Graph Representations, BFS, and Dijkstra's Algorithm

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements

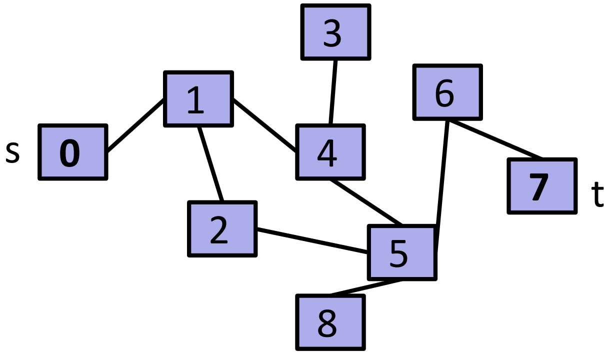
- Midterm is *this Friday*
  - If your student number ends in an odd number, go to KNE 2**1**0
  - If your student ends in an even number, go to KNE 2**2**0
  - Workshops will be focused on your midterm questions
  - Review session Thursday night: 4:30-6:30 @ ARC 147

Lecture Outline

- **Graph Representations**
- Graph Traversals: BFS
- Shortest Paths: Dijkstra's Algorithm

Review: Depth-First Search

```
connected(Node s, Node t) {
  if (s == t) {
    return true;
  } else {
    s.visited = true;
    for (Node n : s.neighbors) {
      if (n.visited) {
        continue;
      }
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```



What data structure should we store the graph in?

Graph ADT

Graph ADT. A collection of vertices and the edges connecting them.

- We can query for vertices connected to, or edges leaving from, a vertex v
- Edges are specified as pairs of vertices
  - We can add/remove edges from the graph

- Key operations include:
  - getAllVertices()
  - getAllEdges()
  - addEdge(v, w)
  - getAllEdgesFrom(v)
  - hasEdge(v, w)

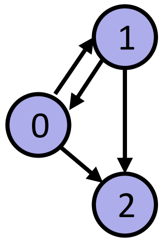
Graph Data Structures

- Just as we saw multiple representations for a tree, there are multiple data structures that implement the Graph ADT
  - Node class with left/right pointers vs Binary Heap's array representation

- Option 1: Adjacency Matrix
- Option 2: Edge Sets
- Option 2: Adjacency List

Graph Data Structure #1: Adjacency Matrix

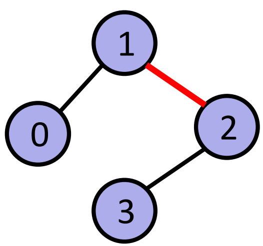
- Directed Graphs:



v \ w	0	1	2
0	F	T	T
1	T	F	T
2	F	F	F

diagonal is always false in a simple graph (both directed & undirected)

- Undirected Graphs:
  - Each edge is represented twice in the matrix: simplicity vs space



v \ w	0	1	2	3
0	F	T	F	F
1	T	F	**T**	F
2	F	**T**	F	T
3	F	F	T	F

matrix is symmetric around the diagonal for undirected graphs only

Poll Everywhere

pollev.com/uwcse373

- What is the runtime for getAllEdges() if we use an **adjacency matrix** representation, where V is the number of vertices and E is the number of edges?

A. Θ(V)
B. Θ(V + E)
C. Θ(V^2)
D. Θ(V*E)
E. I'm not sure …

