Algorithm Charts

Dynamic Programming

| | Recursive Structure/Last thing done | Memory | Order to solve subproblems | Specifics/pseudo-code | Runtime |
|---|---|---|---|---|---|
| Domino Tiling | Current solutions = sum of previous 2 subproblems | linear | Bottom Up | | O(n) |
| Log Cutting | Best way to cut log = best choice for last cut of log that the last cut creates | linear | Bottom Up: need a last cut, look at subproblems before it to find best last cut | Ex: Cut(4) depends on cut(3), cut(2), cut(1)<br>- Value of solution depends on all solutions before it | O(n^2)<br>At each i, find max of i-1 things (n)<br>Do this n times<br>Brute force: 2^n |
| Matrix Chaining | - last thing: final pair of matrices we need to multiply last<br>- take matrices, find which pair = last, find the best way to find those 2 matrices | | Bottom-Up Preferred (fill from diagonal up)<br><br>Top-Down (a lot more overhead b/c recursive calls) | For every cell in memory, needed everything to the left and everything below (entire row and entire column) | n^3 algorithm (else brute force requires 4^n time algo)<br><br>Recursion: Best(i,j) = min(best(i,k)+Best(k+1, j) +r_I,r_K+1, c_j |
| Seam Carving | Add last row of image, find last row of image<br>Use 2nd to last row for S-values, find least energy sequence by finding the smallest of the least energy seams ending there | 2D memory | Bottom Up | - Asking for least energy seam until now | O(nm) where n is the number of rows and m is the number of columns |

| | | | | | |
|---|---|---|---|---|---|
| Longest Common Subsequence | Last char of 2 strings: is it part of a substring? | | Bottom Up<br><br>If matches go diagonal if does not match then go left or top | Fill from top to bottom, left to right<br>Cell(i, j) needs cells (i-1, j-1), (i-1, j), (i, j-1)<br><br>LCS= LCS(i-1,j-1)+1<br>(if it matches, go diagonal)<br>max(LCS(i,j-1),LCS(i-1,j<br>)<br>(if it doesn't, left or top) | O(nm) where x = n and y = m |
| Gerrymandering | Put the last precinct into one of the two districts | 4D | Bottom-up (start with no precincts, build up | Gerrymandering is NP-Complete | Theta(n^4*m^2)<br>● Pseudo-polyno mial |

Greedy Algorithms

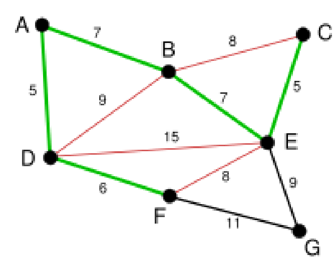
| | Greedy Choice Property | Describe the algorithm | Exchange Argument | Runtime |
|---|---|---|---|---|
| Changemaking | Highest value of coin first | Picked largest coin that is less than target value (only worked with US coins) | | O (k*n)<br>Input size:<br>O( k log n)<br>- pseudo-polynomial time |
| Interval Scheduling | Earliest end time | 1) Find event ending earliest, add to solution<br>2) Remove it and all conflicting events | Take hypothetical optimal, ask what greedy does<br>1) Same selection: we're good | Runtime: linear if sorted, else nlogn to sort |

