**Notes**

* Make sure you write a “return” statement in the algorithm section for written problems.

**Known Graph Problems from Lectures**

**Credit to the person who posted on ED**

**DFS**

DFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root or an arbitrary node of a graph and explores as far as possible along each branch before backtracking.

**Input:**

* G = (V, E)

**Output:**

* ccnum[]
* a topological sort on a DAG (Directed Acyclic Graph).
  + it takes O(1) to access the first or last vertex of the topological sorting.

**More specifically, it outputs:**

* Undirected graph G, where the vertices are labeled by connected component number (ccnum).
* Directed graph G, a list of connected components.

We have access to the prev, pre, and post-arrays.

**Graphs that can use DFS:**

* Unweighted graphs
* Undirected/Directed graphs
* DAGs

**Runtime:**

* O(m + n)

**BFS**

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and explores the neighbor nodes at the present depth before moving on to nodes at the next depth level.

**Input:**

* G = (V, E)
* Start vertex v in V

**Output:**

* dist[]
  + For all vertices u reachable from the starting vertex v, dist[u] is the shortest path distance from v to u. If no such path exists, infinity otherwise.
* prev[]
  + Vertex preceding u in the shortest path from v to reachable vertex u.

**Graphs that can use BFS:**

* Unweighted graphs
* Undirected/Directed graphs

**Runtime:**

* O(m + n)

**Dijkstra**

Dijkstra's algorithm is used to find the shortest distance from a source vertex to all other vertices. A path can be recovered by backtracking over all of the pre-labels. [Dijkstra's in 3 minutes](https://www.youtube.com/watch?v=_lHSawdgXpI)

**Input:**

* G = (V, E)
* Start vertex v in V

**Output:**

* dist[]
  + Shortest distance between vertex v and reachable vertex u or infinity otherwise if not reachable.

**We have access to:**

* prev[]
  + Vertex preceding u in the shortest path from v to reachable vertex u

**Graphs that can use Dijkstra's:**

* Weighted graphs
* Undirected/Directed graphs
* NO negative weights

**Runtime:**

* O((m + n) log n)
* O(m log n) *if the graph is strongly connected*

**Bellman-Ford**

Bellman-Ford is used to derive the *shortest path from s to all vertices in V*. It does not find a path *between all pairs of* vertices in V. To do this, we would have to run BF |V| times. Negative weights are allowed.

[Bellman-Ford: Theory in 4 minutes](https://www.youtube.com/watch?v=9PHkk0UavIM)

[Bellman-Ford: Example in 5 minutes](https://www.youtube.com/watch?v=obWXjtg0L64)

**Input:**

* G = (V, E)
* Start vertex s

**Output:**

* The shortest path from vertex s to all other vertices.

**We have access to:**

* Detect negative weight cycles. We can compare T[n, \*] to T[n - 1, \*].
  + We can only find negative weight cycles that can be reached from starting vertex s.

**Graphs that can use BF:**

* Weighted graphs
* Undirected/Directed graphs
* CAN HAVE negative weights

**Runtime:**

* O(mn)

**SCC**

The SCC algorithm is used to determine the strongly connected components as well as the meta-graph of connected components in a given directed graph.

**Input:**

* G = (V, E)

**Output:**

* meta-graph (DAG) that contains the connected components
* Reverse Topological sorting of the meta-graph

We have access to:

* ccnum[] - strongly connected components produced from the 2nd DFS run

Graphs that can use SCC:

* directed graphs

Runtime:

* O(m + n)

**Kruskal**

Kruskal's is one of the two algorithms used to find the Minimum Spanning Tree (MST) discussed in class. [Kruskal's Algorithm in 2 minutes](https://www.youtube.com/watch?v=71UQH7Pr9kU)

**Input:**

* Connected, Undirected Graph G = (V, E) with edge weights w\_e

**Output:**

* An MST defined by the edges E

**Graphs that can use Kruskal's:**

* Connected
* Undirected
* Weighted

**Runtime:**

* O(m log n)

