Additional Divide and Conquer Algorithms

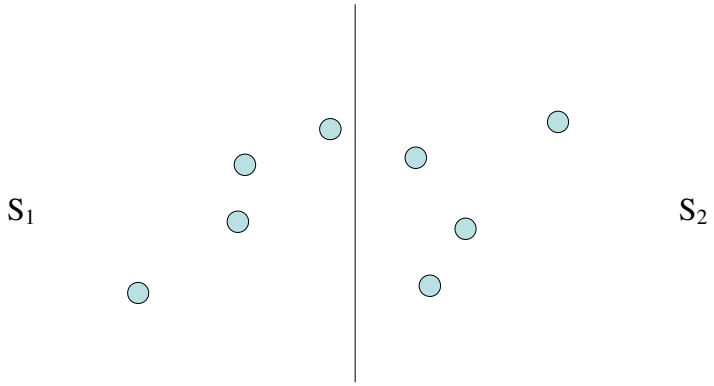
Skipping from chapter 4:

Quicksort
Binary Search
Binary Tree Traversal
Matrix Multiplication

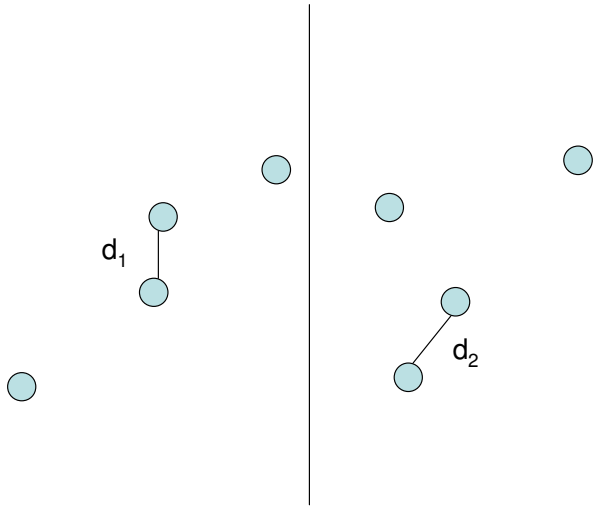
Divide and Conquer Closest Pair

Let's revisit the closest pair problem. Last time we looked at a brute force solution that ran in $O(n^2)$ time. Let $P_1 = (x_1, y_1), \ldots P_n = (x_n, y_n)$ be a set S of n points in the plane, where n, for simplicity is a power of two. We can sort these points in ascending order of their x coordinate using a O(nlgn) sorting algorithm such as mergesort.

Next we divide the points into two subsets S_1 and S_2 where each have n/2 points by drawing a line through the median of all points. We can find the median in O(n) time. This is shown below:

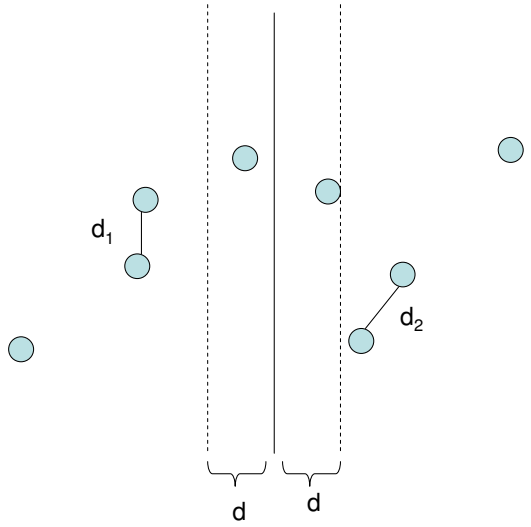


Next, we recursively find the closest pairs for the left subset S_1 and for the right subset S_2. If the set consists of just two points, then there is only one solution (that connects those two points). Let d_1 and d_2 be the smallest distances between pairs of points in S_1 and S_2 as shown below:

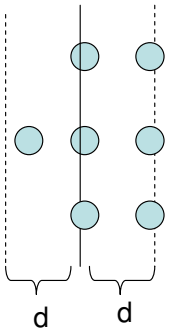


Let d = the minimum of (d_1 and d_2). This is not necessarily our answer, because the shortest distance between points might connect a pair of points on opposite sides of the line. Consequently, we must compare d to the pairs that cross the line.

