CS141: Intermediate Data Structures and Algorithms

Graphs

Amr Magdy
Graph Data Structure

- A set of nodes (vertices) and edges connecting them
Graph Applications

- Road network
- Social media networks
- Knowledge bases
Graph Representations

- Adjacency matrix
  - Storage and access efficient when many edges exist
Graph Representations

- Adjacency matrix
  - Storage and access efficient when many edges exist
Graph Representations

- Incidence Matrix
  - Expensive storage, not popular
Graph Representations

- Adjacency list
  - Storage efficient when few edges exit (sparse graphs)
  - Sequential access to edges (vs random access in matrix)
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest

Fig(i): Connected Graph

Fig(ii): Unconnected Graph

There are three component of above unconnected graph
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest

Acyclic Graph

Cyclic Graph
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest
  - Tree: directed acyclic graph with max of one path between any two nodes
  - Forest: set of disjoint trees
Basic Graph Algorithms

- Graph traversal algorithms
  - Bread-first Search (BFS)
  - Depth-first Search (DFS)
- Topological Sort
- Graph Connectivity
- Cycle Detection
Breadth-first Search (BFS)

- How to traverse?
Breadth-first Search (BFS)

- How to traverse?
- Use a queue
Breadth-first Search (BFS)

- How to traverse?
- Use a queue
- Start at a vertex s
  - Mark s as visited
  - Enqueue neighbors of s
  - while Q not empty
    - Dequeue vertex u
    - Mark u as visited
    - Enqueue unvisited neighbors of u
Breadth-first Search (BFS)
Depth-first Search (DFS)

- How to traverse?
Depth-first Search (DFS)

- How to traverse?
- Use a stack
Depth-first Search (DFS)

- How to traverse?
- Use a stack
- Start at a vertex $s$
  - Mark $s$ as visited
  - Push neighbors of $s$
  
while Stack not empty
  - Pop vertex $u$
  - Mark $u$ as visited
  - Push unvisited neighbors of $u$
Complexity of Graph Traversal

For $G = (V,E)$, $V$ set of vertices, $E$ set of edges

- **BFS**
  - Time: $O(|V|+|E|)$
  - Space: $O(|V|)$ (plus graph representation)

- **DFS**
  - $O(|V|+|E|)$
  - Space: $O(|V|)$ (plus graph representation)
Graph Connectivity

- Checking if graph is connected:
Graph Connectivity

Checking if graph is connected: IsConnected(G)
{
    DFS(G)
    if any vertex not visited
        return false
    else
        return true
}

Time Complexity: \( O(|V|+|E|) \)
Graph Connected Components

Getting the graph connected components

Fig(ii):
Unconnected Graph

There are three component of above unconnected graph
Graph Connected Components

- Getting the graph connected components
- Mark all nodes as unvisited
  \[\text{visitCycle} = 1\]
  \[\text{while( there exists unvisited node n) \{ \}
  \]
  - Start DFS(G) at n, mark visited node with visitCycle
  - Output all nodes with current visitCycle as one connected component
  - visitCycle = visitCycle+1
  \[\}
\]

Time Complexity: \(O(|V|+|E|)\)
Cycle Detection

- Does a connected graph G contain a cycle? (non-trivial cycle)
- General idea: if DFS procedure tries to revisit a visited node, then there is a cycle
Does a graph G contain a cycle? (non-trivial cycle)

IsAcyclic(G) {
    Start at unvisited vertex s
    Mark “s” as visited
    Push neighbors of s in stack
    while stack not empty
        Pop vertex u
        Mark u as visited
        if u has visited neighbors
            return true
        Push unvisited neighbors of u
    return false
}
Cycle Detection in Directed Graphs

\[
\text{visitFlag} = 1 \\
\text{while there exist unvisited node n} \{ \\
\quad - \text{Call IsAcyclic}(G) \text{ with start node n and visitFlag} \\
\quad - \text{visitFlag} = \text{visitFlag} + 1 \\
\}\]

IsAcyclic pseudo code will be modified to have:

\[
\text{if u has visited neighbors marked with visitFlag} \\
\text{return true}
\]
Topological Sort

- Determine a linear order for vertices of a directed acyclic graph (DAG)
  - Mostly dependency/precedence graphs
  - If edge \((u,v)\) exists, then \(u\) appears before \(v\) in the order
Topological Sort

L ← Empty list
S ← Set of all nodes with no incoming edge

while S is non-empty do
    remove a node n from S
    add n to end of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
    return L (a topologically sorted order)
Spanning Tree

Given a connected graph $G=(V,E)$, a spanning tree $T \subseteq E$ is a set of edges that “spans” (i.e., connects) all vertices in $V$.

