ALGORITHM Descriptions / Runtimes:

APSP: shortest paths between all pairs of nodes in a graph. Basically SSSP with the source node changing to be every node in G.V. Uses adjacency matrices, and "extend-shortest-paths" (see defs below) to calculate L^(|v|-1), which is shortest paths between all matrices. Each outer iteration of the loops (there are 3 loops) finds a single new shortest path between two vertices (i.e. one value in answer matrix changes).

- **Slow-All-Pairs-Shortest-Paths:** for m=2 to |v|-1, uses L[m]= "extend-shortest paths" (see defs). The final L[v-1] will contain all-pairs shortest paths. O(V^4). see algorithm on back. fine with negative edges, but no negative cycles.
- **Faster-All-Pairs-Shortest-Paths:** uses fact that L^2 = L*L, L^4 = L^2 * L^2 ("repeated squaring"). See algorithm on back. O(V^3 * logV).
- **Floyd-Warshall:** Cuts down runtime complexity to O(N^3). See algorithm for details (still uses essentially the same idea as slow-all-pairs-shortest-paths).
- **Transitive-Closure:** Assign weight of 1 to each edge and run Floyd-Warshall algorithm (O(N^3)).
- **Johnson's Algorithm:** Best for sparse graphs (with less edges). Works for negative edges, but not negative cycles. Uses reweighting in order to get rid of negative edges, so as to utilize Dijikstra's. We define w'(u,v) = w(u,v) + h(u) - h(v). H(x) is found in preprocessing in O(VE) time by creating "Super Source" node (see def) s0, and h(u) = shortest path from s0 to u (i.e. either 0 or negative) using Bellman-Ford. See algorithm on back.

Max Flow: For multiple source / sinks, add global source / sink nodes. All edges must have non-negative capacity. No self-loops. To be a flow network, must satisfy flow conservation and capacity constraint (see defs below). If edge (u, v) and edge (v, u) exist in graph G, we must add node v' and edges (u,v') = c(u,v) and (v', v) = c(u,v), and the same for edge (v,u). This is because if edge (u,v) exists, for a flow network edge (v,u) must not exist. |f| = f_out(A) - f_in(A) where A is a flow network (or cut of one). **Using Orlin's algorithm, can be O(V*E).** Edmunds Carp = O(V * E^2)

- **Ford-Fulkerson:** uses max-flow min-cut thm, residual networks, and augmenting paths. See alg. on back, O(E * |F*|), where |F*| = max flow of a network (because ford-fulkerson increases flow by 1 unit at a time, see alg.)
- **Max Bipartite-Matching:** requires a bipartite graph, finds maximum number of matchings between nodes on left and nodes on right. Uses super-sink and super-source nodes. E.g. left = machines, right = tasks, edge exists if machine can do task. Then max matching will provide work for maximum number of machines. Set all edges (incl. super source / sink) capacity to 1, then do Ford-Fulkerson. O(VE), since |F*| = E as each edge has unit weight. No alg.

BFS: finds all nodes reachable from source. Creates tree of all reachable nodes. Finds shortest paths from s to v by number of edges. Works on directed / undirected graphs. Cycles ok.

DFS: finds all nodes reachable from source. Creates forest of trees. same pre/post conditions as BFS

- **Strongly Connected Components Decomposition:** All nodes in component are reachable by all other nodes in component. Run DFS w start / finish times from arbitrary start node. Then, find G' = G transpose. Then, run DFS on each node in G' starting at last finish time to first. All nodes reachable from this node during this DFS form a SCC. O(V + E).
- **Topological Sort:** Use DFS with start / finish times, but add node to front of list when setting finish time. List at end is topological sort. Cannot do on graphs with cycles. O(V + E).
- **Transitive Closure:** which nodes are reachable from which other nodes? Construct matrix, run DFS from each node. O(N) * O(N + E) = O(N^2 + NE).

Dynamic Programming: steps: 1) characterize optimal substructure (base cases) 2) recursively define value of optimal solution and 3) compute the value of optimal solution in bottom-up fashion

Minimum Spanning Tree: can be unique. Finds min weight path to connect all nodes in G from S. Directed or undirected ok. Greedy algorithms. Depending on data representation, replace logV with V.

- **Prims:** Q = {s}. Find min weight edge to connect node not in Q to Q. repeat. (single set at any time). O(E + VlogV)
- **Kruskal's:** MST (like prims). Find smallest weight edge (u,v) and create set Q = (u,v) if u and v don't belong to same set already. Will have multiple sets; at end, connect sets. O(ElogV)

SSSP: for directed graphs. works for quantities we want to minimize (or max) along a path. mods: single destination by doing SSSP from destination on G-transpose, can add global source and global sink nodes. Max weight by negating edges. Undefined for neg. weight CYCLES (everything else ok). Uses v.d, v.pi attributes. Creates predecessor subgraph.