We can limit our attention to points in the symmetrical vertical strip of width 2d since the distance between any other pair of points is greater than d:



For every point between the left dashed line and the symmetrical line in the middle we must inspect the distance to points on the right side of the line to the right dashed line. However, we only need to look at those points that are within a distance of d from the current point. The key here is that there can be no more than six such points because any pair of points in the right half is at least d apart from each other. The worst case is shown below:



If we maintain an additional list of the points sorted by y coordinate then we can limit the points we examine to +/-d in both the x and y direction for these border points.

To perform the step of comparing distances between points on both sides of the line requires O(n) runtime. For each of up to n points we have a constant number (up to 6) of other points to examine. This process is similar to the "merge" time required for MergeSort. We then compare the smallest of these distances to d, and choose the smallest distance as the solution to the problem.

Our total runtime is then:

$T(n) = 2T(n/2) + O(n)$ ' Time to split in half plus line comparison

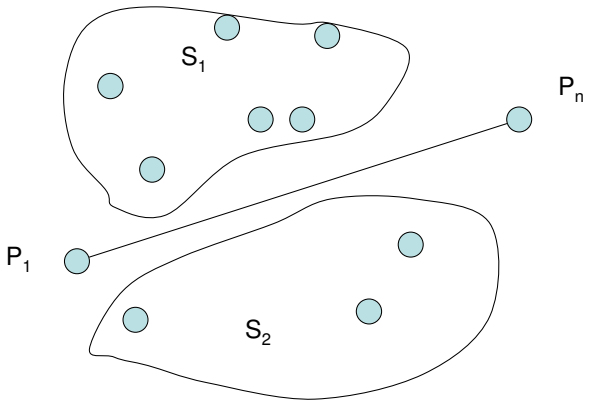
We can solve this using a variety of methods as O(nlgn). This is the best we can do as an efficiency class, since it has been proven that this problem is Ω(nlgn).

Divide and Conquer Convex Hull Problem

Earlier we saw a brute force $O(n^3)$ algorithm to find the convex hull. Let's look at an expected O(nlgn) algorithm called QuickHull that is somewhat similar to Quicksort.

Once again, we have a set of points P located in a 2D plane. First, sort the points in increasing order of their x coordinate. This can be done in O(nlgn) time.

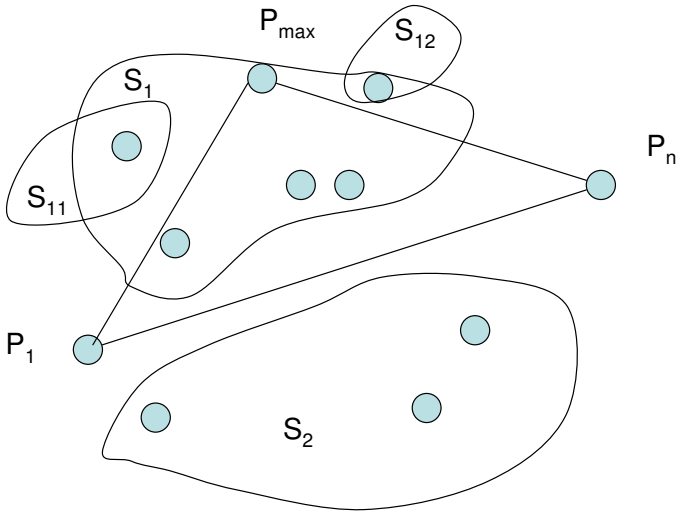
It should be obvious that the leftmost point P_1 and the rightmost point P_n must belong to the set's convex hull. Let $\mathbf{P_1P_n}$ be the line drawn from P_1 to P_n. This line separates P into two sets, S_1 to the left of the line, and S_2 to the right of the line. (left is counter clockwise when connecting the points). S_1 constitutes the boundary of the upper convex hull and S_2 the boundary of the lower convex hull:



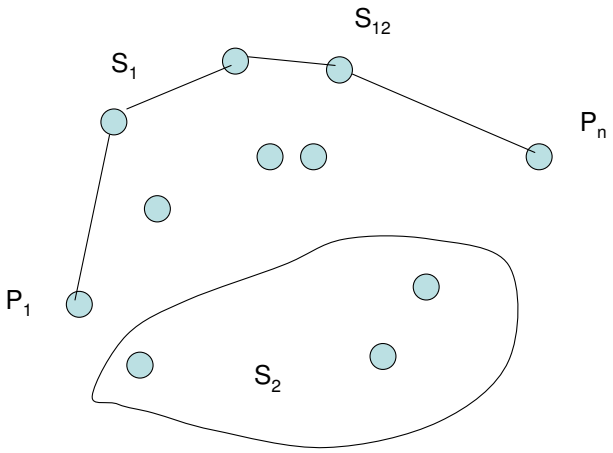
If we're lucky, the line exactly separates the points in half, so half are in S_1 and half are in S_2. In the worst case, all other points are in S_1 or S_2. If there is some randomness the on general we can expect to cut the problem close to in half as we repeat the process.