A **Minimum Spanning Tree (MST)**: a spanning tree with minimum total weight on edges of $T$

Application:
- The wiring problem in hardware circuit design
Spanning Tree: Example

A --- 1 --- B
  
  3 4

D --- 4 --- E
  
  4 2

  7

  F --- 5 --- C
Spanning Tree: Not MST

Total weight = 21
Spanning Tree: MST

Total weight = 16
Spanning Tree: Another MST

Total weight = 16
Finding MST: Kruskal’s algorithm

- Sort all the edges by weight
- Scan the edges by weight from lowest to highest
- If an edge introduces a cycle, drop it
- If an edge does not introduce a cycle, pick it
- Terminate when $n-1$ edges are picked
(n: number of vertices)
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm
Finding MST: Kruskal’s algorithm

A

B

1

C

D

E

2

4

F

3
Finding MST

› Kruskal’s algorithm: greedy
  › Greedy choice: least weighted edge first
  › Complexity: $O(E \log E)$ – sorting edges by weight
  › Edge-cycle detection: $O(1)$ using hashing of $O(V)$ space

› Prim’s algorithm: greedy
  › Complexity: $O(E + V \log V)$ – using Fibonacci heap data structure
Shortest Paths in Graphs

- Given graph $G=(V,E)$, find shortest paths from a given node $source$ to all nodes in $V$. (Single-source All Destinations)
Shortest Paths in Graphs

- Given graph $G=(V,E)$, find shortest paths from a given node $source$ to all nodes in $V$. (Single-source All Destinations)

- If negative weight cycle exist from $s \rightarrow t$, shortest is undefined
  - Can always reduce the cost by navigating the negative cycle

- If graph with all +ve weights $\rightarrow$ Dijkstra’s algorithm

- If graph with some -ve weights $\rightarrow$ Bellman-Ford’s algorithm
Dijkstra’s Algorithm

Initialize:

\[ Q: \ | A | B | C | D | E \]
\[ 0 | \infty | \infty | \infty | \infty \]

\[ S: \ \{ \} \]

\[ \text{Prev: \{A, U, U, U, U, U\}} \]
Dijkstra’s Algorithm

Graph representation with weighted edges:
- A to B: 10
- B to C: 1
- B to D: 2
- C to D: 4
- C to E: 8
- D to E: 7
- A to E: 3

Priority Queue (Q):
- A: 0
- B: ∞
- C: ∞
- D: ∞
- E: ∞
Dijkstra’s Algorithm

\[ Q: \begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty
\end{array} \]

\[ S: \{ A \} \]

\[ \text{Prev: \{A,A,A,U,U\}} \]
Dijkstra’s Algorithm

Q: \( \begin{array}{cccccc} A & B & C & D & E \\ 0 & \infty & \infty & \infty & \infty \\ 10 & 3 & \infty & \infty & \infty \end{array} \)

S: \{A, C\}

Prev: \{A,A,A,U,U\}
Dijkstra’s Algorithm

\[ Q: \begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
7 & 11 & 5 & & & \\
\end{array} \]

\[ S: \{ A, C \} \]

\[ \text{Prev: } \{ A, C, A, C, C, C \} \]
Dijkstra’s Algorithm

$Q:\begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
7 & 11 & 5 & & \\
\end{array}$