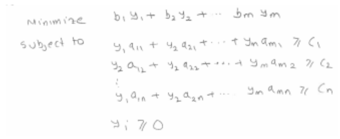
- **Bellman-Ford:** negative edge weights ok - will return false if negative cycle. Relax each edge |V|-1 times. Then, relax all edges once more; if any v.d values change, negative cycle exists. O(VE). Correctness: triangle inequality (see below)
- **DAG-Shortest Paths:** Topologically sort G, then starting from first node v in topological sort: for each node u in G.adj[v], relax edge (v, u, w). Runtime: O(V+E)
- **Dijikstra:** all edge weights must be non-negative. See algorithm on back. O(V^2) to O(VlogV + E) using fib heap.

- **Constraint Graphs:** for xi - xj <= Bk, set up G.V = x nodes and G.E = (xj, xi) with weight Bk for all constraints. Add a super source s0 (see below). Then, if negative cycle exists, NO SOLUTION. Bellman-Ford will give shortest paths = solution, or return false if no solution. For single variables xj <= Bk, make dummy variable xi = 0 such that xj - xi <= Bk.

Linear Programming: standard form = maximize objective function given ax + … <= B0 … for equality, do <= and >= (two separate constraints).

without positive constraint (x >= 0), use x = x_pos - x_neg where x_pos, x_neg >= 0. Then substitute x_pos-x_neg for x.

- Certificate of Optimality: constraints of original LP are satisfied; constraints of dual LP are satisfied, and objective function = upper bound
- Dual LP: given an objective function maximize ax + by subject to constraints, create new LP where we want to minimize upper bound for original LP. Basically, find max values objective function could be given the constraints of the original LP. see picture below
- Mixed-Integer Linear Programming (MILP): convert OR constraints to AND constraints
  by setting one of the OR constraints to <= infinity, then doing the other etc
- Use simplex to solve these LP problems
- Vertex Cover: x(v) = {0, 1} for v exist in G.v; minimize sigma x(v) subject to:
  x(u) + x(v) <= 1 for each edge (u,v); 0<= x(v) <= 1;
  minimal cover is set of v for which x(v) >= 1/2


Minimize $b_1 y_1 + b_2 y_2 + \cdots b_m y_m$

subject to $y_1 a_{11} + y_2 a_{21} + \cdots + y_m a_{m1} \geq c_1$

$y_2 a_{12} + y_2 a_{22} + \cdots + y_m a_{m2} \geq c_2$

$\vdots$

$y_1 a_{1n} + y_2 a_{2n} + \cdots \quad y_m a_{mn} \geq c_n$

$y_i \geq 0$

Tries: search= O(d); insert = O(dr); remove = O(dr), can be optimized to O(d+r) by storing subfield which shows # nodes at each branch;

space for uncompr = O(r*N), compress = O(nr + N) (N= num chars in dict). r = # of letters / digits in ea string.

Finite Automata: like circuits. O(m^3(N) + n), where N = alphabet, m = pattern string to match, n = text to match

KMP: O(m + n); preprocessing can be done to make it O(n) where n = string to match against

<u>DEFINITIONS / THEOREMS:</u>

Augmenting Paths: If a simple path from s to target exists in residual graph, then it is a flow that should be added to max flow

Capacity Constraint: 0 <= f(u,v) <= c(u,v) for all (u,v) exists in G.E

Constraint Graphs: 2-SAT (2 variables expressions in form xi - xj = Bk with as many x's as necessary). Uses SSSP

Critical Path: longest path in a Dag, use SSSP/DAG-SP to find after negating edge weights in G.

Edges: tree if v is found from (u,v); back if (u,v) connects u to ancestor v (cycle), forward if non-tree edge and (u,v) connects u to descendant v, and cross is all other edges. Forward and directed only work with directed graphs.

Extend-Shortest-Paths: for i, for j, for k, l'[i][j] = min(l'[i,j], l[i][j] + w(k,j)). O(V^3)

Flow Conservation: Sigma (for all v exists in V) of f(v,u) = Sigma (for all v exists in V) of f(u,v) ie what comes in goes out

Max Flow Min-Cut Thm: f = flow in flow network G, then: f is a max flow if residual G has no augmenting paths and |f| = c(S, T) for cuts S, T of G

Multigraph: 2 nodes can be directly connected by more than 1 edge (i.e. 2x (u,v) exists in G.E)

Parenthesis Thm: 3 possibilities: u,v could or could not be descendants of each other depending on DFS start and finish times (not connected, or one completely contained in other).

Predecessor Subgraph: created by SSSD and APSP: like BFS tree, but with shortest paths by weights, not number of edges. Shortest and only path in subgraph is shortest path in regular graph.

Proofs: explanation, induction, proof by contradiction

Repeated Squaring: 2^8 = (2^4)^2, 2^4 = (2^2)^2, 2^2 = (2^1)^2, etc. can use binary representation to calculate 2^k in log(k) calculations for powers that are not a power of 2.