Next we'll find the convex hull for S_1. We can repeat the same exact process to solve S_2. To find the convex hull for S_1, we find the point in S_1 that is farthest from the line segment $\mathbf{P_1P_n}$. Let's say this point is P_{max}. If we examine the line segment from P_1 to P_{max} then we can recursively repeat the algorithm for all points to the left of this line (The set S_{11} in the diagram below). Similarly, if we examine the line segment from P_n to P_{max} then we can recursively repeat the algorithm for all points to the right of this line (The set S_{12} in the diagram below).

All points inside the triangle can be discarded, since they are in the triangle and can't be part of the convex hull.



If we try to find the points in the convex hull for a set with only one point, then that point must be in the set. At this point we have determined the upper convex hull:



If we repeat the process on the lower convex hull we will complete the problem and find all of the points for the entire convex hull.

For an animated applet that illustrates this algorithm, see:
http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html

What is the runtime? Let's say that we always split the set of points in half each time we draw the line segment from P_1 to P_n. This means we split the problem up into two subproblems of size n/2. Finding the most distant point from the line segment takes O(n) time. This leads to our familiar recurrence relation of:

$T(n) = 2T(n/2) + O(n)$ which is O(nlgn).

Decrease and Conquer

The decrease and conquer technique is similar to divide and conquer, except instead of partitioning a problem into multiple subproblems of smaller size, we use some technique to reduce our problem into a single problem that is smaller than the original.

The smaller problem might be:

- Decreased by some constant
- Decreased by some constant factor
- Variable decrease

Decrease and conquer is also referred to as inductive or incremental approach.

Here are some examples of Decrease and Conquer algorithms:

Decrease by one:
- Insertion sort
- Graph search algorithms:
  - DFS
  - BFS
  - Topological sorting
- Algorithms for generating permutations, subsets

Decrease by a constant factor
- Binary search
- Fake-coin problems
- Josephus problem

Variable-size decrease
- Euclid's algorithm : gcd(m,n) = gcd(n, m mod n)
- Selection by partition

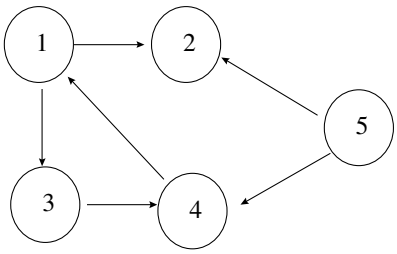
Review of Basic Graph Algorithms

A graph is composed of edges *E* and vertices *V* that link the nodes together. A graph G is often denoted G=(V,E) where V is the set of vertices and E the set of edges.

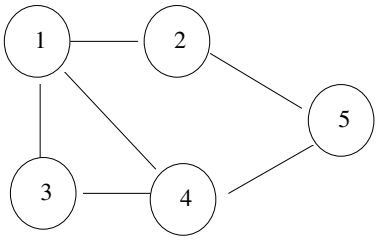
Two types of graphs:

1. Directed graphs: G=(V,E) where E is composed of ordered pairs of vertices; i.e. the edges have direction and point from one vertex to another.
2. Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. the edges are bidirectional.

Directed Graph:



Undirected Graph:



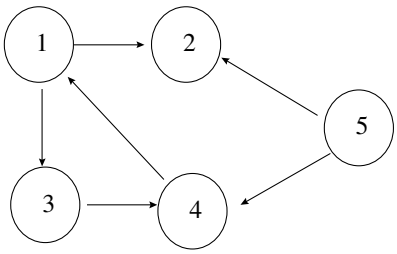
The **degree** of a vertex in an undirected graph is the number of edges that leave/enter the vertex. The degree of a vertex in a directed graph is the same,but we distinguish between in-degree and out-degree. Degree = in-degree + out-degree.

