Graphs

League of Programmers

ACA, IIT Kanpur

Outline

What are Graphs?

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- Graph G=(V,E) , $E \subseteq V * V$

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- Graph G=(V,E) , $E \subseteq V * V$
- We will label the nodes from 1 to n (V)

Basics

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- Graph G=(V,E) , $E \subseteq V * V$
- We will label the nodes from 1 to n (V)
- m edges connect some pairs of nodes (E)

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- Graph G=(V,E) , $E \subseteq V * V$
- We will label the nodes from 1 to n (V)
- m edges connect some pairs of nodes (E)
- Edges can be either one-directional (directed) or bidirectional

What are Graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- Graph G=(V,E) , $E \subseteq V * V$
- We will label the nodes from 1 to n (V)
- m edges connect some pairs of nodes (E)
- Edges can be either one-directional (directed) or bidirectional
- Nodes and edges can have some auxiliary information

Terminologies

Terminologies

- Vertex: node of a graph

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$
- Degree(u) $= |$Adjacency(u)$|$

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$
- Degree(u) $= |$Adjacency(u)$|$
- Subgraph: A subset of vertices and edges is a subgraph

Basics

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$
- Degree(u) $= |$Adjacency(u)$|$
- Subgraph: A subset of vertices and edges is a subgraph
- Walk: A sequence $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$

Basics

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$
- Degree(u) $= |$Adjacency(u)$|$
- Subgraph: A subset of vertices and edges is a subgraph
- Walk: A sequence v_1, v_2, $\ldots$, v_k such that $(v_i, v_{i+1}) \in E$
- Trial: A trial is a walk in which no edge occurs twice

Basics

Terminologies

- Vertex: node of a graph
- Adjacency(u) $= \{v | (u, v) \in E\}$
- Degree(u) $= |$Adjacency(u)$|$
- Subgraph: A subset of vertices and edges is a subgraph
- Walk: A sequence v_1, v_2, . . . , v_k such that $(v_i, v_{i+1}) \in E$
- Trial: A trial is a walk in which no edge occurs twice
- Closed path: A walk where starting and ending vertex are the same

Storing Graphs

Storing Graphs

- We need to store both the set of nodes V and the set of edges E

Storing Graphs

- We need to store both the set of nodes V and the set of edges E
- Nodes can be stored in an array. Edges must be stored in some other way.

Storing Graphs

- We need to store both the set of nodes V and the set of edges E
- Nodes can be stored in an array. Edges must be stored in some other way.
- We want to support the following operations

Storing Graphs

- We need to store both the set of nodes V and the set of edges E
- Nodes can be stored in an array. Edges must be stored in some other way.
- We want to support the following operations
  - Retrieving all edges incident to a particular node

Storing Graphs

- We need to store both the set of nodes V and the set of edges E
- Nodes can be stored in an array. Edges must be stored in some other way.
- We want to support the following operations
  - Retrieving all edges incident to a particular node
  - Testing if given two nodes are directly connected

Adjacency Matrix

Basics

Adjacency Matrix

- An easy way to store connectivity information

Adjacency Matrix

- An easy way to store connectivity information
- Checking if two nodes are directly connected: $O(1)$ time

Adjacency Matrix

- An easy way to store connectivity information
- Checking if two nodes are directly connected: O(1) time
- Make an n x n matrix A

Adjacency Matrix

- An easy way to store connectivity information
- Checking if two nodes are directly connected: O(1) time
- Make an n x n matrix A
- a[i][j] = 1 if there is an edge from i to j

Adjacency Matrix

- An easy way to store connectivity information
- Checking if two nodes are directly connected: O(1) time
- Make an n x n matrix A
- a[i][j] = 1 if there is an edge from i to j
- a[i][j] = 0 otherwise

Basics

Adjacency Matrix

- An easy way to store connectivity information
- Checking if two nodes are directly connected: O(1) time
- Make an n x n matrix A
- a[i][j] = 1 if there is an edge from i to j
- a[i][j] = 0 otherwise
- Uses O(n^2) memory. So, use when n is less than a few thousands, AND when the graph is dense

Adjacency List

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths

Basics

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths
- We can use: vector< vector<int> >

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths
- We can use: vector< vector<int> >
- Space usage: $O(n + m)$

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths
- We can use: vector< vector<int> >
- Space usage: $O(n + m)$
- Checking if edge (Vi,Vj) is present in G:

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths
- We can use: vector< vector<int> >
- Space usage: $O(n + m)$
- Checking if edge (Vi,Vj) is present in G:

Adjacency List

- Each vertex maintains a list of vertices that are adjacent to it.
- Lists have variable lengths
- We can use: vector< vector<int> >
- Space usage: O(n + m)
- Checking if edge (Vi,Vj) is present in G: O(min(deg(Vi),deg(Vj)))