Residual Network: for a flow network, if f(u,v) != c(u,v), then add edge (v,u) = f(u,v) and edge (u,v) = c(u,v)-f(u,v) to residual graph (i.e. now the edge (u,v) is the remaining capacity of the original edge, and the edge (v,u) is now f(u,v) in the original G).

Super Source/Sink: "dummy" nodes with paths to all other nodes of weight 0, used for SSSP

Tree: Connected, acyclic, undirected graph with |V-1| edges (if it has V-1 edges, connected and acyclic mean the same thing)

Transpose: Take G and reverse all its edges; this is G^T

Triangle Inequality Thm: used by SSSP and APSP: for all edges (u,v), SP(s,v) <= SP(s, u) + w(u,v) <SP = shortest path>

White-Path Thm: If G is any graph, v is a descendant of u if when "u" (yes) is discovered there is a path of white vertices from u to v in DFS.

ALGORITHMS (mostly alphabetical):

```
DIJKSTRA$(G, w, s)$
INITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

```
FLOYD-WARSHALL$(W)$
$n = W.rows$
$D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

```
FORD-FULKERSON$(G, s, t)$
for each edge $(u, v) \in G.E$
    $(u, v).f = 0$
while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$
    $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
    for each edge $(u, v)$ in $p$
        if $(u, v) \in G.E$
            $(u, v).f = (u, v).f + c_f(p)$
        else $(v, u).f = (v, u).f - c_f(p)$
```

```
JOHNSON$(G, w)$
compute $G'$, where $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
    $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD$(G', w, s)$ == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
        set $h(v)$ to the value of $\delta(s, v)$
            computed by the Bellman-Ford algorithm
    for each edge $(u, v) \in G'.E$
        $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
    let $D = (d_{uv})$ be a new $n \times n$ matrix
    for each vertex $u \in G.V$
        run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
        for each vertex $v \in G.V$
            $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
    return $D$
```

```
MST-KRUSKAL$(G, w)$
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET$(v)$
sort the edges of $G.E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET$(u) \neq$ FIND-SET$(v)$
        $A = A \cup \{(u, v)\}$
        UNION$(u, v)$
return $A$
```

```
TRANSITIVE-CLOSURE$(G)$
$n = |G.V|$
let $T^{(0)} = \left(t_{ij}^{(0)}\right)$ be a new $n \times n$ matrix
for $i = 1$ to $n$
    for $j = 1$ to $n$
        if $i == j$ or $(i, j) \in G.E$
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for $k = 1$ to $n$
    let $T^{(k)} = \left(t_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

```
MST-PRIM$(G, w, r)$
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi =$ NIL
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN$(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

ADDITIONAL:

Substitution: guess complexity of algorithm (recurrence), use induction to check. (i.e. for all n0 > n … T(N) <= d*guess) for some const. d >0

if works for all d > 0, then bound is not tight. d should come out to be an actual constant

Recursion Tree: start with constant work = root, (i.e. c*n^2), then children are next recursive calls, etc. Total levels = log_2(n)-1, then add up work

done at each level (sigma) and factor out constant factor. Find upper bound to sigma, this is answer!

Master Thm: Either O(n^log_b(a)), O(f(n)), or O(n^(log_b(a)) * log_2(n)). NOTE: f(n) (the constant factor) must be polynomially smaller than

n^log_b(a): ie. their ratio must be n^e, for some e > 0 in order to use the master theorem. Otherwise, use substitution or recursion tree.

Tail Recursive: recursive function should not have to calculate ANYTHING after recursive call, just return.

PROBLEMS:

- Max Subarray - Divide and Conquer (nlogn)
- Robot Job Partition - Divide and Conquer (can it be done in "k" time / binary search on k to find min k value) O(#boards*log(sumBoards))
- 2-SAT (x1 and x2) or (x3 and not-x4) etc can be solved if only has 2 var expr. using SCC-D. Check if any SCC has both x and not x; if not, it has a solution. if so, it does not. Graph: for each condition, add both possibilities; for x1 or ~x2, add (x1,x2) and (~x1, ~x2) to graph
  if ea SCC doesn't have x,~x, then it has solution. Finding it: reverse topolog. SCCs.- if no truth value is set for a var., set it to true.
- Graph Coloring (n classes, many students, how to schedule exams?): n = classes, e = students who share classes, ideal graph coloring scheme = min number of exam time slots necessary to prevent conflicts. NP-complete except for 2-coloring, which using BFS = linear. 4 colors suffices to color any planar graph
- Precedence Constrained Scheduling (job 3 must finish before job 5 can start): DAG-SSSP to find longest path (may need global source etc)
- Constraint Graph with Rb = bad section: take union of constraints and do constraint graph procedure for SSSP. Neg Cycle here = GOOD
- Increasing subsequence: Dynamic programming (edge (u,v) if u < v etc)
- Job scheduling (no overlap): Dynamic Programming with edge (u,v) if jobs do not overlap
- Disjoint Paths: Max Flow with capacity of each edge = 1. O(V*E)