V

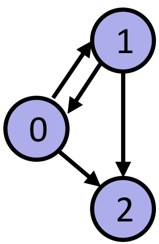
V

	0	1	2	3
0	F	T	F	F
1	T	F	**T**	F
2	F	**T**	F	T
3	F	F	T	F

Graph Data Structure #2: Edge Set

- A simple collection of edges
  - Eg: HashSet<Edge>. Recall that each edge is a (possibly directed) pair of vertices

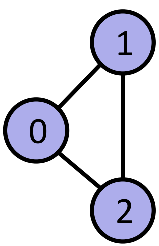
- Directed Graph:



{0, 1}, {1, 0}, {1, 2}, {0, 2}

(vertex ordering matters for directed graphs)

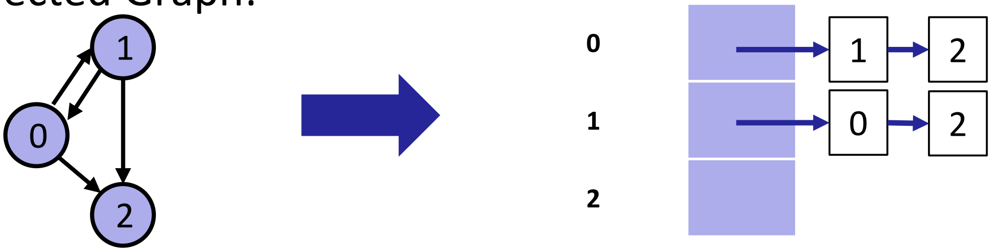
- Undirected Graph:



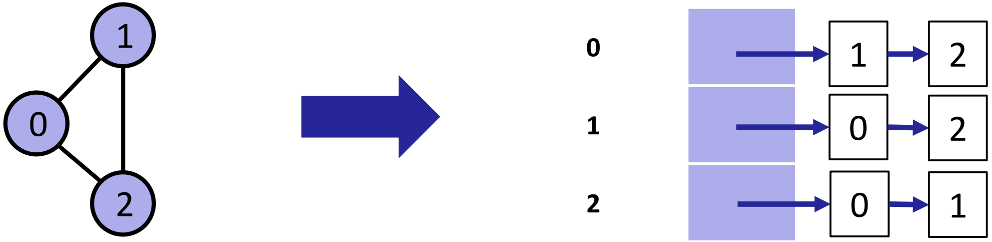
{0, 1}, {1, 2}, {0, 2}

Graph Data Structure #3: Adjacency List

- Array of lists of vertices, indexed by vertex
  - Most popular approach
- Directed Graph:



- Undirected Graph:
  - As with adjacency matrix, each edge is represented twice

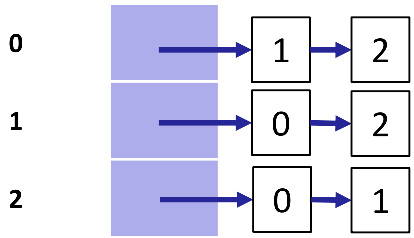


Poll Everywhere

pollev.com/uwcse373

- What is the runtime for getAllEdges() if we use an **adjacency list** representation, where V is the number of vertices and E is the number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$
E. I'm not sure …



Graph Representations

- In practice, adjacency lists are most common
  - Many graph algorithms rely heavily on getAllEdgesFrom(v)
  - Most graphs are sparse (ie, not many edges)

	getAllEdgesFrom(v)	hasEdge(v, w)	getAllEdges()
Adjacency Matrix	Θ(V)	Θ(1)	Θ(V^2)
Edge Set	Θ(E)	Θ(E)	Θ(E)
Adjacency List	O(V)	Θ(degree(v))	Θ(E + V)

best and worst case don't match, so no Θ bound exists

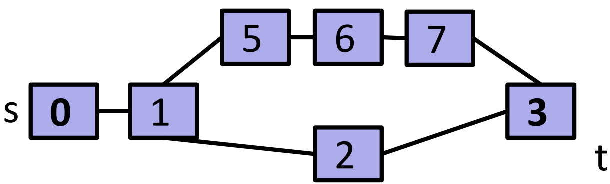
Lecture Outline

- Graph Representations
- **Graph Traversals: BFS**
- Shortest Paths: Dijkstra's Algorithm

Review: Depth-First Search

- DFS goes "deep" instead of "broad"
  - 0, 1, 2, 3
  - 0, 1, 5, 6, 7, 3

} both are possible DFS paths

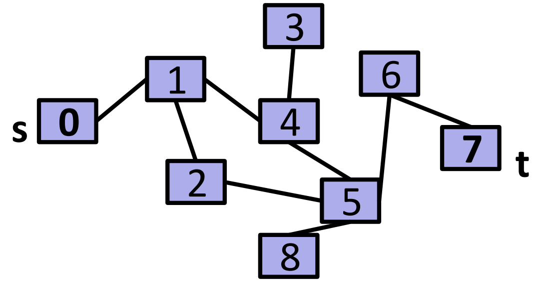



```
connected(Node s, Node t) {
  if (s == t) {
    return true;
  } else {
    s.visited = true;
    for (Node n : s.neighbors) {
      if (n.visited) {
        continue;
      }
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```

Breadth-First Search (1 of 2)

- Breadth-First Search (BFS) is the graph analogue of a tree's level-order traversal
  - Goes "broad" instead of "deep"
  - Added benefit: finds the shortest path from as source to all other vertices, not just a single target t!