Special Graphs

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)
  - connected graph with n-1 edges

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)
  - connected graph with n-1 edges
  - An acyclic graph with n-1 edges

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)
  - connected graph with n-1 edges
  - An acyclic graph with n-1 edges
  - There is exactly one path between every pair of nodes

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)
  - connected graph with n-1 edges
  - An acyclic graph with n-1 edges
  - There is exactly one path between every pair of nodes
  - An acyclic graph but adding any edge results in a cycle

Special Graphs

- **Implicit graphs**
  Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.
- **Tree**: a connected acyclic graph
  The most important type of graph in CS
  Alternate definitions (all are equivalent!)
  - connected graph with n-1 edges
  - An acyclic graph with n-1 edges
  - There is exactly one path between every pair of nodes
  - An acyclic graph but adding any edge results in a cycle
  - A connected graph but removing any edge disconnects it

Special Graphs

Special Graphs

- **Directed Acyclic Graph (DAG)**

Basics

Special Graphs

- **Directed Acyclic Graph (DAG)**
- **Bipartite Graph**

  Nodes can be separated into two groups S and T such that edges exist between S and T only (no edges within S or within T)

Outline

Traversal

Graph Traversal

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- We will cover two algorithms

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- We will cover two algorithms
  - Depth-First Search (DFS): uses recursion

Traversal

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- We will cover two algorithms
  - Depth-First Search (DFS): uses recursion
  - Breadth-First Search (BFS): uses queue

Traversal

DFS

Traversal

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
- Replace recursive calls with a stack

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
- Replace recursive calls with a stack
- Complexity

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
- Replace recursive calls with a stack
- Complexity

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
- Replace recursive calls with a stack
- Complexity
  Time: O(|V|+|E|)

DFS

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
- Replace recursive calls with a stack
- Complexity
  Time: O(|V|+|E|)
  Space: O(|V|) [to maintain the vertices visited till now]

DFS: Uses

Traversal

DFS: Uses

- Biconnected components

DFS: Uses

- Biconnected components
- A node in a connected graph is called an articulation point if the deletion of that node disconnects the graph.

DFS: Uses

- Biconnected components
- A node in a connected graph is called an articulation point if the deletion of that node disconnects the graph.
- A connected graph is called biconnected if it has no articulation points. That is, the deletion of any single node leaves the graph connected.

BFS

BFS(v): visits all the nodes reachable from v in breadth-first order

BFS

BFS(v): visits all the nodes reachable from v in breadth-first order

- Initialize a queue Q

Traversal

BFS

BFS(v): visits all the nodes reachable from v in breadth-first order

- Initialize a queue Q
- Mark v as visited and push it to Q

Traversal

BFS

BFS(v): visits all the nodes reachable from v in breadth-first order

- Initialize a queue Q
- Mark v as visited and push it to Q
- While Q is not empty:
  - Take the front element of Q and call it w
  - For each edge $w \rightarrow u$:
    - If u is not visited, mark it as visited and push it to Q

Traversal

BFS

BFS(v): visits all the nodes reachable from v in breadth-first order

- Initialize a queue Q
- Mark v as visited and push it to Q
- While Q is not empty:
  Take the front element of Q and call it w
  For each edge $w \to u$:
    If u is not visited, mark it as visited and push it to Q
- Same Time and Space Complexity as DFS

BFS: Uses

BFS: Uses

- Finding a Path with Minimum Number of edges from starting vertex to any other vertex.

BFS: Uses

- Finding a Path with Minimum Number of edges from starting vertex to any other vertex.
- Solve Shortest Path problem in unweighted graphs

BFS: Uses

- Finding a Path with Minimum Number of edges from starting vertex to any other vertex.
- Solve Shortest Path problem in unweighted graphs
- Spoj Problem
  `http://www.spoj.pl/problems/PPATH/`

Topological Sort

- Input: a DAG $G = V, E$

Topological Sort

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers
- Pseudocode

Topological Sort

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers
- Pseudocode
  - Precompute the number of incoming edges deg(v) for each node v

Topological Sort

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers
- Pseudocode
  - Precompute the number of incoming edges deg(v) for each node v
  - Put all nodes with zero degree into a queue Q

Topological Sort

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers
- Pseudocode
  - Precompute the number of incoming edges deg(v) for each node v
  - Put all nodes with zero degree into a queue Q
  - Repeat until Q becomes empty:

      Take v from Q

      For each edge $v \to u$

        Decrement deg(u) (essentially removing the edge $v \to u$)

