EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Graph Terminology

- Directed, Undirected
- Paths, simple paths, cycles, simple cycles
- Connected graphs
- Trees, forests
- Euler paths, existence of Euler cycles
- Degree of a vertex, neighbors of vertices
- Connectivity in directed paths
- Graph representations
  - Adjacency list, forward star, adjacency matrix

Graph Algorithms

- Graph traversals
  - BFS, DFS, how they are used for solving
- Minimum Spanning Trees
  - Prim’s, Kruskal’s, Boruvka’s
- Single source shortest path algorithms
  - Dijkstra’s, Bellman-Ford
- All pairs shortest paths
  - Floyd-Warshall, Johnson’s
- Single source, single destination
  - A* search

Breadth-First Search (uses Queue)

1. Mark all vertices as unvisited, parents as NULL, depth as -1
2. Choose any unvisited vertex, mark it as visited and enqueue it onto queue
3. While the queue is not empty:
   - Dequeue top vertex v from the queue. Do work to be done on that vertex
     - If parent[v] == NULL, set depth to 0; otherwise, set depth to depth[parent[v]] + 1
     - For each vertex adjacent to v (e.g. in out list) that has not been visited: Mark it visited, mark its parent as v, and enqueue it
     - Mark v as done
4. If there are unvisited vertices, choose any unvisited vertex, mark it as visited, enqueue it and repeat step 3

- This can handle graphs that are not connected
  - Marking as visited avoids cycles
  - Complexity: O(#V + #E), reduces to O(#E) if strongly connected
  - Size of queue is O(#V)

Depth-First Search

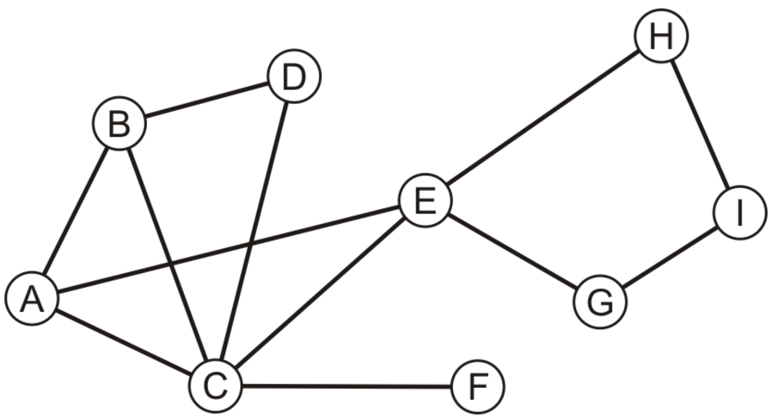
Recursive implementation:

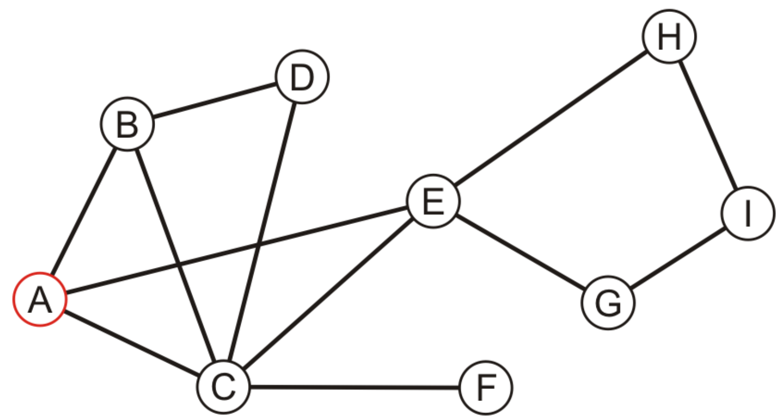
1. Mark all vertices as unvisited; mark all parents as NULL
2. While there are vertices marked as unvisited:
   – Select unvisited vertex v, mark as visited:
   – Do DFS(vertex)

DFS(vertex):
   – For neighbors of vertex
      – If neighbor is unvisited, mark as visited and do DFS(neighbor)

- This can handle graphs that are not strongly connected
  – Marking as visited avoids cycles
  – Complexity: O(#V + #E), reduces to O(#E) if strongly connected
  – Size of queue is O(#V)

Example: BFS





A					

Queue

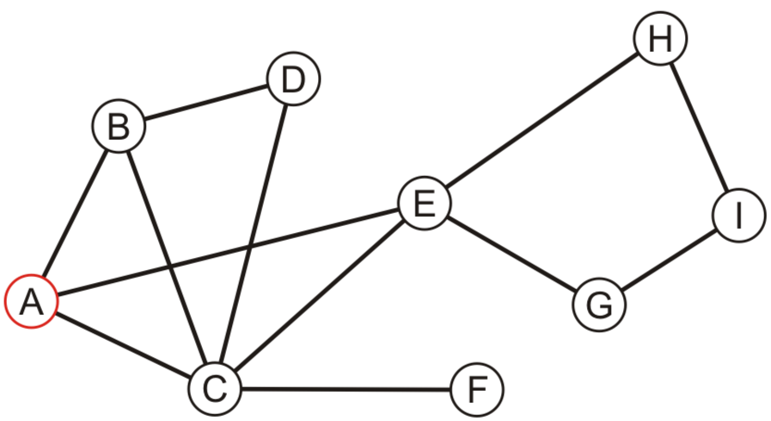
B	C	E			

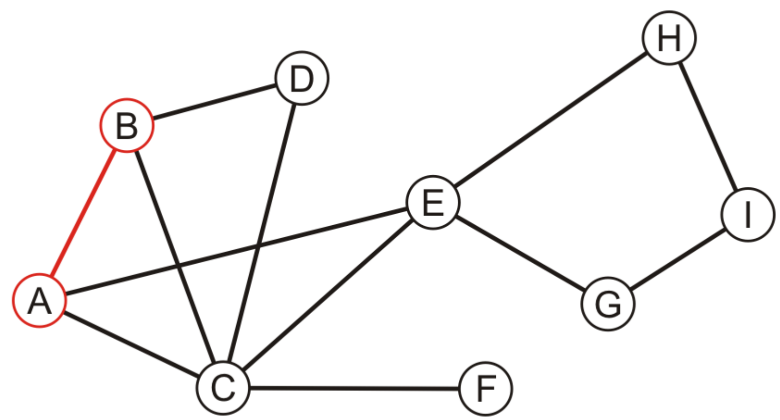
Example

Performing a recursive depth-first traversal:

– Insert A: Visited: A, B
  Stack: A, B

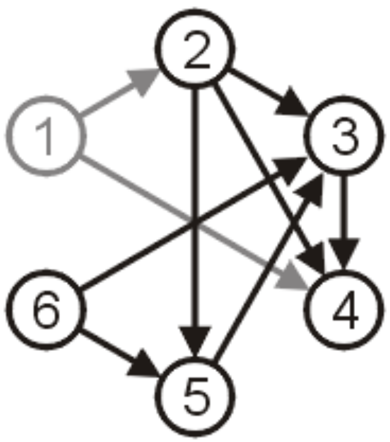
Examine B: Visited A, B, C,
Stack A, B, C





DFS for Topological Sort

- Alternative algorithm: **recursive** DFS (O(#E))
  - Order in which vertices are completed is reverse order of a topological sort!