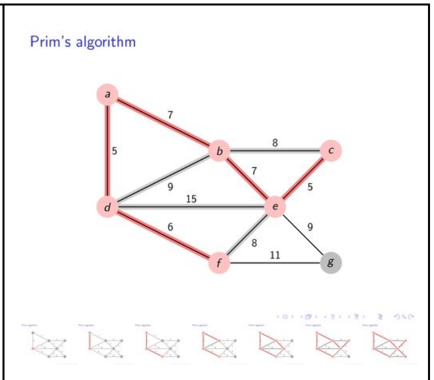
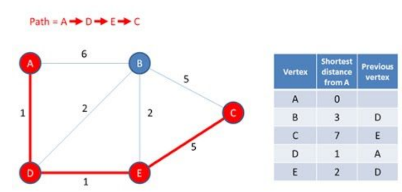
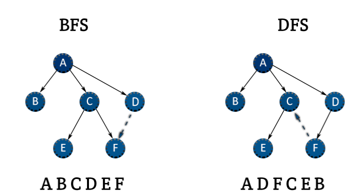
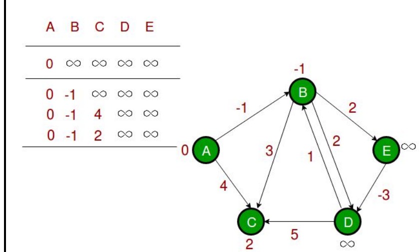
| | | | | |
|---|---|---|---|---|
| | | 3) Repeat until all events = removed, return solution | 2) Different selection:<br>- Let a be the 1st interval to end in OPTi, j and a* = first interval in [i, j] to final overall<br>- by def, a* ends before a and doesn't conflict with any other events<br>--> know this bc a has first end time, and a != a, so a ends after a* bc no intervals overlap with al<br>- same number of intervals, have size OPTi,j +1 -1<br><br>Remove a from opt, include a*, no intervals overlap but it's still the same size | |
| Huffman Coding | Least frequent pair and combine them into a subtree<br>- subproblem size = n-1 | 1) Choose the least frequent pair, combine into a subtree | 1) Show any optimal tree is full (each node has either 0 or 2 children)<br>2) Show optimal substructure (treating c1, c2 as a new combined character gives an optimal solution) | Worst-case: n^2 |
| Belady Cache | Evict the item accessed | - When we load | Sff: schedule chosen by | O(kn^2) |

| | | | | |
|---|---|---|---|---|
| Replacement | furthest in the future | something new into cache must eliminate something already there<br>- Want the best cache "schedule" to minimize # of misses<br>- Only works if you know access pattern in advance, but schedule = optimal | farthest future rule<br>Si - same schedule that agrees with Sff for first i memory addresses<br><br>Show si+1 agrees with sff for the first i + 1 memory accesses, and has no more misses than Si | |

Graphs

- **Go over cut theorem**

| | Algorithm/Pseudocode | Runtime | Example |
|---|---|---|---|
| Kruskal | 1. Start with an empty tree A<br>2. Add to A the lowest weight edge that does not create a cycle<br><br>(Forest of trees → connect min-weight edges of nodes that haven't been reached yet)<br>mst | - Keep edges in a disjoint-set (data structure)<br>- O (E log v) |  |

| | | | |
|---|---|---|---|
| Prim’s | 1. Start with an empty tree A<br>2. Pick a cut (S, V-S) which A respects. Add the min-weight edges which crosses (S, V-S)<br>→ Repeat V-1 times<br><br>(Start with one vertex, keep adding min weight edge that crosses the cut)<br>mst | - Keep edges in a heap/priority queue<br>- O (E log v) |  |
| Dijkstra’s | 1. Given some start node S<br>2. Start with an empty tree A<br>3. Add the “nearest” node not yet in A → repeat V-1 times<br>-GREEDY | - O(E log v + v log v) |  |
| Breadth First Search | Input: a node S<br>Behavior: starts with node s, visit all neighbors of s, then all neighbors of neighbors of s<br>* **Shortest number of hops from s to u** | - O(V + E)<br>- if adj list is used, O(V+E)<br>- if adj matrix is used, O(v^2) |  |
| Bellman Ford | Start node = 0 bc no edges<br>Fill in table 1 row at a time where value in each row is the neighbors & weight of edge find shortest path from e to each node traversing 1 edge, look at previous row to find value of neighbor and if | Worst case is v^2 if dense graph bc initialize vxv memory<br><br>Else, O(ve) runtime because looping through each v and checking each edge, memory lookups = constant |  |

| | | | |
|---|---|---|---|
| | previous cell + weight of edge<br>< what's immediately above<br>---<br>1) Look at all neighbors, find the shortest path through i (traverse max i edges)<br>2) Update table with newest shortest path | | |
| All Pairs Shortest Path | Find the quickest way to get from each place to every other place | Can do in O(v^2* e) time by running bellman ford for each choice of start node | |
| Floyd-Warshall | Find all pairs in O(v^3)<br>Uses DP, growing number of intermediate nodes in path<br>Short(i, j, k) = the length of the shortest path from node $i$ to node $j$ using only intermediate nodes 1, ... , $k$ | Find all pairs in O(v^3)<br><br>Recurrence relation:<br>Short(i,j,k)=<br>Short(i, k,  k-1)<br>Short(i, j, k-1) | K can go from 1 - v for i, j of each node<br>Have vxvxv memory for initialization as well<br>Constant time lookups, takes v^3<br>Final: O(v^3) |