- Challenge: how would you implement BFS from s?
  - Hint 1: How will you visit vertices in BFS order?
  - Hint 2: You'll need to use some kind of data structure
  - Hint 3: Don't use recursion



Breadth-First Search (2 of 2)

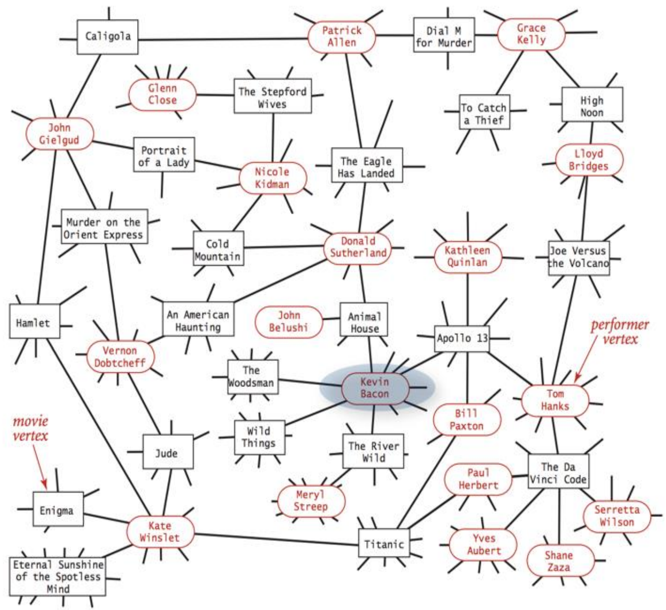
- Demo: https://docs.google.com/presentation/d/1JoYCelH4YE6IkSMq_LfTJMzJ00WxDj7rEa49gYmAtc4/present?ueb=true&slide=id.g76e0dad85_2_380

- Note: we call the need-to-explore vertices "the fringe"

```
shortestPaths(Node s, Graph g) {
  Queue fringe;
  fringe.enqueue(s);

  Map<Node, Integer> distances;
  distances.setAll(g.allNodes(), ∞);

  Map<Node, Node> previousNode;

  while (! fringe.isEmpty()) {
    Node n = fringe.dequeue();
    for (Node i : n.neighbors) {
      if (distances[i]
        < distances[n] + 1) {
        continue;
      } else {
        distances[i] =
          distances[n] + 1;
        fringe.enqueue(i);
        previousNode[i] = n;
      }}}}
```

Breadth-First Search Application

- Graph with vertices as actors and edges as movies
- Perform BFS from Kevin Bacon (or your actor of choice)
  - https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon
  - https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Bacon_number

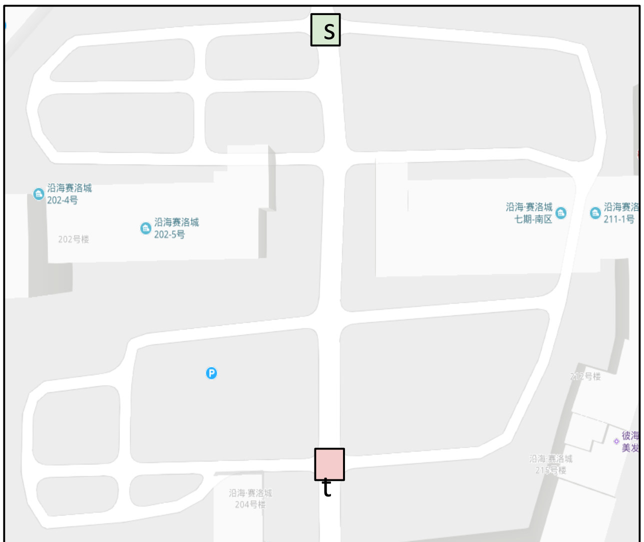


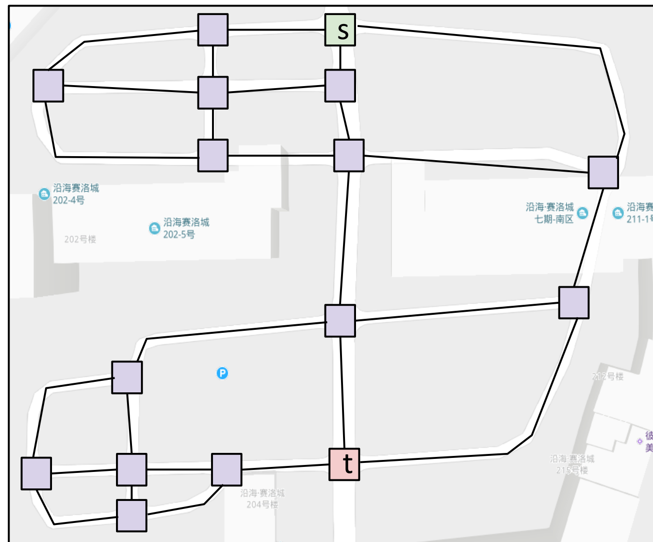
Lecture Outline

- Graph Representations
- Graph Traversals: BFS
- **Shortest Paths: Dijkstra's Algorithm**