Stack: 1 2 4 Completed: 4
Stack: 1 2 3 Completed: 3
Stack: 1 2 5 Completed: 5
Stack: 1 2 Completed: 2
Stack: 1 Completed: 1
Stack: 6 Completed: 6
Reverse Order: 6,1,2,5,3,4

DFS for Biconnectivity

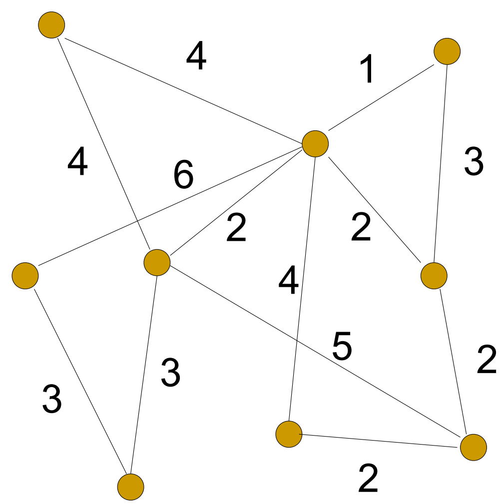
- Connected **undirected** graph is **biconnected** if there are no nodes whose removal disconnects the graph
- Nodes whose removal disconnect the graph are known as **articulation points**
- DFS can be used to find articulation points:
  - Algorithm: Number nodes in Depth First Search order, in the order in which they are inserted into the execution stack of the recursive Depth-First Search. This creates a spanning tree in the graph. Call this number NUM(n) for node n

DFS for Strongly-Connected Components

- Kosaraju's Algorithm
  - Perform DFS on graph G = (V, E),
    - Number vertices according to their finishing time in DFS of G
  - Perform DFS on Gr = (V,Er), where Er are reverse of edges in E, selecting nodes to start in the stack, in decreasing order of finishing time in previous DFS
  - Strongly connected components = reachable trees obtained in last DFS

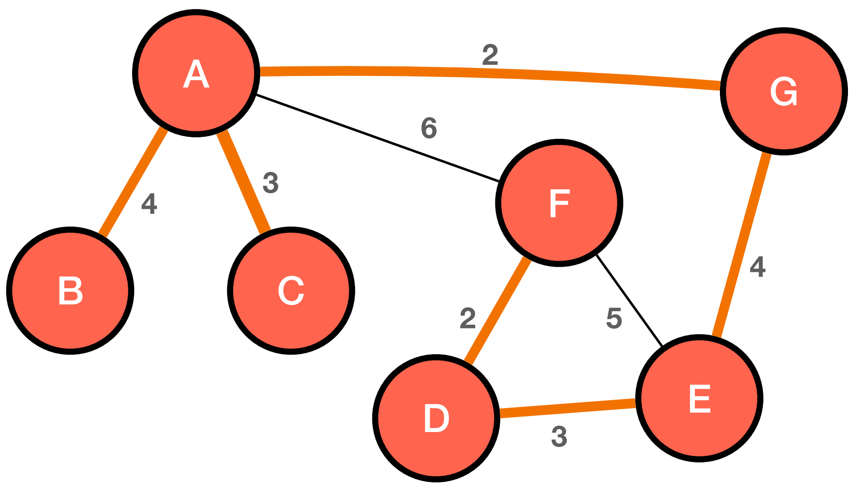
Weighted Graphs

- A weighted graph $G = (V, E)$ is a graph along with a weight function $w : E \to \Re$
- Weighted graphs can be directed or undirected



Spanning Trees

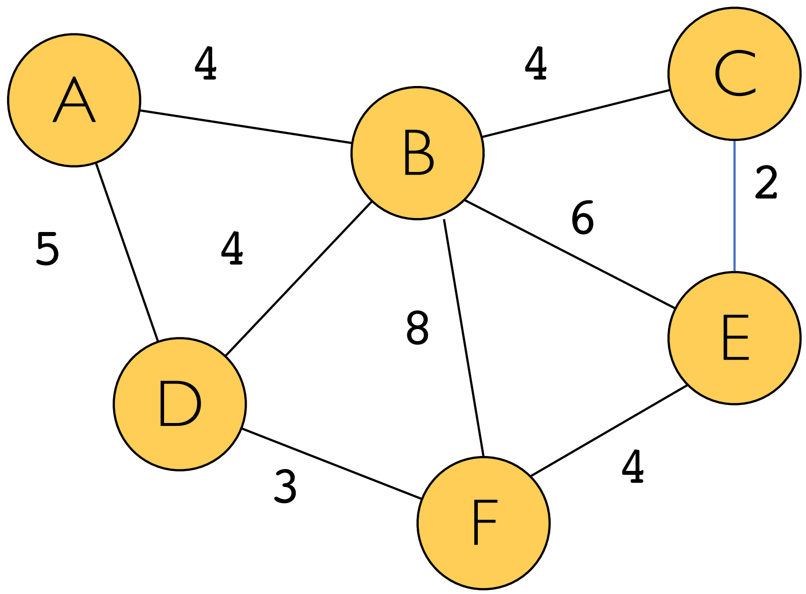
- A **spanning tree** of an undirected graph is
  - edge subset forming a tree that spans every vertex, has #V - 1 edges
- A **minimum spanning tree** (MST) of an undirected weighted graph (V, E) with weights $w(\cdot)$ is a spanning tree with the smallest sum of the weights of its edges



Kruskal's Algorithm

- Sort edges by weight in ascending order
- Start with empty set T (note: it is promising)
- For each edge e in sorted list
  - If adding edge e to T does not create cycle in $(V, T \cup e)$
  - …add it to MST: $T = T \cup \{e\}$
  - Claim: T is now promising set with one more edge
- Stop when you have $\#V - 1$ edges in T