| Differences Between Two Algorithms: | Explanation: |
|---|---|
| Dijkstra’s vs. Bellman Ford | Dijkstra's:<br>1. Only works for positive weight,<br>2. O(ElogV),<br>3. Not good for dynamic graphs<br>a. Must recalculate from scratch<br>Bellman:<br>1. Can do negative weights (faster) |

| | |
|---|---|
| | 2. O(EV)<br>3. More efficient for dynamic graphs<br>4. O(E) time to recalculate |
| Prims vs. Kruskal | Prims:<br>1. Connects the next lowest weight edge to a forming MST<br>2. Grows tree from seed (arbitrary node)<br><br>Kruskal:<br>1. Add the lowest weight edge to a list of edges<br>2. Selects edges in scattered fashion and puts them together |
| Prims vs. Dijkstra’s | Prims:<br>1. Adds the least weight edge<br>Dijkstra’s:<br>1. Adds the least weight path, Calculate next node to be added by keeping track of min path<br>2. Don’t pick starting node (it’s given) |
| Dijkstra’s vs. BFS | Dijkstra’s:<br>1. Uses priority queue<br>BFS:<br>1. Uses queue<br><br>**How to convert Dijkstra's into BFS?<br>Replace priority queue with reg queue, do this iteratively |
| BFS vs. DFS | BFS:<br>2. Uses queue<br>DFS:<br>2. Uses stack |
| Ford Fulkerson vs. Edmund Karp | Edmund Karp: chooses augmenting path with fewest edges |

Djikstra vs Floyd Warshall: Floyd Warshall finds shortest path between a vertice and every other vertice

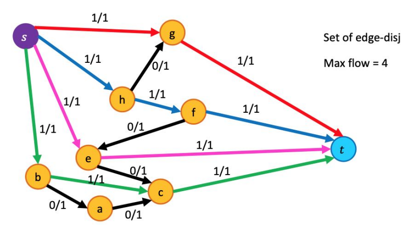
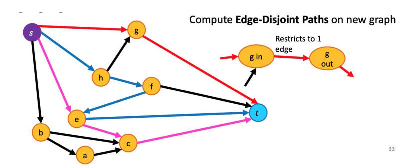
Dijkstra: given a source vertex it finds shortest path from source to all other vertices.

BellmanFord: shortest path from source to every other node (Runtime:V*E)

Network Flow

| | Algorithm | Correctness/Proof | Runtime |
|---|---|---|---|
| Ford-Fulkerson | 1) Take flow graph, build residual<br>2) As long as you find aug path in residual graph, add flow across it<br>3) Look for min weight edge in residual (this is the bottleneck of flow), add that much flow to every edge along that path | Relate to mincut<br>Both = basically the same<br>---<br>cost of cut = sum of capacities of edges that cross the cut from source to sink (ignore sink to source edges)<br>Minimize sum of capacities of edges going from source to sink | **O(e*f)** where e >= v<br>* Takes v+e time to check if sink = reachable from source<br>* Finding aug path = v + e with bfs<br>* finding min weight path = v<br>* How many times to find aug path? Up to size of total max flow we push through graph<br>- F = total amount of flow<br>* Algorithm is **not** polynomial<br>→ pseudo-polynomial |
| Edmunds-Karp | Similar to Ford-Fulkerson but choose augmenting path with fewest hops → use BFS to find augmenting path | - Claim: Max flow in a flow network G is always upper-bounded by the cost any cut that separates s and t<br>- Proof: conservation of flow<br>→ all from from s must eventually get to t<br>→ to get from s to t, all flow must cross the cut somewhere<br>- Conclusion: Max flow in G is <u>at most</u> cost of min cut in G | O(min(E\|f\|, VE^2))<br>---<br>Guaranteed you don't do more than VE times |