Prim's algorithm is the second and final algorithm used to find the MSTs as discussed in class. [Prim's Algorithm in 2 minutes](https://www.youtube.com/watch?v=cplfcGZmX7I)

**Input:**

* Connected, Undirected Graph G = (V, E) with edge weights w\_e

**Output:**

* An MST defined by the prev[] array

**Graphs that can use Prim's:**

* Connected
* Undirected
* Weighted

**Runtime:**

* O(m log n) if graph is connected
* O((m + n) log n) if graph is not connected

**Floyd-Warshall**

FW is primarily used to find the shortest path from ALL nodes to all other nodes where negative weights are allowed. [Floyd-Warshall in 4 minutes](https://www.youtube.com/watch?v=4OQeCuLYj-4)

**Input:**

* G = (V, E)

**Output:**

* The shortest path from all vertices to all other vertices

**We have access to:**

* We can detect negative weight cycles by checking the diagonals (T[n, i, i]).

**Graphs that can use FW:**

* Weighted graphs
* Undirected/Directed graphs
* CAN HAVE negative weights

**Runtime:**

* O(n^3)

**2-SAT**

The 2-SAT problem is to determine whether there exists an assignment to variables of a given Boolean formula in 2-CNF (conjunctive normal form) such that the formula evaluates to true. The algorithm for solving 2-SAT uses graph theory by constructing an implication graph and then checking for the existence of a path that satisfies the conditions.

**Input:**

* A Boolean formula in 2-CNF is represented as a set of clauses where each clause is a disjunction of exactly two literals.

**Output:**

* A Boolean value indicates whether the given 2-CNF formula is satisfiable. If it is satisfiable, the algorithm may also provide a satisfying assignment of variables.

**Graphs that can use 2-SAT:**

* Directed graphs
  + The implication graph is inherently directed since each implication (¬x → y) has a direction.

**Runtime:**

* O(m + n) - *m* is the number of clauses in the 2-CNF formula, *n* is the number of literal or variables.
  + This runtime stems from the linear runtime of SCC finding algorithms and the construction of the implication graph.

**Ford-Fulkerson**

A greedy algorithm to find max flow on networks. The algorithm continually sends flow along paths from the source (starting node) to the sink (end node), provided there is available capacity on all edges involved. This flow continues until no further augmenting paths with available capacity are detected.

[Ford-Fulkerson in 5 minutes](https://www.youtube.com/watch?v=Tl90tNtKvxs)

[Ford-Fulkerson Algorithm Web Animation](https://algorithms.discrete.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html)

**Input:**

* G = (V, E)
* Flow capacity c
* Source node s
* Sink node t

**Output:**

* Max flow

**We have access to:**

* Can trivially create the final residual network with G
* Max flow of G

**Example use:**

We run FF on the flow network to get the maximum flow.

We use this to construct the residual graph.

Graphs that can use FF:

* Directed graphs with capacity of edges

**Runtime:**

* O(C \* m)
  + *C* is the maximum flow in the network
  + *m* is the number of edges

**Edmonds-Karp**

The Edmonds-Karp (EK) algorithm is utilized to determine the maximum flow in a network. This is analogous to the Ford-Fulkerson method but with one distinct difference: the order of search for finding an augmenting path must involve the shortest path with available capacity (BFS for G where all edge weights equal 1).

**Input:**

* G = (V, E)
* Flow capacity c
* Source node s
* Sink node t

**Output:**

* Max flow

**We have access to:**

* Can trivially create the final residual network with G
* Max flow of G

**Example use:**

We run EK on the flow network to get the maximum flow.

We use this to construct the residual graph.

**Graphs that can use EK:**

* Directed graphs with capacity of edges

**Runtime:**

* O(nm^2)
  + *n* is the number of vertices
  + *m*^2 is the number of edges