The running time of a graph algorithm expressed in terms of E and V, where E = |E| and V=|V|; e.g. G=O(EV) is |E| * |V|

We can implement a graph in three ways:

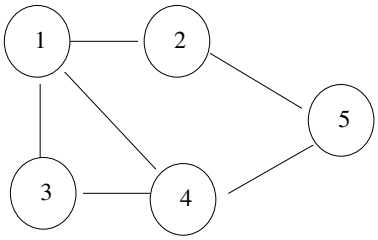
1. Adjacency List
2. Adjacency-Matrix
3. Pointers/memory for each node (actually a form of adjacency list)

Using an Adjacency List: List of pointers for each vertex



A
1 → 2 → 3

2 →
3 → 4
4 → 1
5 → 2 → 4

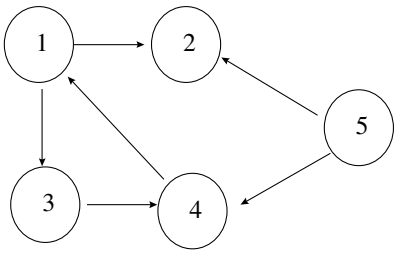


A

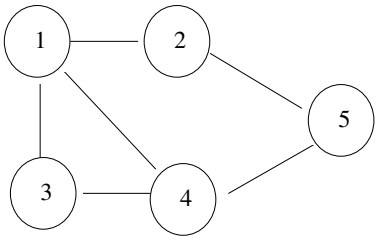
1 → 2 → 3 → 4
2 → 1 → 5
3 → 4 → 1
4 → 1 → 3
5 → 2 → 4

The sum of the lengths of the adjacency lists is 2|E| in an undirected graph, and |E| in a directed graph.
The amount of memory to store the array for the adjacency list is O(max(V,E))=O(V+E).

Using an Adjacency Matrix:



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	0
3	0	0	0	1	0
4	1	0	0	0	0
5	0	1	0	1	0



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

When is using an adjacency matrix a good idea? A bad idea?
The matrix always uses $\Theta(v^2)$ memory. Usually easier to implement and perform lookup than an adjacency list.

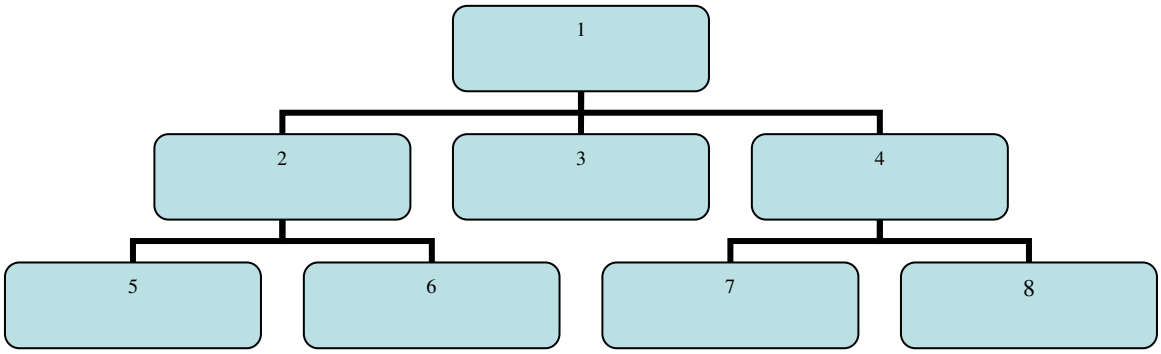
Sparse graph: very few edges.
Dense graph: lots of edges. Up to $O(v^2)$ edges if fully connected.

The adjacency matrix is a good way to represent a *weighted graph*. In a weighted graph, the edges have weights associated with them. Update matrix entry to contain the weight. Weights could indicate distance, cost, etc.

Search: The goal is to methodically explore every vertex and every edge; perhaps to do some processing on each. Will assume adjacency-list representation of the input graph.

Breadth-First-Search (BFS) :

Example 1: Binary Tree. This is a special case of a graph. The order of search is across levels. The root is examined first; then both children of the root; then the children of those nodes, etc.



Sometimes we can stop the algorithm if we are looking for a particular element, but the general BFS algorithm runs through every node.