Shortest Paths in Weighted Graphs

- Breadth-First Search (BFS) finds the shortest path from as source to all other vertices in an unweighted graph
- Let’s try it in a map

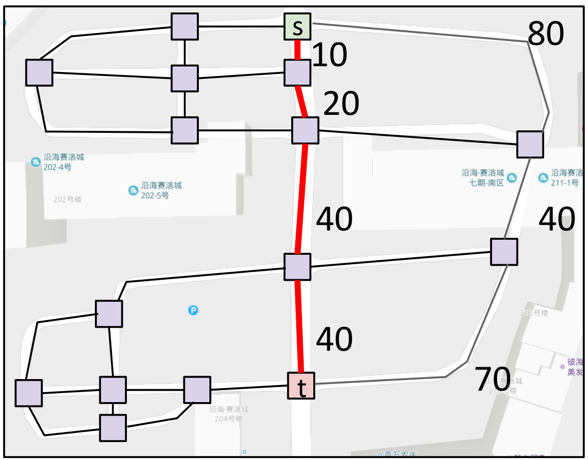




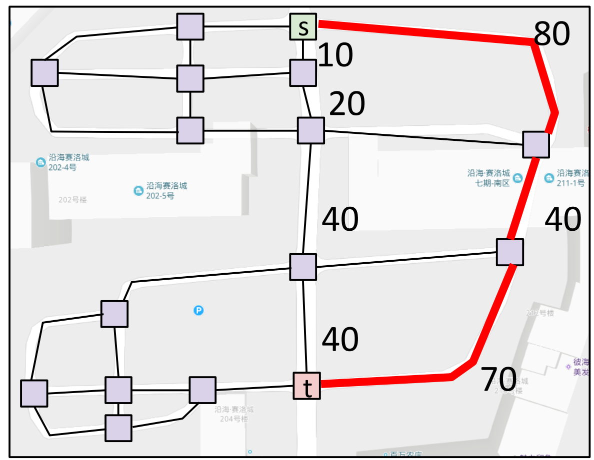
Breadth First Search for Mapping Applications

- BFS yields the wrong route from s to t
  - Length of ~190 instead of ~110
  - We need an algorithm that takes into account edge weights!

Correct Result



BFS Result



Adapting BFS

- Can we adapt BFS to accommodate edge weights?

```
shortestPaths(Node s, Graph g) {
  Queue fringe;
  fringe.enqueue(s);

  Map<Node, Integer> distances;
  distances.setAll(g.allNodes(), ∞);

  Map<Node, Node> previousNode;

  while (! fringe.isEmpty()) {
    Node n = fringe.dequeue();
    for (Node i : n.neighbors) {
      if (distances[i]
        < distances[n] + 1) {
        continue;
      } else {
        distances[i] =
          distances[n] + 1;
        fringe.enqueue(i);
        previousNode[i] = n;
      }}}}
```

"ijk" spells "Dijkstra"

Dijkstra's Algorithm

```
dijkstras(Node s, Graph g) {
  PriorityQueue unvisited;
  unvisited.addAll(g.allNodes(), ∞)
  unvisited.changePriority(s, 0);

  Map<Node, Integer> distances;
  Map<Node, Node> previousNode

  while (! unvisited.isEmpty()) {
    Node n = unvisited.removeMin();
    for (Node i : n.neighbors) {
      if (distances[i]
        < distances[n]
          + g.edgeWeight(n, i)) {
        continue;
      } else {
        distances[i] = distances[n]
          + g.edgeWeight(n, i);
        unvisited.changePriority(i,
          distances[i]);
        previousNode[i] = n;
      }}}}
```

```
shortestPaths(Node s, Graph g) {
  Queue fringe;
  fringe.enqueue(s);


  Map<Node, Integer> distances;
  distances.setAll(g.allNodes(), ∞);
  Map<Node, Node> previousNode;

  while (! fringe.isEmpty()) {
    Node n = fringe.dequeue();
    for (Node i : n.neighbors) {
      if (distances[i]
        < distances[n] + 1) {
        continue;

      } else {
        distances[i] =
          distances[n] + 1;
        fringe.enqueue(i);

        previousNode[i] = n;
      }}}}
```