Example




```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

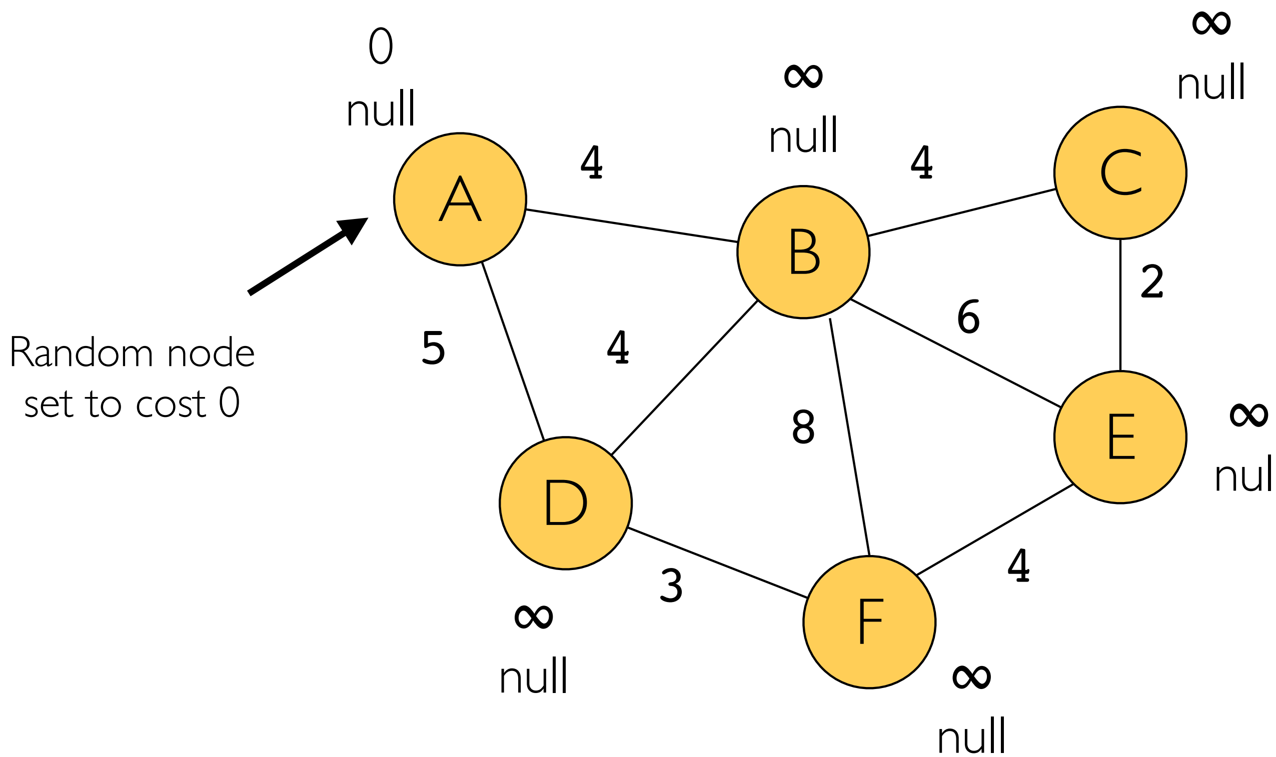
Kruskal Runtime

- $O(|V|)$ for iterating through vertices
- $O(|E|\log|E|)$ for sorting edges
- $O(|E|\times 1)$ for iterating through edges and merging clouds with path compression
- $O(|V|+|E|\log|E|+|E|\times 1)$
  - $= O(|V|+|E|\log|E|)$
- $O(|V|+|E|\log|E|)$
  - Better than simple $O(|V|^3)$ without disjoint sets

Prim-Jarnik

- Traverse G = (V,E) starting at any node
  - Maintain priority queue of nodes (e.g. binary heap, Fibonacci heap)
  - set priority to weight of the cheapest edge that connects them to MST
- Un-added nodes start with priority ∞
- At each step
  - Add the node with lowest cost to MST
  - Update ("relax") neighbors as necessary
- Stop when all nodes added to MST

Example



PQ = [(0,A),(∞,B),(∞,C),(∞,D),(∞,E),(∞,F)]

Runtime

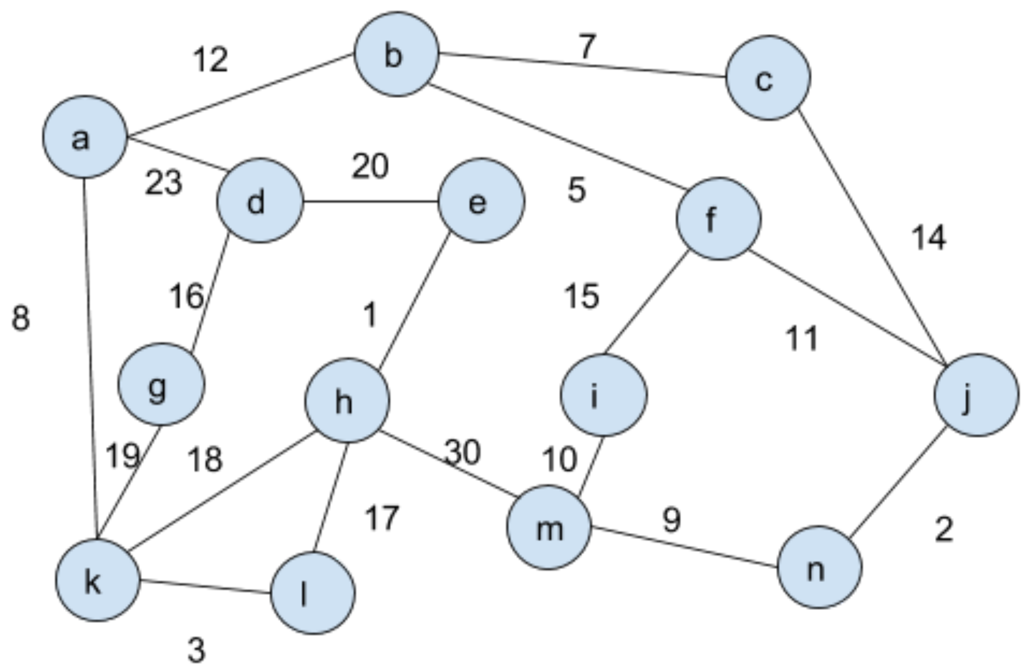
- Initializing nodes with distance and previous pointers is O(|V|); putting nodes in PQ is O(|V|)
- While loop runs |V| times
  - removing vertex from PQ is O(log|V|)
  - So O(|V|log|V|)
- For loop (in while loop) runs |E| times in total
  - Determining whether v' is in PQ: O(1) if we build index into PQ
  - Decreasing vertex's key in the PQ is log|V| (binary heap), or amortized to $O(1)$ if we use Fibonacci or rank-pairing heaps
  - So O(|E|) in complex data or O(|E| log|V|)
- Overall runtime
  - O(|V| + |V|log|V| + |E|) = O(|E| + |V|log|V|)

Borůvka’s Algorithm

- Earliest MST algorithm: 1926. Application: design of power grid
- For **every** connected set in a forest,
  - Select smallest weight edge that leaves connected set
  - Add it to the MST
  - But don’t add it twice if same edge selected by two connected sets
- In principle, merges at least half of the trees at each time: O(log(n)) iterations
    - Easy to parallelize
- Each pass is O(#E)

Example

- Start with every vertex in a separate connected set, partial MST empty
  -



MHT Algorithms are Greedy Algorithms

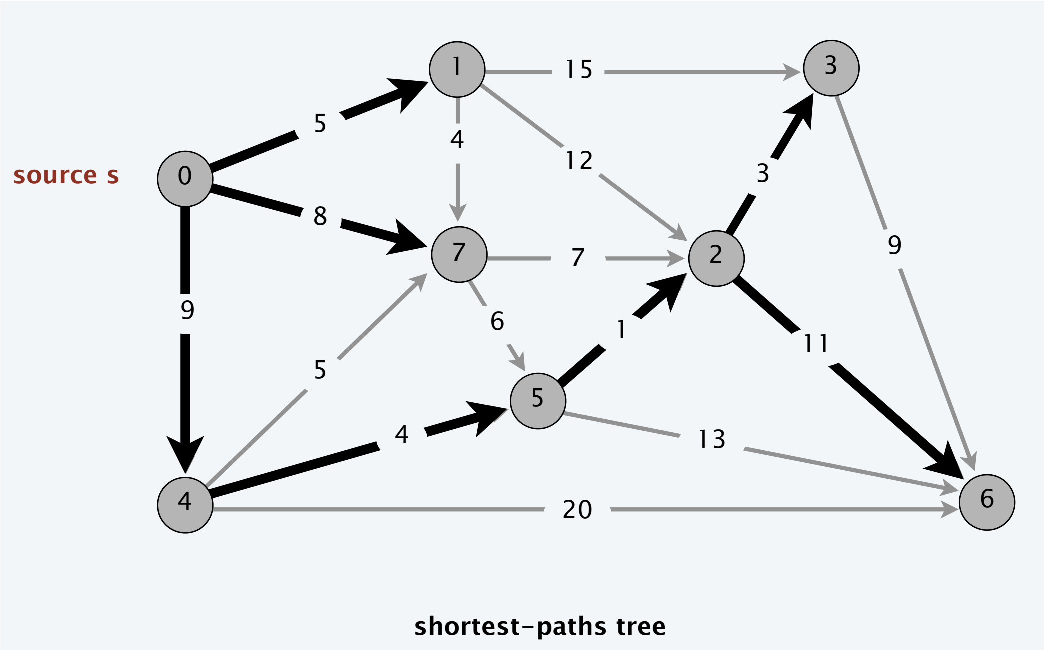
- Greedy algorithm: an algorithm that builds a solution adding an element a a time that is the locally optimal choice at that time
  - Uses a simple rule e.g. MST adds edge with minimum weight across a cut
  - No backtracking (!!!)
  - Greedy algorithms are not always optimal
  - Special classes of problems can be solved to optimality by greedy algorithms

Other Problems with Greedy Algorithm Solutions