Example 2: directed graph:

1. Pick a source vertex S to start.
2. Find (or discover) the vertices that are adjacent to S.
3. Pick each child of S in turn and discover their vertices adjacent to that child.
4. Done when all children have been discovered and examined.

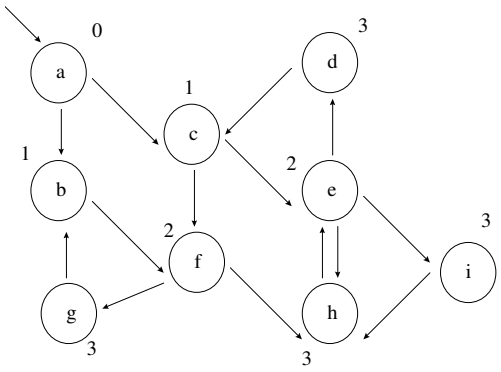
This results in a tree that is rooted at the source vertex S.

The idea is to find the distance from some Source vertex by expanding the "frontier" of what we have visited.

Pseudocode: Uses FIFO Queue Q

```
BFS(s)                              ; s is our source vertex
    for each u∈V - {s}              ; Initialize unvisited vertices to ∞
        do d[u]← ∞
    d[s] ←0                         ; distance to source vertex is 0
    Q←{s}                           ; Queue of vertices to visit
    while Q<>0 do
        remove u from Q
        for each v ∈ Adj[u] do      ; Get adjacent vertices
            if d[v]= ∞
                then d[v] ←d[u]+1   ; Increment depth
                put v onto Q        ; Add to nodes to explore
```

Example: (this is the final state, start with 0 and infinity as values)



Initially, d[a] is set to 0 and the rest to ∞.
Q $\leftarrow$ [a].
Remove head: Q$\leftarrow$[]
children of a are c,b
d[c]= ∞, d[b]= ∞ so d[c] $\leftarrow$d[a]+1=1, d[b] $\leftarrow$d[a]+1=1
Q$\leftarrow$[c b]

Remove head: Q$\leftarrow$[b]
children of c are e,f
d[e]= ∞, d[f]= ∞ so d[e] $\leftarrow$d[c]+1=2, d[f] $\leftarrow$d[c]+1=2
Q$\leftarrow$[b e f]

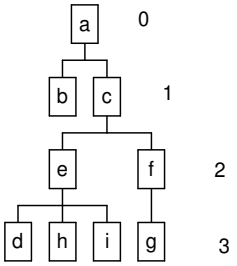
Remove head: Q$\leftarrow$[e f]
children of b is f
d[f] <> ∞, nothing done with it

Remove head: Q$\leftarrow$[f]
children of e is d, i, h
d[d]= ∞, d[i]= ∞, d[h]= ∞ so d[d] $\leftarrow$d[i] $\leftarrow$d[h] $\leftarrow$d[e]+1=3
Q$\leftarrow$[f d i h]

Remove head: Q$\leftarrow$[d i h]
children of d is g
d[g]= ∞, so d[g] $\leftarrow$d[d]+1 = 3
Q$\leftarrow$[d i h g]

Each of these has children that are already has a value less than ∞, so these will not set any further values and we are done with the BFS.

Can create a tree out of the order we visit the nodes:



Memory required: Need to maintain Q, which contains a list of all fringe vertices we need to explore.

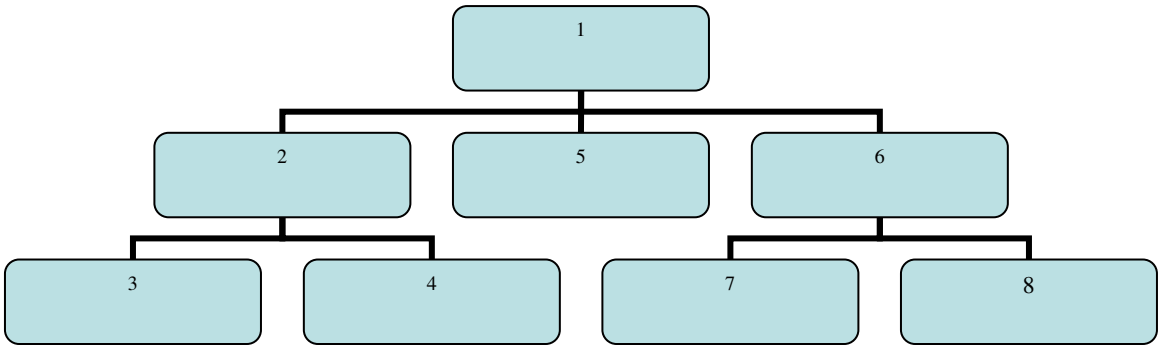
Runtime: O(V+E) ; O(E) to scan through adjacency list and O(V) to visit each vertex. This is considered linear time in the size of G.

