Graphs

We have two main algorithms with which we may deal with graphs: Depth-first search (DFS) and breadth-first search (BFS).

BFS(G, s)

```
1  for each vertex u ∈ G.V − {s}
2        u.color = WHITE
3        u.d = ∞
4        u.π = NIL
5  s.color = GREY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13             if v.color = WHITE
14                   v.color = GREY
15                   v.d = u.d + 1
16                   v.π = u
17                   ENQUEUE(Q, v)
18       u.color = BLACK
```

This algorithm runs in $O(|V| + |E|)$. Additionally for an unweighted graph it correctly computes the distance from s to any other vertex in the graph. The correctness of BFS is shown by first inducting to demonstrate that each $v.d$ has $\delta(s, v)$ as a lower bound, and then showing that the distances increase the later a vertex is enqueued, while the most recent vertex in Q will never have more than a step's worth of distance away from s than the oldest vertex.

DFS(G)

```
1  for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6        if u.color = WHITE
7              DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time                    // Start time of u
3  u.color = GRAY
4  for each vertex v ∈ G.Adj[u]
5        if v.color = WHITE
6              v.π = u
7              DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time                    // End time of u
```

One of the most valuable theorems regarding DFS is the parenthesis theorem, which demonstrates that either one will have a path between u and v, in which case one encapsulates the other (the ancestor on the DFS-tree), or no path exists and thus the start and finish intervals are disjoint.

Strongly Connected Components

A strongly connected component of G is a maximal subgraph within which every vertex is reachable from every other vertex. The component graph G^{SCC} is the graph with each vertex representing a component of G, an edges corresponding to the edges between components. This is always a directed acyclic graph (dag).

To determine the strongly connected components of a graph, we use the following algorithm:

STRONGLY-CONNECTED-COMPONENTS(G)

```
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^⊤
3  call DFS(G^⊤), but in the main loop of DFS, consider the vertices
   in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3
   as a separate strongly connected component
```

For $U \subseteq V$ we write $f(U) = \max\{u.f \mid u \in U\}$, $d(U) = \min\{u.d \mid u \in U\}$. If we have components C_0 and C_1 with an edge (u, v) with $u \in C_0$, $v \in C_1$, then if C_0 is visited first all of C_1 will descend from it, giving $f(C_0) > f(C_1)$. Otherwise all of C_1 will be visited first, and then complete before C_0 is visited again giving $f(C_0) > f(C_1)$.

From this we have that if the maximum $u.f$ are considered first, they will be the ones that have outgoing edges to unvisited components, and thus in $E^\top$ no new components will be visited.

Divide and Conquer

Master theorem

Theorem 1 *Let $a \geq 1$ and $b > 1$ be constant integers, and let $f : \mathbb{N} \to \mathbb{R}$ be a function with a positive tail. With $T : \mathbb{N} \to \mathbb{R}$ by $T(n) = aT(n/b) + f(n)$ (technically $aT(n/b) := \sum_{k=1}^{a} T([n/b]_k)$), the following asymptotic bounds hold:*
1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Dynamic Programming

Dynamic programming is a solution method similar to divide and conquer, except considering cases where the subproblems overlap, unlike divide and conquer which divides a problem into disjoint subproblems. A divide and conquer method applied to these cases would do extraneous work, due to solving every subproblem without using information from solutions to other subproblems.

To solve a problem using dynamic programming, it must have optimal substructure. This means that the solution to the overall problem contains the solutions to subproblems. To demonstrate this, we show that any solution consists of making a choice between several options, and that this choice leaves a subproblem to be solved. Suppose that we have the optimal solution to this subproblem, and determine the solution to the problem overall in terms of this. Show then that any general solution will be improved by the solution to the subproblem.

Greedy Algorithms

A greedy algorithm is a simpler alternative to dynamic programming, whereby instead of determining the manner in which local choices ought to be made from a global perspective, the local choice which is most immediately appealing is used.

A prominent greedy algorithm in graph theory is the minimum spanning tree algorithm. The logic of this algorithm is to grow a set of edges, maintaining the invariant that the set of edges is always a subset of a minimum spanning tree. To do this, note that with any cut of the edges of a graph, the edge crossing the cut with minimum weight will always be part of a minimum spanning tree. We get the following template algorithm:

GENERIC-MST(G, w)

```
1  A = ∅
2  while A does not form a spanning tree
3        find an edge (u, v) that is safe for A
4        A = A ∪ {(u, v)}
5  return A
```

The algorithms of Kruskal and Prim are derived from this. In the case of Kruskal, we look at edges in order of increasing weight, taking the union of the components they connect if the components are separate. This uses a disjoint set data structure. Prim's algorithm performs a Dijkstra-like search, building a tree while checking in increasing order of distance from its closest visited parent. Otherwise put, the algorithm adds the edge which does the least damage to the tree.