- Scheduling
  - Interval scheduling: one processor, jobs with start and end times, maximize number of jobs done: schedule by earliest finish time
  - Scheduling to minimize lateness; jobs with start times, deadlines. Minimize maximum lateness —> Earliest deadline first scheduling
  - Single server, N jobs with different processing times, all available to start right away: to minimize sum of finishing times over jobs —> smallest processing time first
  - Single server, N jobs, unit processing times, hard deadlines d(j), value of scheduling V(j): Try to add to feasible set in order of decreasing value, verify feasible using just-in-time schedule
- Fractional Knapsack: schedule in order of decreasing value/size ratio.

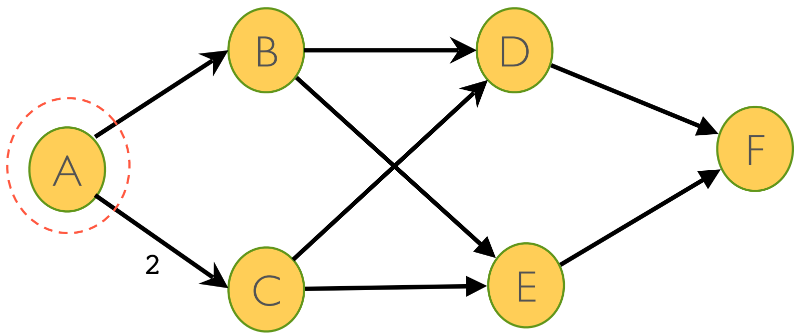
Single Source Shortest Paths (SSSP)

- Given a graph and a source vertex
  - find the shortest paths to all other vertices
  - results in a shortest-path tree
  - Single directed path to every other vertex



Important Property of Shortest Paths

- Lemma: The shortest path from vertex s to a vertex t is composed of shortest paths to and from any intermediate vertices
- Bellman's Principle of Optimality
- Leads to dynamic programming



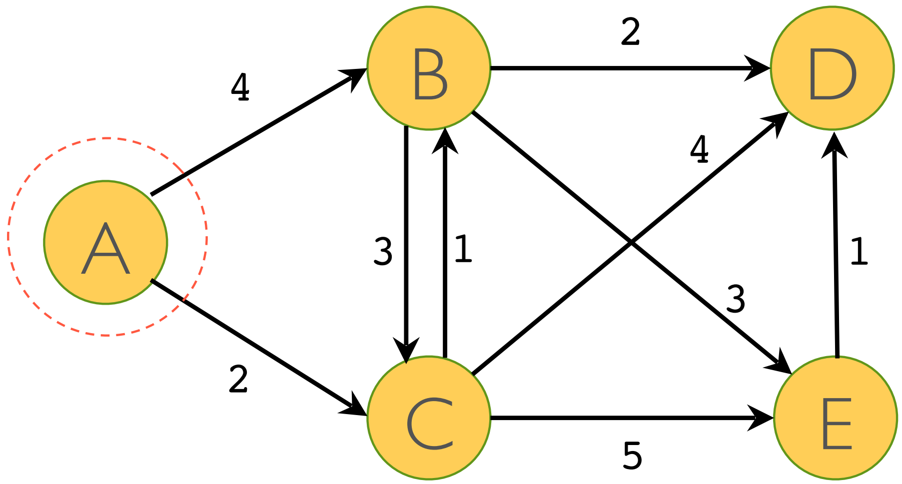
Dijkstra's Algorithm

- **Greedy** Algorithm: Assumes all edges have **nonnegative** weights
  - Maintain a set of **explored** nodes S for which algorithm has determined D[u] = length of a shortest s to u path
  - Initialize S = {s}, D[s] = 0; D[v] = infinity
  - Choose unexplored node $v \notin S$ which minimizes $D[u] + w(u, v), u \in S$

    Set $D[v] = \min_{\{(u,v)\in E : u \in S\}} D[u] + w(u, v)$, and add v to S

    Set pred[v] = vertex u in S that achieves d[v]
  - Repeat until all vertices are explored, so S = V
    - Path to any vertex can be found by using pred[] labels
- Complexity O(|E| + |V| log(|V|)) with Fibonacci or Rank-pairing heaps

Another Example

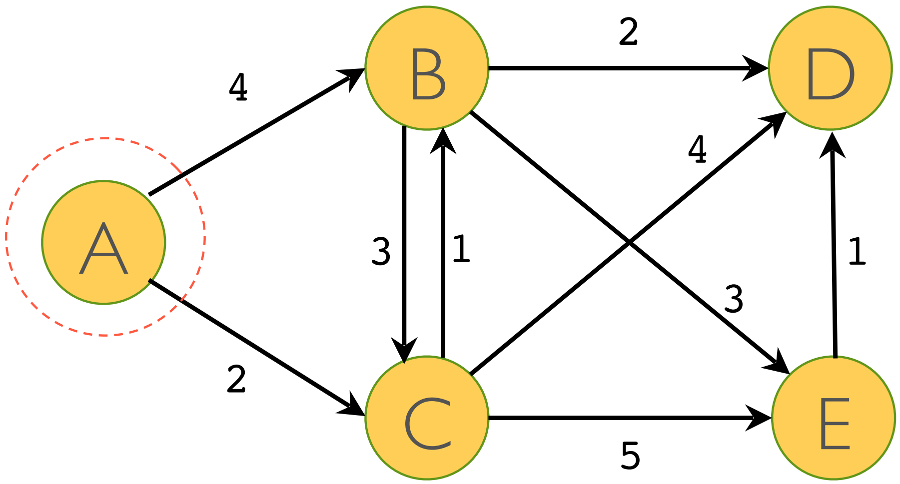


	A	B	C	D	E
`D[]`	0	∞	∞	∞	∞
`pred[]`					

Bellman-Ford Algorithm

- Algorithm converges to shortest paths in graphs with negative distances, provided no negative cycles exist in graph: O(|E||V|) worst case with very simple data structures
- Use a queue of vertices where distances have changed
  - Initialize $D[s] = 0;\ D[v] = \infty, v \neq s;$ pred$[v] = null$
  - Insert s into queue Q; mark inqueue[s] = true, mark inqueue[v] = false, $v \neq s$
  - While Q is not empty:
    - Select u out of queue, mark inqueue[u] = false
    - For each edge (u,v) in E:
      - If $D[v] > D[u] + w(u,v)$:
        - Set $D[v] = D[u] + w(u,v)$, set pred[v] = u
        - if inqueue[v] = false, add v to Q, mark inqueue[v] = true

Another Example



	A	B	C	D	E
D[]	0	∞	∞	∞	∞
pred[]					

All Pairs Shortest Paths

Input: Directed graph G = (V, E), where V = {1, 2, ..., n}, with edge-weight function w : E → R

- Weights may be negative, but no negative cycles

Output: n × n matrix of shortest-path lengths D(i, j) for all i, j ∈ V (and routes)

Floyd-Warshall: initially $D(i,j) = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$

- For k in V: For i in V: For j in V:

  If $D(i,j) > D(i,k) + D(k,j)$:

  set $D(i,j) = D(i,k) + D(k,j)$

- Complexity: $O(|V|^3)$

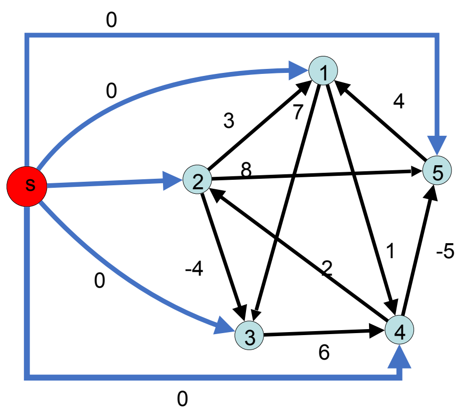
All Pairs Shortest Paths: Alternative

Alternative: Run Dijkstra's algorithm from every possible starting edge

Problem: negative edges

Can fix with Johnson's algorithm: preprocess with Bellman-Ford and rescale edge distances using weights

- **Complexity** O(|V|(|E| + |V| log(|V|)) with fancy heaps



Shortest Path from s to a single t

- A^* search
  - Will search in direction of t for shortest path; a modification of Dijkstra
- Need heuristic function to estimate distance remaining to t
  - admissible: for each vertex v: $h(v) \le D(v,t)$, where $D(v,t)$ is best distance
  - consistent: For each (u,v) in E: $h(u) \le w(u,v) + h(v)$
- Admissible, consistent heuristic leads to optimality

Maximum Flow in Flow Networks

- A feasible st-flow (flow) $f : E \rightarrow \Re^+$ is a function that satisfies

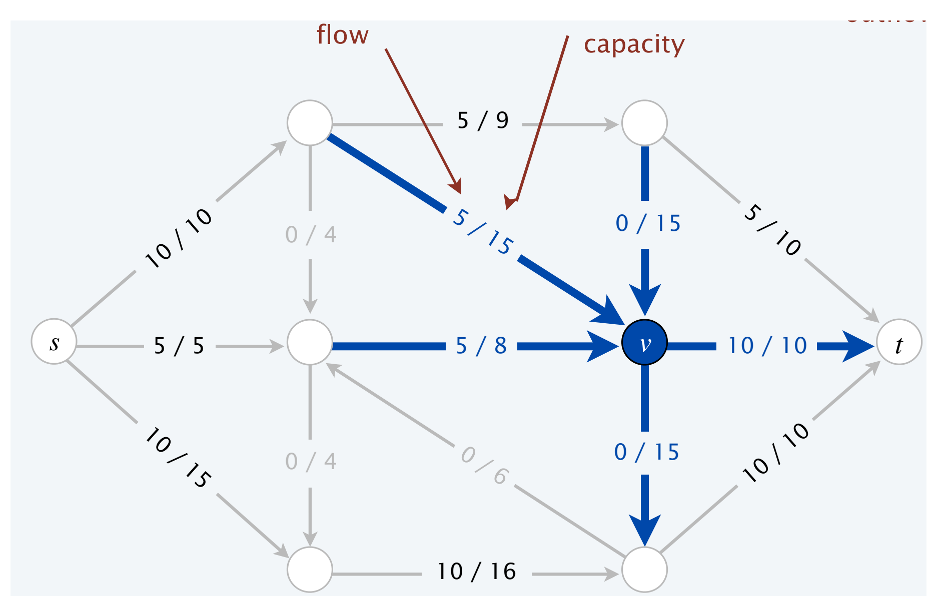
  $0 \le f(u,v) \le c(u,v)$ for all edges $(u\,.\,v) \in E$ (capacity satisfied)

  For every $v \in V - \{s,t\}$, flow is conserved: $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$

- Value of flow: net flow out of s

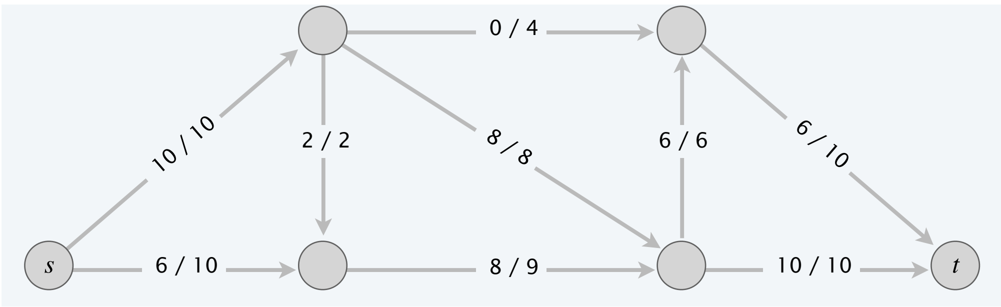
  $$val(f) = \sum_{(s,v)\in E} f(s,v) - \sum_{(v,s)\in E} f(v,s)$$

- Maximum flow problem: find flow f that maximizes $val(f)$
- Value of max-flow = value of min-cut
- Any s-t cut is an upper bound to the max flow problem
-

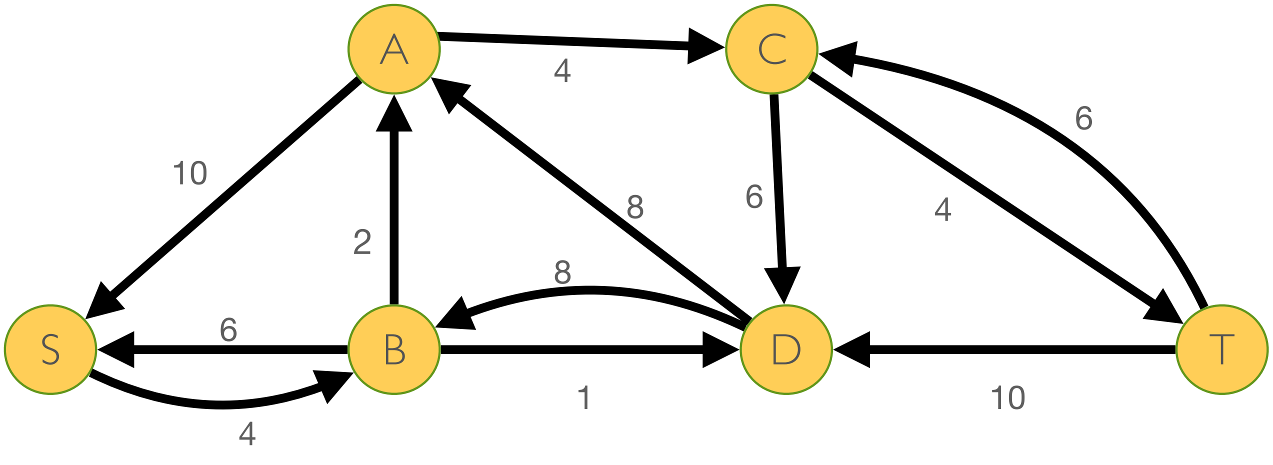


Residual Network

- Current solution



- Residual network has capacity



Ford-Fulkerson Algorithm

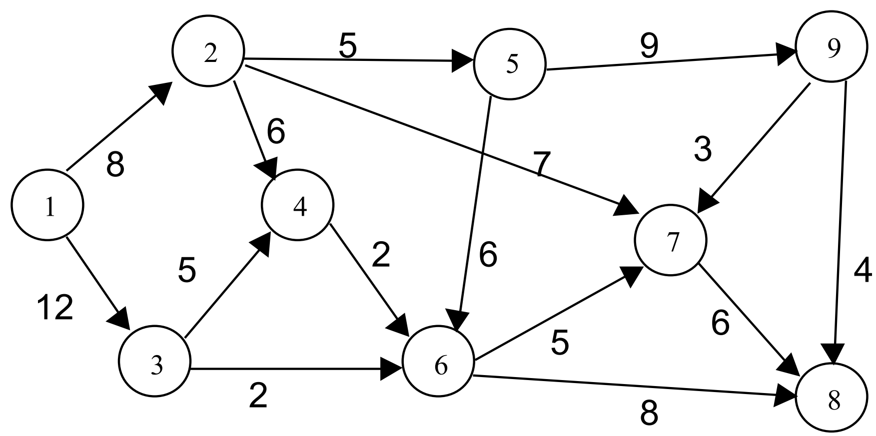
- Start with flow f(u,v) = 0, $(u, v) \in E$. Form the residual network $G_f = G$