        If deg u becomes zero, push u to Q

Topological Sort

- Input: a DAG G = V, E
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v. There can be many answers
- Pseudocode
  - Precompute the number of incoming edges deg(v) for each node v
  - Put all nodes with zero degree into a queue Q
  - Repeat until Q becomes empty:

    Take v from Q

    For each edge $v \to u$

    Decrement deg(u) (essentially removing the edge $v \to u$)

    If deg u becomes zero, push u to Q
- Time complexity: $O(n + m)$

Outline

Minimum Spanning Tree

Minimum Spanning Tree

- Input: An undirected weighted graph $G = V, E$

Minimum Spanning Tree

- Input: An undirected weighted graph $G = V, E$
- Output: A subset of E with the minimum total weight that connects all the nodes into a tree

Minimum Spanning Tree

- Input: An undirected weighted graph $G = V, E$
- Output: A subset of E with the minimum total weight that connects all the nodes into a tree
- There are two algorithms:

Minimum Spanning Tree

- Input: An undirected weighted graph $G = V, E$
- Output: A subset of E with the minimum total weight that connects all the nodes into a tree
- There are two algorithms:
  - Kruskal's algorithm

- Input: An undirected weighted graph $G = V, E$
- Output: A subset of E with the minimum total weight that connects all the nodes into a tree
- There are two algorithms:
  - Kruskal's algorithm
  - Prim's algorithm

Kruskal's Algorithm

Kruskal's Algorithm

- Main idea: the edge e with the smallest weight has to be in the MST

- Main idea: the edge e with the smallest weight has to be in the MST
- Keep different supernodes, which are "local MST's" and then join them by adding edges to form the MST for the whole graph

- Main idea: the edge e with the smallest weight has to be in the MST
- Keep different supernodes, which are "local MST's" and then join them by adding edges to form the MST for the whole graph
- Pseudocode:

```
Sort the edges in increasing order of weight
Repeat until there is one supernode left:
  Take the minimum weight edge e*
  If e* connects two different supernodes:
    Connect them and merge the supernodes
  Otherwise,
    ignore e*
```

Prim's Algo

Reading Homework

Outline

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- All pair shortest distance

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time
- Algorithm

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j
    - f(i, j, 0) = cost(i, j)

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in O(n^3) time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 ... k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j
    - f(i, j, 0) = cost(i, j)
  - The optimal path for f i, j, k may or may not have k as an intermediate node

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in O(n^3) time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j
    - f(i, j, 0) = cost(i, j)
  - The optimal path for f i, j, k may or may not have k as an intermediate node
    - If it does, f (i, j, k) = f (i,k k-1) + f(k, j, k-1)

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in $O(n^3)$ time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j
    - f(i, j, 0) = cost(i, j)
  - The optimal path for f i, j, k may or may not have k as an intermediate node
    - If it does, f (i, j, k) = f (i,k k-1) + f(k, j, k-1)
    - Otherwise, f (i, j, k) = f (i, j, k-1)

Floyd-Warshall Algorithm

- All pair shortest distance
- Runs in O(n^3) time
- Algorithm
  - Define f(i, j, k) as the shortest distance from i to j, using 1 … k as intermediate nodes
    - f(i, j, n) is the shortest distance from i to j
    - f(i, j, 0) = cost(i, j)
  - The optimal path for f i, j, k may or may not have k as an intermediate node
    - If it does, f (i, j, k) = f (i,k k-1) + f(k, j, k-1)
    - Otherwise, f (i, j, k) = f (i, j, k-1)
  - Therefore, f (i, j, k) is the minimum of the two quantities above

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- Pseudocode:

```
Initialize D to the given cost matrix
For k = 1 ...n:
For all i and j:
```

$$d_{ij} = min\{d_{ij}, d_{ik} + d_{kj}\}$$

Floyd-Warshall Algorithm

- Pseudocode:

```
Initialize D to the given cost matrix
For k = 1 ...n:
For all i and j:
```

  $d_{ij} = min\{d_{ij}, d_{ik} + d_{kj}\}$

- Can also be used to detect negative weight cycles in graph?

Floyd-Warshall Algorithm

- Pseudocode:

```
Initialize D to the given cost matrix
For k = 1 ...n:
For all i and j:
```

  $$d_{ij} = min\{d_{ij}, d_{ik} + d_{kj}\}$$

- Can also be used to detect negative weight cycles in graph?

Floyd-Warshall Algorithm

- Pseudocode:

```
Initialize D to the given cost matrix
For k = 1 ...n:
For all i and j:
```

  $d_{ij} = min\{d_{ij}, d_{ik} + d_{kj}\}$

- Can also be used to detect negative weight cycles in graph? How?

