayList<ArrayList<Node>> graph = new ArrayList<ArrayList<Node>>();

    for (int i = 0; i < N; i++) {

        graph.add(new ArrayList<Node>());

    }
    ```
    ```

Adjacency lists

- To connect two nodes, just add the node to the adjacency list:
  - `graph.get(i).add(j)`
- For bidirectional connections, you'll also need to add the reverse:
  - `graph.get(j).add(i)`

Adjacency lists

- The main reason adjacency lists are preferred is they easily let you find nodes that are connected to a specific node ("adjacent" nodes, or neighbors)
- The neighbors are stored directly! All you need to do is iterate over them:
  - ```
    for (Node adj : graph.get(current)) {
        // whatever you want
    }
    ```
    ```

Adjacency lists

- There are many equivalent ways to represent adjacency lists
  - ie: arrays, HashMaps of Lists, using Sets instead of List, etc.
- The general idea is that you store the neighbors of each node directly
- How you represent nodes is up to you, in the simplest case they can be just integers
- Other times you may create a Node/State/Vertex class

Basic BFS structure

- Queue of nodes to process
- Set of visited nodes
- Each turn, process one node
  - Add any neighbors that are not visited to the queue
  - Calculate any information based on neighbors

```java
Queue<Node> queue = new ArrayDeque<Node>();
Set<Node> visited = new HashSet<Node>();
queue.offer(start);
visited.add(start);
while (!queue.isEmpty()) {
    Node current = queue.poll();
    // Process the current state, check for goal, etc
    if (current.equals(goal)) {
        // Done, found the goal
    }
    for (Node adj : graph.get(current)) {
        if (!visited.contains(adj)) {
            // Handle new neighbor states (distances, etc)
            visited.add(adj);
            queue.offer(adj);
        }
    }
}
```

Usage in competitive programming

- What’s the runtime of a typical BFS?
- Processes each node once for sure
- How about edges?

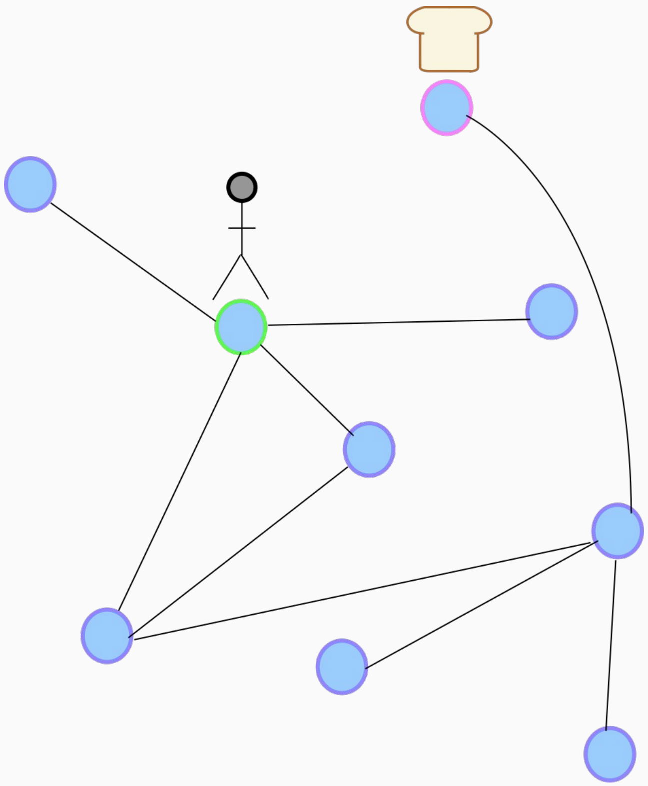
Usage in competitive programming

- $O(V + E)$
- V is number of nodes, E is number of edges
- Sometimes E can be $O(V^2)$, read problem carefully

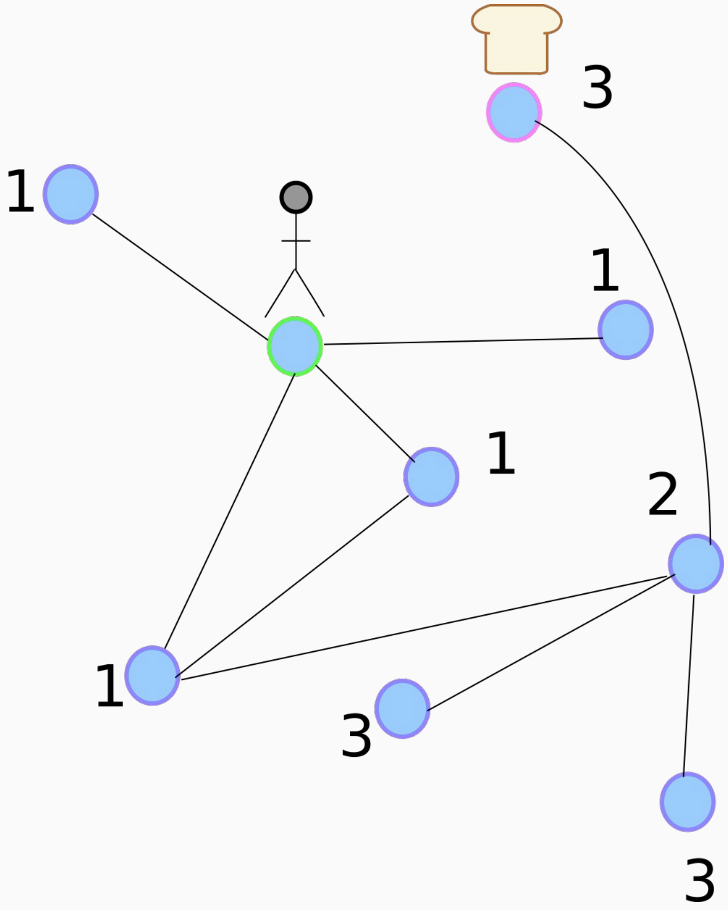
Usage in competitive programming

- How do you recognize a BFS problem?
- Looking for shortest paths in an unweighted graph
  - Could be an integer grid, like 2114 maze solver
  - Could be an explicit graph in the problem
  - Could be a “state space” exploration, where the graph is implied by transitions between states
- Could be a general graph traversal (although DFS also works)
- Could be processing levels in increasing distance (in a tree or a graph)

Sample problem



Sample problem illustration (https://contest.spruett.me/problems)



Sample problem illustration

```java
Scanner rdr = new Scanner(System.in);
int N = rdr.nextInt();
int E = rdr.nextInt();
int S = rdr.nextInt();
int B = rdr.nextInt();

// Initialize graph

for (int i = 0; i < E; i++) {
    int u = rdr.nextInt();
    int v = rdr.nextInt();
    // Connect u <--> v edge
}

int min = .... // from your algorithm!
int max = ....

System.out.println(min + " " + max);
```

Sample I/O code (http://spruett.me/blog/static/sampleio.html)

- https://spruett.me/blog/static/Bread.html
- https://spruett.me/blog/static/BreadNode.html