- While there exists an s —> t path P in the residual network
  - Compute residual capacity δ on P, and augment flow f using flow δ on path P
  - Update residual network using new flow f, as G_f
- When no path can be found, return flow f

- Complexity O(|V||E| C), where C is largest capacity: pseudopolynomial

- Variation: Edmonds-Karp —> Find minimum hop augmenting paths, guaranteed polynomial complexity $O(|V||E|^2)$

Example

- Numbers are capacities, s = 1, t = 8



- BFS: augment 1—>2—>7—>8, capacity 6

Preflow-Push Algorithms ('88)

- A **preflow** is a function $x : E \rightarrow \Re^{+}$, where $0 \leq x(u,v) \leq c(u,v)$ and $e(v) = \sum_{(u,v)\in E} x(u,v) - \sum_{(v,w)\in E} x(v,w) \geq 0$, for $v \in V - \{s,t\}$
  - e(v) is the excess at vertex v, required to be non-negative
- Let G_x be residual network for a preflow x(). Distance labels d( ) are valid for G_x if d(t) = 0 and d(v) ≤ d(u) + 1 for each $(u,v) \in G_x$
- Let r(u,v) be the capacity of edge (u,v) in residual network G_x. An edge (u, v) is admissible if r(u,v) > 0 and d(u) = d(v) + 1

Goldberg-Tarjan Preflow Push Algorithm

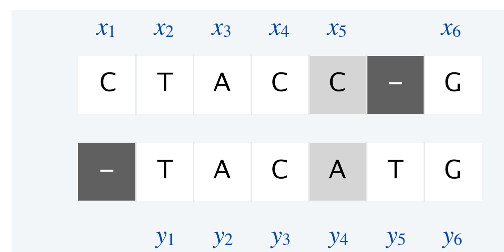
- Initialize:
  - $G_x = G; x(u, v) = 0,\ (u, v) \in E$
  - Using BFS reverse from t, compute distance d(v) for every vertex v
  - For every $(s, v) \in E$, set $x(s, v) = c(s, v)$; set e(v) = c(s,v); set d(s) = |V|
  - Update G_x, with residual capacities $r(u, v),\ (u, v) \in E_x$
- While there is an active node in G_x, select active vertex v and push/relabel(v):
  - If there is admissible edge (v,w): $x(v, w) := x(v, w) + \min(e(v), r(v, w))$
  - Otherwise increase d(v): $d(v) = \min\{d(w) + 1 : (v, w) \in E_r\}$
- Once there are no active nodes, send all excess flow back to s

Dynamic Programming (DP)

- A general approach for breaking solutions of large problems into sequence of solutions of smaller problems
  - Used in shortest path algorithms (BF, FW, A*)
- Weighted interval scheduling, maximum subarray sum, rod cutting
  - Examples of how to use DP for new problems
- Integer Knapsack: Solvable by DP
  - pseudo-polynomial O(nC), C is knapsack size
- Sequence Alignment
  - Solvable by DP, polynomial O(mn)

j\k	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	7	22	23	28	29	29	40
5	0	1	6	7	7	7	22	28	29	34	35	40

$$Cost(M) = \sum_{i,j \text{ matched}} s(x_i, y_j) + \sum_{i:x_i \text{ unmatched}} \delta + \sum_{j:y_j \text{ unmatched}} \delta$$