Claim: BFS always computes the shortest path distance in d[I] between S and vertex I. We will skip the proof for now.

What if some nodes are unreachable from the source? (reverse c-e,f-h edges). What values do these nodes get?

Depth First Search: Another method to search graphs.

Example 1: DFS on binary tree. Specialized case of more general graph. The order of the search is down paths and from left to right. The root is examined first; then the left child of the root; then the left child of this node, etc. until a leaf is found. At a leaf, backtrack to the lowest right child and repeat.



Example 2: DFS on directed graph.

1. Start at some source vertex S.
2. Find (or explore) the first vertex that is adjacent to S.
3. Repeat with this vertex and explore the first vertex that is adjacent to it.

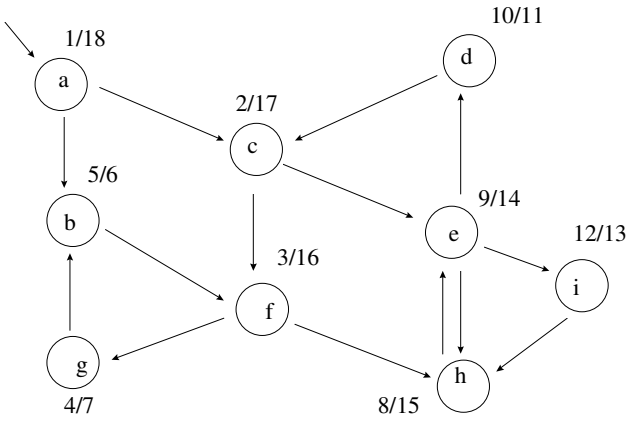
4. When a vertex is found that has no unexplored vertices adjacent to it then backtrack up one level
5. Done when all children have been discovered and examined.

Results in a forest of trees.

Pseudocode:

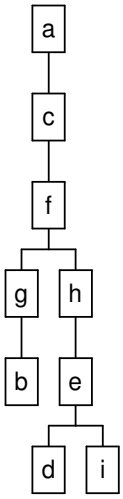
```
DFS(s)
        for each vertex u ∈ V
                do color[u] ← White                    ; not visited
        time ← 1                                       ; time stamp
        for each vertex u ∈ V
                do if color[u]=White
                   then DFS-Visit(u,time)

DFS-Visit(u,time)
        color[u] ← Gray                                ; in progress nodes
        d[u] ← time                                    ; d=discover time
        time ← time+1
        for each v ∈ Adj[u] do
                if color[u]=White
                   then DFS-Visit(v,time)
        color[u] ← Black
        f[u] ← time ← time+1                           ; f=finish time
```

Example:

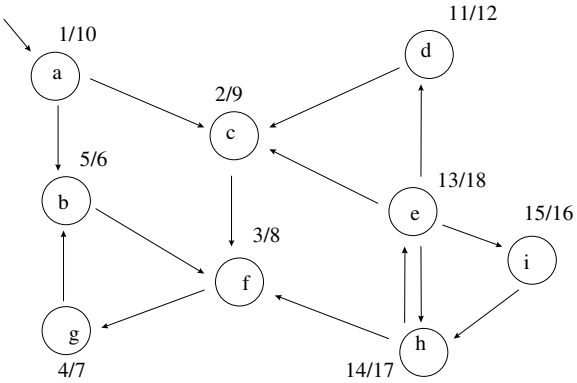


Numbers are Discover/Finish times. We could have different visit times depending on which edges we pick to traverse during the DFS.

The tree built by this search looks like:



What if some nodes are unreachable? We still visit those nodes in DFS. Consider if c-e, f-h links were reversed. Then we end up with two separate trees:



Still visit all vertices and get a forest: a set of unconnected graphs without cycles (a tree is a connected graph without cycles).

Time for DFS:

$O(V^2)$ - DFS loop goes O(V) times once for each vertex (can't be more than once, because a vertex does not stay white), and the loop over Adj runs up to V times.