Floyd-Warshall Algorithm

- Pseudocode:

```
Initialize D to the given cost matrix
For k = 1 ...n:
For all i and j:
```

  $$d_{ij} = min\{d_{ij}, d_{ik} + d_{kj}\}$$

- Can also be used to detect negative weight cycles in graph? How?
  If $d_{ij} + d_{ji} < 0$ for some i and j, then the graph has a negative weight cycle

Dijkstra's Algorithm

Dijkstra's Algorithm

- Used to solve Single source Shortest Path problem in Weighted Graphs

- Used to solve Single source Shortest Path problem in Weighted Graphs
- Only for Graphs with positive edge weights.

Dijkstra's Algorithm

- Used to solve Single source Shortest Path problem in Weighted Graphs
- Only for Graphs with positive edge weights.
- The algorithm finds the path with lowest cost (i.e. the shortest path) between that source vertex and every other vertex

Dijkstra's Algorithm

- Used to solve Single source Shortest Path problem in Weighted Graphs
- Only for Graphs with positive edge weights.
- The algorithm finds the path with lowest cost (i.e. the shortest path) between that source vertex and every other vertex
- Greedy strategy

Dijkstra's Algorithm

- Used to solve Single source Shortest Path problem in Weighted Graphs
- Only for Graphs with positive edge weights.
- The algorithm finds the path with lowest cost (i.e. the shortest path) between that source vertex and every other vertex
- Greedy strategy
- Idea: Find the closest node to s, and then the second closest one, then the third, etc

Dijkstra's Algorithm

Dijkstra's Algorithm

- Pseudo code:

- Pseudo code:
  - Maintain a set of nodes S, the shortest distances to which are decided

- Pseudo code:
  - Maintain a set of nodes S, the shortest distances to which are decided
  - Also maintain a vector d, the shortest distance estimate from s

Dijkstra's Algorithm

- Pseudo code:
  - Maintain a set of nodes S, the shortest distances to which are decided
  - Also maintain a vector d, the shortest distance estimate from s
  - Initially, S = s, and d_v = cost(s, v)

Dijkstra's Algorithm

- Pseudo code:
  - Maintain a set of nodes S, the shortest distances to which are decided
  - Also maintain a vector d, the shortest distance estimate from s
  - Initially, S = s, and d_v = cost(s, v)
  - Repeat until S = V:

    Find v $\notin$ S with the smallest d_v, and add it to S

    For each edge $v \rightarrow u$ of cost c:

    $d_u = min\{d_u, d_v + c\}$

Dijkstra's Algorithm

Dijkstra's Algorithm

- Time complexity depends on the implementation:
  Can be O(n^2 + m), O(m log n), O(n log n)

Dijkstra's Algorithm

- Time complexity depends on the implementation:
  Can be $O(n^2 + m)$, O(m log n), O(n log n)
- Use priority_queue<node> for implementing Dijkstra's

Dijkstra's Algorithm

- Time complexity depends on the implementation:
  Can be O(n^2 + m), O(m log n), O(n log n)
- Use priority_queue<node> for implementing Dijkstra's
- SPOJ Problem
  `http://www.spoj.pl/problems/CHICAGO`

Bellman-Ford Algorithm

Bellman-Ford Algorithm

- Single source shortest path for negative weights

Bellman-Ford Algorithm

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles
- Pseudo code:

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles
- Pseudo code:
  - Initialize $d_s = 0$ and $d_v = \infty \; \forall v \neq s$

Bellman-Ford Algorithm

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles
- Pseudo code:
  - Initialize $d_s = 0$ and $d_v = \infty \ \forall v \neq s$
  - For k = 1 . . . n-1:

Bellman-Ford Algorithm

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles
- Pseudo code:
  - Initialize $d_s = 0$ and $d_v = \infty \; \forall v \neq s$
  - For k = 1 ... n-1:
  - For each edge $u \rightarrow v$ of cost c:
    $d_v = min\{d_v, d_u + c\}$

Bellman-Ford Algorithm

- Single source shortest path for negative weights
- Can also be used to detect negative weight cycles
- Pseudo code:
  - Initialize $d_s = 0$ and $d_v = \infty \ \forall v \neq s$
  - For k = 1 ... n-1:
  - For each edge $u \to v$ of cost c:

    $d_v = min\{d_v, d_u + c\}$
- Runs in O(nm) time

Outline

Problems

Links:

1. http://www.spoj.pl/problems/IOPC1201/
2. http://www.spoj.pl/problems/TRAFFICN/
3. http://www.spoj.pl/problems/PFDEP/
4. http://www.spoj.pl/problems/PRATA/
5. http://www.spoj.pl/problems/ONEZERO/
6. http://www.spoj.pl/problems/PPATH/
7. http://www.spoj.pl/problems/PARADOX/
8. http://www.spoj.pl/problems/HERDING/
9. http://www.spoj.pl/problems/PT07Z/