Dijkstra's Algorithm

- Replace the *queue* with a *priority queue* whose priorities are distance from s
- The key operation: "relaxing" an edge on the fringe
- Dijkstra's returns optimal results only in graphs with non-negative edge weights!

```
dijkstras(Node s, Graph g) {
  PriorityQueue unvisited;
  unvisited.addAll(g.allNodes(), ∞)
  unvisited.changePriority(s, 0);

  Map<Node, Integer> distances;
  Map<Node, Node> previousNode

  while (! unvisited.isEmpty()) {
    Node n = unvisited.removeMin();
    for (Node i : n.neighbors) {
      if (distances[i]
        < distances[n]
          + g.edgeWeight(n, i)) {
        continue;
      } else {
        distances[i] = distances[n]
          + g.edgeWeight(n, i);
        unvisited.changePriority(i,
          distances[i]);
        previousNode[i] = n;
      }}}}
```

Dijkstra's Algorithm: Why It Works

- Invariants ("something that always holds true")
  - `distances` contains the best known total distance from s to any v
    - If v has a finite distance, then it must be the *optimal* distance
  - `unvisit` contains all unvisited vertices, ordered by distance from s
    - Thus, we visit vertices in order of total distance from s

- Proof sketch
  - Base case: distances[s] = 0, which is optimal
  - Inductive step: after relaxing all edges from s, let v be the minimum vertex in unvisit. Claim: distances[v] is optimal
    - Proof by contradiction: there is an unvisited path e_1, e_2, … e_n whose sum is less than w. However if that were true, we would have visited each of e_1, e_2, … e_n already *or* one of those edges has a negative weight. Therefore our claim is true

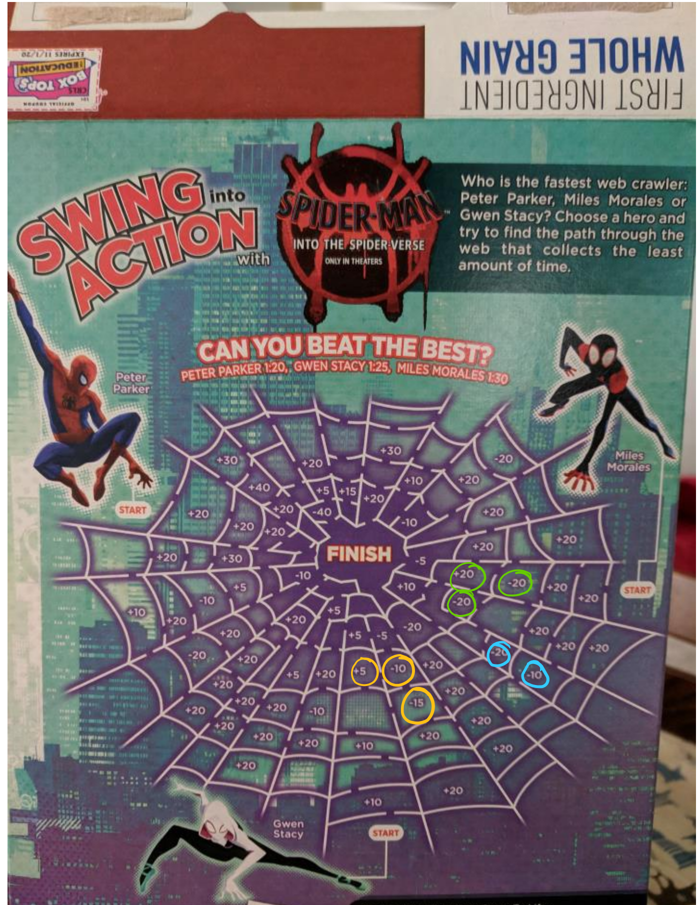
Dijkstra's Algorithm: Demo

- Demo: https://docs.google.com/presentation/d/1_bw2z1ggUkquPdhl7gwdVBoTaoJmaZdpkV6MoAgxlJc/pub?start=false&loop=false&delayms=3000

Negative Weights vs Negative Cycles

- Negative weights: Dijkstra's won't guarantee correct results
  - But other algorithms might

- Negative cycles: no algorithm can find a finite optimal path
  - Because you can always decrease the path cost by going through the negative cycle a few more times



Dijkstra's Algorithm: Runtime

- Assuming a binary heap implementation

	# Operations	Cost per operation	Total Cost
PQ add	V	O(log V)	O(V log V)
PQ removeMin	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

- Runtime: O(V logV + V logV + E logV)
  - Assuming E > V, Dijkstra's is O(E log V)

tl;dr

- Graph implementations have an impact on algorithm runtime
- BFS and Dijkstra's search "shallow" nodes before searching "deep" nodes, whereas DFS searches "deep" nodes first
- BFS finds optimal paths in an unweighted graph; Dijkstra's in a weighted (non-negative) graph