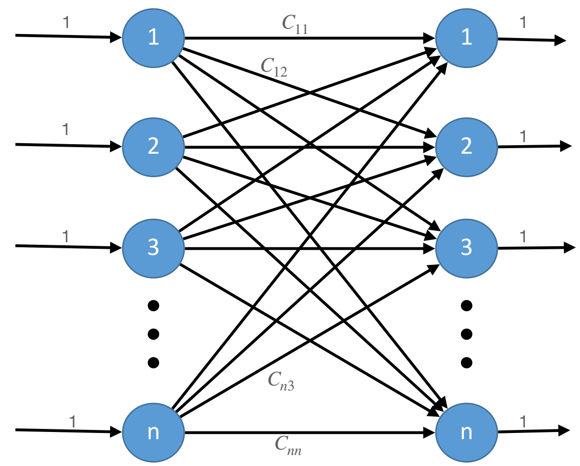
an alignment of CTACCG and TACATG

$M = \{ x_2–y_1, x_3–y_2, x_4–y_3, x_5–y_4, x_6–y_6 \}$

Assignment Problems

\

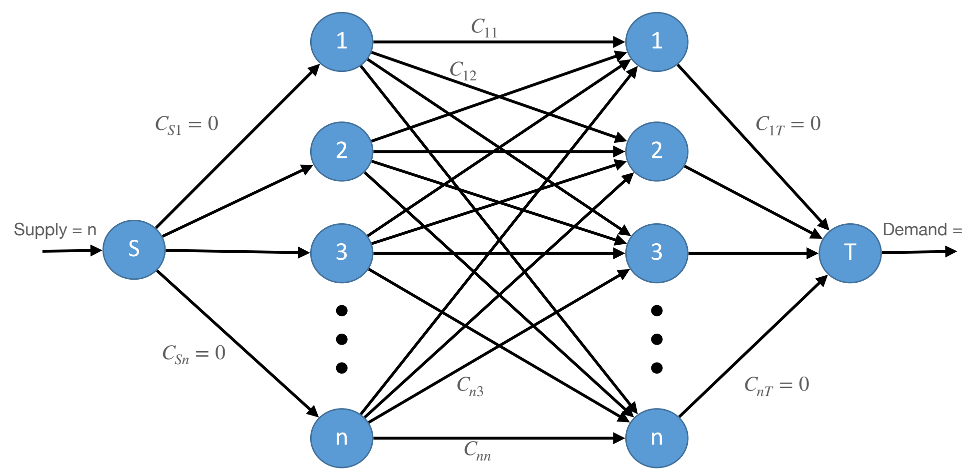
$$\min_{\{x_{ij}\in\{0,1\}\}} \sum_{(i,j)\in E} C_{ij}x_{ij} \text{ subject to constraints}$$



$$\sum_{i:(i,j)\in E} x_{ij} = 1 \;, j \in 1,\ldots,n; \quad \sum_{j:(i,j)\in E} x_{ij} = 1 \;, i \in 1,\ldots,n$$

Graph can be sparse, edges have capacity 1

Equivalent network flow representation



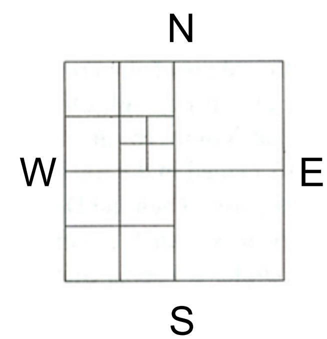
Successive Shortest Path Algorithm

\

- Define prices $q_i, i = 1,\ldots,n$ for persons and prices $p_j, j = 1,\ldots,n$ for objects
- Define reduced costs of edges with those prices as $c^r_{p,q}(i,j') = C_{ij'} + q_i - p_{j'}$
- Initially, set $x_{ij'} = 0,\ (i,j') \in E$, matching $M' = \varnothing$, set prices $p_{j'} = \min\limits_{(i,j')\in E} C_{ij'},\ q_i = 0,\ i, j' \in 1,\ldots,n$
- Construct the residual network (V, E^r) given matching M' and the prices $\{p, q\}$
  - Cost of arcs $(i,j) \in E$: $c^r_{p,q}(i,j)$; cost of reverse arcs (j,i), where $(i,j) \in E$: $-c^r_{p,q}(i,j)$
- Find shortest augmenting path P from s to t; compute the shortest distances d(i), d(j') to vertices i=1, …, n, j' = 1, …,n
- Raise prices $q_i := q_i + d(i),\ i = 1,\ldots,n; \quad p_j := p_{j'} + d(j'),\ j' = 1,\ldots,n$
- Modify assignments on augmenting path P by one unit
- Repeat above iteration n times until complete matching is found
- Complexity $O\big(|V||E| + |V|^2 \log(|V|)\big)$

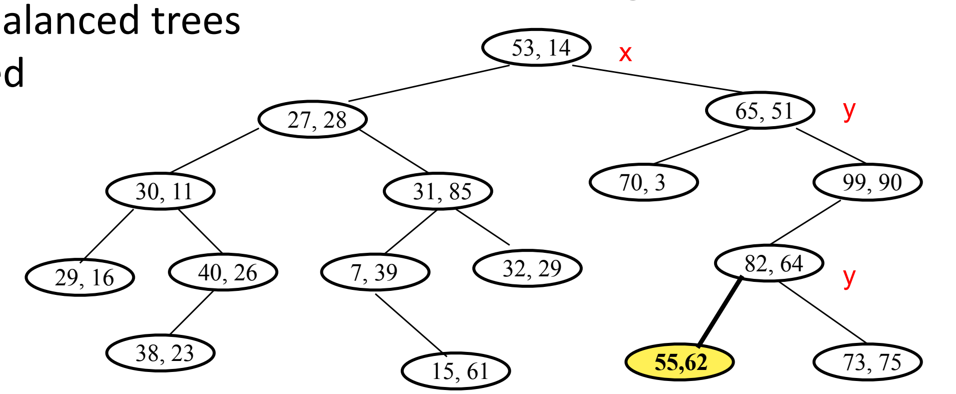
Data Structures for Multidimensional Search

- PR Quadtrees are tries
  - Children of a node: four quadrants of partition of a region
  - If a leaf has more than one point, it splits into 4 subregions
  - Insert, delete, search



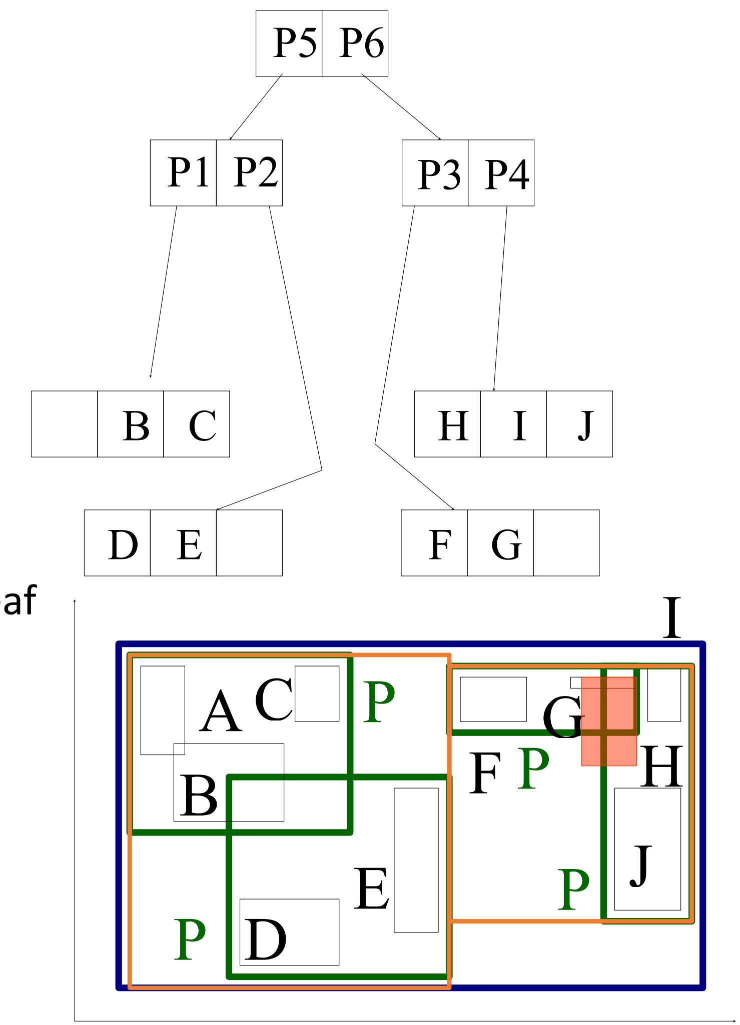
- k-d trees
  - binary search tree where branching decisions are made based on different coordinates at each level
  - Batch construction using medians result in balanced trees
  - One-by-one insertion can be very unbalanced
  - Insertion, deletion, search

Find min x on right



R-trees

- Storage for regions
  - Keys: n-dimensional rectangles, (2 points)
  - All leaf nodes appear on the same level
  - Every node contains between m and M entries
    - $m \leq M/2$ is the minimum entries per node
  - Root node has at least 2 entries (children)
- Insert
  - Insert into rectangle that increases the least by adding