$S: \{ A, C, E \}$

$Prev: \{ A, C, A, C, C \}$
Dijkstra’s Algorithm

\[ Q: \begin{array}{cccccc} A & B & C & D & E \\ 0 & \infty & \infty & \infty & \infty \\ 10 & 3 & 8 & 8 & 8 \\ 7 & 11 & 5 & \end{array} \]

\[ S: \{A, C, E\} \]
\[ \text{Prev: \{A,C,A,C,C\}} \]
Dijkstra’s Algorithm

$Q: \begin{array}{cccccc} & A & B & C & D & E \\ 0 & \infty & \infty & \infty & \infty & 8 \\ 10 & 3 & \infty & 8 & 8 \\ 7 & 3 & 11 & 11 & 5 \end{array}$

$S: \{ A, C, E, B \}$

$Prev: \{ A, C, A, C, C \}$
Dijkstra’s Algorithm

\[ Q: \begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
7 & 7 & 11 & 11 & 5 \\
\end{array} \]

\[ S: \{ A, C, E, B \} \]

\[ \text{Prev: \{A,C,A,B,C\}} \]
Dijkstra’s Algorithm

Q:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

S: { A, C, E, B, D }
Prev: { A, C, A, B, C }
Dijkstra’s Algorithm

A: $A \to A$
B: $A \to C \to B$
C: $A \to C$
D: $A \to C \to B \to D$
E: $A \to C \to E$

$Q: \begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 7 & 3 & \infty & \infty \\
7 & 7 & 11 & 11 & 5 \\
\end{array}

S: \{ A, C, E, B, D \}
Prev: \{A,C,A,B,C\}
Dijkstra’s Algorithm

```python
function Dijkstra(Graph, source):
    create vertex set Q

    for each vertex v in Graph:   //Initialization
        Dist[v] ← INFINITY       //Unknown distance from source to v
        Prev[v] ← UNDEFINED      //Previous node in path from source to v
        add v to Q                //All nodes initially unvisited (in Q)

    Dist[source] ← 0             // Distance from source to source = 0
    Prev[source] ← source

    while Q is not empty:
        u ← vertex in Q with min Dist[u]  //Node with the least distance
                                       // will be selected first
        remove u from Q

        for each neighbor v of u in Q:   //v is still in Q.
            tmp ← Dist[u] + edge_length(u, v)  //trying u as “source->u->v”
            if tmp < Dist[v]:                 //A shorter path to v has been found
                Dist[v] ← tmp
                Prev[v] ← u

    return Dist[], S[]
```
What is the maximum amount we can ship from Vancouver to Winnipeg?
Network Max Flow

- What the maximum amount we can ship from Vancouver to Winnipeg?
- Pseudo code

```pseudo
MaxFlow(G, s, t) {
    max_flow = 0
    while (∃ a simple path p:s→t){
        curr_flow = min weight in p
        max_flow = max_flow + curr_flow
        for each (edge e ∈ p) {
            e.weight = e.weight - curr_flow
        }
    }
    return max_flow
}
```
Network Max Flow

- What is the maximum amount we can ship from Vancouver to Winnipeg?
Network Max Flow

What is the maximum amount we can ship from Vancouver to Winnipeg?

max_flow = 12
Network Max Flow

- What is the maximum amount we can ship from Vancouver to Winnipeg?

max_flow = 12
Network Max Flow

What is the maximum amount we can ship from Vancouver to Winnipeg?

max_flow = 16
Network Max Flow

What is the maximum amount we can ship from Vancouver to Winnipeg?

\[
\begin{align*}
\text{Edmonton} & \\
\text{Saskatoon} & \\
\text{Vancouver} & \\
\text{Winnipeg} & \\
\text{Calgary} & \\
\text{Regina} & \\
\end{align*}
\]

\[\text{max}\_\text{flow} = 16\]
Network Max Flow

What is the maximum amount we can ship from Vancouver to Winnipeg?

max_flow = 16
Network Max Flow

What is the maximum amount we can ship from Vancouver to Winnipeg?

\[
\text{max}_\text{flow} = 23
\]
Network Max Flow

- What is the maximum amount we can ship from Vancouver to Winnipeg?

\[ \text{max} \_ \text{flow} = 23 \]
Book Readings & Credits

› Book Readings:
› Ch. 22, 23.2, 24.3, 26.1, 26.2

› Credits:
› Figures:
› Wikipedia
› btechsmartclass.com
› Prof. Ahmed Eldawy notes
› Laksman Veeravagu and Luis Barrera