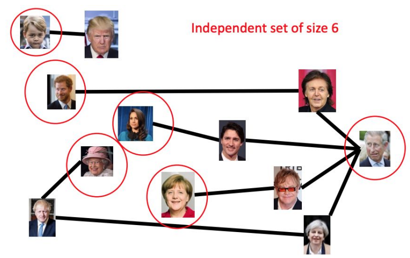
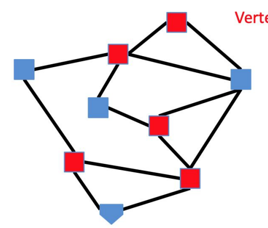
| | | | |
|---|---|---|---|
| | | - Correctness: Ford-Fulkerson terminates when there are no more augmenting paths in the residual graph Gf, which means that f is a maximum flow | |
| Max Flow, Min-Cut | Show there exists a cut through graph whose cut matches flow | 1) If f = maxflow, G has no aug paths, FF stops<br>- else, if some AP in res --> add additional flow through graph using AP --> flow != max<br><br>2) Cut whose cost == maxFlow<br>- no AP --> no math from source to sink traversing only positive weight edges<br><br>All nodes = reachable from source by traversing only pos weight edges in res graph<br>Start at source, wander away from source to all pos edges, every reachable node = on source team<br><br>Cost of cut == total amount of flow through graph:<br>Have a cut whose cost = flow crossing this cut, amount of flow crossing == total amount of flow running | Max flow of network coincides with min cut of graph<br>- Finding either min cut or max flow yields solution to other<br>- Max \|f\| = min \|\|s, t\|\| |

| | | | |
|---|---|---|---|
| | | through network bc that cut separates source and sink<br><br>Maximal flow == minimal cost cut | |
| Edge-Disjoint Path | Given a graph G = (V, E), a start node s and a destination node t, give the max number of paths from s to t which share no edges<br>1. Make s and t the source and the sink, give each edge capacity 1, find the max flow | |  |
| Vertex-Disjoint Path | Give max number of paths from s to t which share no vertices<br>1. Make 2 copies of each node, one connected to incoming edges, the other to outgoing edges | |  |
| Max Bipartite Matching | Find largest number edges at each node touches max 1 choice on either side<br>1) Get source and sink node, draw edges with capacity 1 (inflow == outflow)<br>2) find max flow<br>3) Total amount of flow in graph = sum of edges exiting the sourcenode | 1) Get source and sink node, draw edges with capacity 1 (inflow == outflow)<br>2) find max flow<br>3) Total amount of flow in graph = sum of edges exiting the sourcenode | O(E*v) where v = minimum number of nodes |

Min cost Max Flow

- - A cost is associated with each unit of flow sent along an edge
- - Goal: maximize flow while minimizing cost

Reductions

| | Description? | Example | Reduction |
|---|---|---|---|
| Maximum Independent Set | No 2 nodes = neighbors (find largest independent set for given graph)<br><br>An independent set where no two nodes share an edge |  | S is an independent set of G iff V-S is a vertex cover of G<br><br>Max Ind Set → reduces to Min Vertex Cover<br><br>Solution for max ind set by taking complement of Y solution ← Y solution for min vert cover |
| Minimum Vertex Cover | Find the min vertex cover C (find the least number of nodes where each edge is reached from at least one endpt)<br><br>Where every edge in a graph is “covered” by a given subset of nodes |  | * Same way as above but flipped<br><br>Loop thru edges check to make sure 1 endpoint we found is in list of nodes |

| NP Hard | 3 SAT | K Independence Set | K Vertex Cover | K Clique |
|---|---|---|---|---|
| - At least as hard as NP | Given a 3-CNF formula (logical AND of clauses, each an OR of 3 variables), is there an assignment of T/F to each variable to make the formula true?<br><br>Boolean expression that must be a conjunction (AND) of disjunctions (OR) | Given a graph G = (V, E) and a number k, **determine whether there is an independent set S of size k** | Given a graph G = (V, E) and a number k, **determine whether there is a vertex cover C of size k** | Given a graph G = (V, E) and a number k, is there a clique of size k?<br>- Clique: A complete subgraph (every node connected to every other)<br><br>-if C is NP hard, c reduces to B<br>-use harder prob to solve easier prob |
| Proof of NP Complete → NP hard and NP | | | | |

- Lower bound: A->B, A is slow, use A to solve B know that B is slow to
- NP Hard is at least as hard as NP
- NP Complete is NP (verifiable in polynomial time) and NP hard
- P is solvable in polynomial time
- Np is verifiable in Polynomial time

3 Variants of Problems:

| | Approach | Statement |
|---|---|---|
| Decision Problem | Is there a solution? Is there a vertex cover of size k? | True/False |
| Search Problem | Find a solution. Give a vertex cover of size k. | Complex |

| | | |
|---|---|---|
| Verification Problem | Give a potential solution. Is this a vertex cover of size k? | True/False |