  - Increase measure by perimeter or area — descend to leaf
  - If node saturates (M+1) entries, must split
    - Linear or Quadratic criteria to pick seeds
    - Add to seeds by smallest increase in area
    - Guarantee minimum of m in each split



Computation Complexity

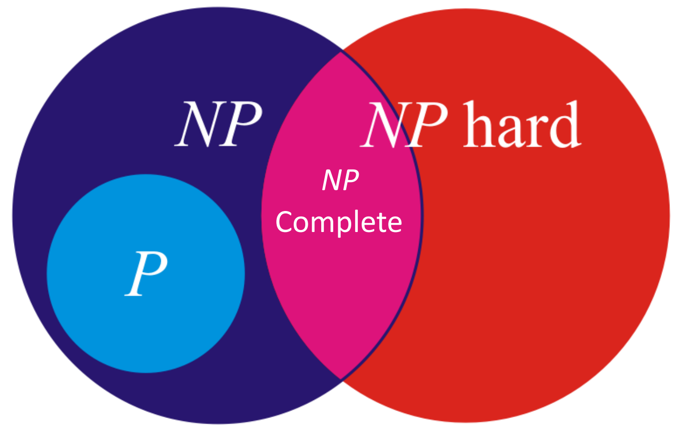
\

- D**ecision problems** have a yes or no solution. Abstract decision problem is a function which maps problem instances I into {yes, no}
- Deterministic, non-deterministic Turing machines …
- Class **P:** there is a algorithm solving the problem with a running time on a deterministic machine that is polynomial in the input size
- Class **NP (**non-deterministic polynomial): given a candidate solution, there is a polynomial-complexity algorithm to verify whether the answer is yes or no for that solution
- Problem A is polynomially reducible to problem B if there exists an algorithm for solving problem A in polynomial time if we could solve arbitrary instances of problem B at unit cost ($A \leq_P B$)
  - If $A \in \mathbf{P}$ and $B \leq_P A$, then $B \in \mathbf{P}$
- Problem A is NP-complete, if $A \in \mathbf{NP}$ and every problem $B \in \mathbf{NP}$ can be polynomially reduced to A. That is, $B \leq_P A$

Other Complexity Concepts

\

- **NP hard problem:** there is an NP-complete problem Y, such that Y is reducible to X in polynomial time (but X may not be in NP)
- **Pseudopolynomial** complexity: If K is the size of the largest number, and n is the size of the input, then the worst case complexity is polynomial in n and K (e.g. Integer knapsack)
- **Strongly Polynomial** complexity: worst case complexity is polynomial in input size, independent of largest value of number in input
- **Strongly NP-complete problems**: If one restricts the size of the largest number in the problem to K, where K is a polynomial in the input size n, then the problem is still NP-complete
  - e.g. Clique



Approximate Algorithms

- Objective: Find approximate algorithms for NP-hard problems with performance guarantees
  - Solution of approximate algorithm is within a factor of optimal solution
- Integer Knapsack: Greedy achieves at least 50% of optimal value
- TSP: MST heuristics can generate tour that is no longer r than twice the distance D* of the optimal TSP tour (can improve to 1.5)
- For pseudopolynomial complexity problems, can usually approximate within epsilon in time that grows as 1/epsilon using rounding