Graph Algorithms

What is a graph?

V - vertices (or nodes)

E ⊆ V x V - edges

directed / undirected

Representation:

- adjacency matrix
- adjacency lists

Why graphs?

Sum of the degrees:

3 + 3 + 2 + 4 + 4 + 3 + 3 + 4
= 26
= twice the #edges
⇒ |E| = 13
Graph Algorithms

What is a graph?

\[ V - \text{vertices} \]
\[ E \subseteq V \times V - \text{edges} \]

Why graphs?

Representation:
- adjacency matrix
- adjacency lists

adjacency checking: \( O(1) \) \( O(\deg) \)
listing all neighbors: \( O(n) \) \( O(\deg) \)
\[ n = |V| \]
\[ m = |E| \]
\[ m = O(n^2) \]
Graph Algorithms

Graph properties:

- connected (undirected graph):
  - there is a path from every vertex to every other vertex
- cyclic
- ...

Tree - a connected acyclic (undirected) graph
Graph Traversals

Objective: list all vertices reachable from a given vertex s
Breadth-first search (BFS)

Finds all vertices "reachable" from a starting vertex.

Byproduct: computes distances from the starting vertex to every vertex

BFS (G=(V,E), s)
1. seen[v]=false, dist[v]=∞ for every vertex v
2. beg=1; end=2; Q[1]=s; seen[s]=true; dist[s]=0;
3. while (beg<end) do
   4. head=Q[beg];
   5. for every u s.t. (head,u) is an edge and not seen[u] do
   7.     seen[u]=true; end++;
9.     beg++;
Depth-first search (DFS)

Finds all vertices “reachable” from a starting vertex, in a different order than BFS.

**DFS-RUN** (G=(V,E), s)
1. seen[v]=false for every vertex v
2. DFS(s)

**DFS(v)**
1. seen[v]=true
2. for every neighbor u of v
3. if not seen[u] then DFS(u)

---

*Running time:*
- \(O(n+m)\) if adjacency lists
- \(O(n^2)\) if adjacency matrix

Better for sparse graphs: scanning across all unvisited vertices, \(O(m)\)
Applications of DFS: topological sort

Def: A **topological sort** of an acyclic directed graph is an order of vertices such that every edge goes from “left to right.”

**idea 1:**

1. start with empty order
2. while there are vertices
3. find a vertex with no outgoing edges
4. remove all ingoing edges into v
5. place v at the beg. of the order
6. remove v
Applications of DFS: topological sort

TopSort (G=(V,E))
1. for every vertex v
2. seen[v]=false
3. fin[v]=∞
4. time=0
5. for every vertex s
6. if not seen[s] then
7. DFS(s)

DFS(v)
1. seen[v]=true
2. for every neighbor u of v
3. if not seen[u] then
4. DFS(u)
5. time++
6. fin[v]=time (and output v)
Applications of DFS: topological sort

TopSort ( G=(V,E) )
1. for every vertex v
2. seen[v]=false
3. fin[v]=∞
4. time=0
5. for every vertex s
6. if not seen[s] then
7. DFS(s)

DFS(v)
1. seen[v]=true
2. for every neighbor u of v
3. if not seen[u] then
4. DFS(u)
5. time++
6. fin[v]=time (and output v)

What if the graph contains a cycle? Then no top sort exists, can check this in O(n+m) time
Appl. DFS: strongly connected components

Vertices u,v are in the same strongly connected component if there is a (directed) path from u to v and from v to u.
Appl. DFS: strongly connected components

Vertices $u,v$ are in the same strongly connected component if there is a (directed) path from $u$ to $v$ and from $v$ to $u$.

How to find strongly connected components?
Appl. DFS: strongly connected components

**STRONGLY-CONNECTED COMPONENTS ( G=(V,E) )**

1. for every vertex v
2. seen[v]=false
3. fin[v]=1
4. time=0
5. for every vertex s
6. if not seen[s] then
7. DFS(G,s) (the finished-time version)
8. compute $G^T$ by reversing all arcs of G
9. sort vertices by decreasing finished time
10. seen[v]=false for every vertex v
11. for every vertex v do
12. if not seen[v] then
13. output vertices seen by DFS(v)
Many other applications of D/BFS

DFS
• find articulation points
• find bridges

BFS
• e.g. sokoban