But…

The for loop in DFS-Visit looks at every element in Adj once. It is charged once per edge for a directed graph, or twice if undirected. A small part of Adj is looked at during each recursive call but over the entire time the for loop is executed only the same number of times as the size of the adjacency list which is Θ(E).

Since the initial loop takes Θ(V) time, the total runtime is Θ(V+E). This is considered linear in terms of the size of the input adjacency-list representation. So if there are lots of edges then E dominates the runtime, otherwise V does.

Note: Don't have to track the backtracking/fringe as in BFS since this is done for us in the recursive calls and the stack. The stack makes the nodes ordered LIFO. The amount of storage needed is linear in terms of the depth of the tree.

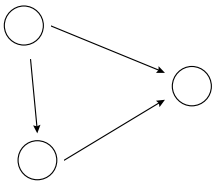
Types of Edges: There are 4 types. DFS can be modified to classify edges as being of the correct type:

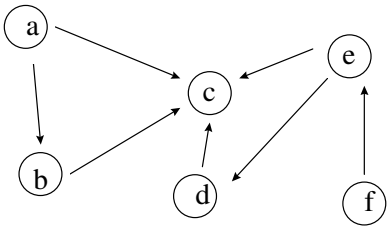
1. Tree Edge: An edge in a depth-first forest. Edge(u,v) is a tree edge if v was first discovered from u.
2. Back Edge: An edge that connects some vertex to an ancestor in a depth-first tree. Self-loops are back edges.
3. Forward Edge: An edge that connects some vertex to a descendant in a depth-first tree.
4. Cross Edge: Any other edge.

DAG's

Nothing to do with sheep! A DAG is a Directed Acyclic Graph. This is a directed graph that contains no cycles.

Examples:



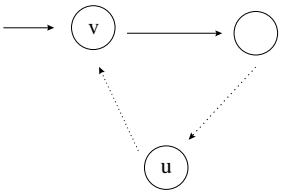


A directed graph D is acyclic iff a DFS of G yields no back edges.

Proof: Trivial. Acyclic means no back edge because a back edge makes a cycle. Suppose we have a back edge (u,v). Then v is an ancestor of u in the depth-first forest. But then there is already a path from v to u and the back edge makes a cycle.

Suppose G has a cycle c. But then DFS of G will have a back edge. Let v be the first vertex in c found by DFS and let u be a vertex in c that is connected back to v. Then

when DFS expands the children of u, the vertex v is found. Since v is an ancestor of u the edge (u,v) is a back edge.



Topological Sort of a dag

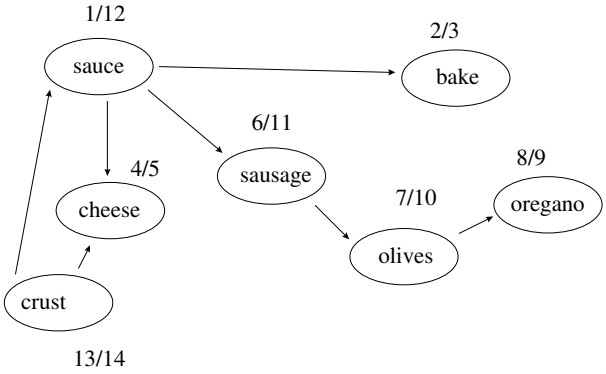
A topological sort of a dag is an ordering of all the vertices of G so that if (u,v) is an edge then u is listed (sorted) before v. This is a different notion of sorting than we are used to. a,b,f,e,d,c and f,a,e,b,d,c are both topological sorts of the above dag. There may be multiple sorts; this is okay since a is not related to f, either vertex can come first.

Main use: Indicate order of events, what should happen first

Algorithm for Topological-Sort:
1. Call DFS(G) to compute f(v), the finish time for each vertex.
2. As each vertex is finished insert it onto the front of the list.
3. Return the list.

Time is Θ (V+E), time for DFS.

Example: Pizza directed graph



DFS: Start with sauce.
The numbers indicate start/finish time. We insert into the list in reverse order of finish time.

Crust, Sauce, Sausage, Olives, Oregano, Cheese, Bake

Why does this work? Because we don't have any back edges in a dag, so we won't return to process a parent until after processing the children. We can order by finish times because a vertex that finishes earlier will be dependent on a vertex that finishes later.