Advanced Algorithms Final Notes

Recurrences

Master Method

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$$f(n) = \begin{cases} O(n^{log_b^a}) & \text{if } f(n) = O(n^{log_b^a - \epsilon}) \\ O(f(n)logn) & \text{if } f(n) = O(n^{log_b^a}) \\ O(f(n)) & \text{if } f(n) = O(n^{log_b^a + \epsilon}) \end{cases}$$

Sum of Sequences

$$\frac{n(a_1 + a_{50})}{2}$$

Order Notation

$$O = \text{Upper Bound} \quad \Theta = \text{Tight Bound} \quad \Omega = \text{Lower bound}$$

Binary Search

1. Sorted Sequence
2. Check if middle value is the value you want
3. If not, rerun alogrithm on the top or bottom partition based on the number you wants value
4. if low $\leq$ hi then the value is not found

Power function

```
power(X,n)
if n == 0
    return 1
else if n == 1
    return X
else
    S = power(x, n/2)
    if n is odd
        return S*S*X
```

```
    if n is even
        return S * S
```

Merge sort

```
Merge(A,B,P,q,r)
//Precondition: A[p...q], A[q+1...r)] are sorted
//B is for temp work
Copy A[p...r] into B[p...r]
i = p
j = q+1
for k=p to r
    if j > r or (i <= q and B[i] <= B[j])
        A[k] = B[i++]
    else
        A[k] = B[j++]
```

Mergesort use merge after spliting each side into two parts, then run mergesort of them

Rod-Cutting problem

Input: n, P[1...n]
Output: max revenue from rod of length n

Recurrence

$$r(n) = \begin{cases} max(p_i + r(n-i)) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

Dynamic program

```
cutRod(Price[], int n)
    price[0] = 0
    for i = 1 to n
        for j = 0 to i
            max_val = max(max_val, price[j] + val[i-j-1])
    val[i] = max_val
return val[n]
```

Longest Common Subsequence

Input: X[1...m] Y[1...n]
Output: Z[1...k] that is a subsequence of X and Y

Recurrence

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & \text{if } x[i] = y[i] \\ max(LCS(i-1,j), LCS(i,j-1)) & \text{if } x[i] \neq y[i] \end{cases}$$

Dynamic program

```
for i = 0 to m
    L[i,0] = 0
for j = 0 to n
    L[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x[i] = y[j]
            L[i,j] = 1 + L[i-1,j-1]
            D[i,j] = 1
        else if L[i,j-1] > L[i-1,j]
            L[i,j] = L[i-1,j]
            D[i,j] = 2
        else
            L[i,j] = L[i,j-1]
            D[i,j] = 3
```

Reconstructing

```
\\Input: x,y,L,D
\\Output: Z[1...k]
K = L[m,n]
i = m
j = n
while k > 0
    if D[i,j] = 1
        Z[k] = x[i]
        k--
        i--
        j--
    else if D[i,j] = 2
        i--
    else
        j--
return Z
```

Activity Selection

Input: StartTimes s, FinishTimes f, Values v
Output: Find a compatable subset Q

Recurrence

$$ASP(i) = \begin{cases} 0 & \text{if } i = 0 \\ max(ASP(i-1), v_i + ASP(j)) & \text{if } i > 0 \end{cases}$$

Dynamic program

```
A[0] = 0
for i = 1 to n
    j = i-1
    while f[j] > s[i]
        j--
        A[i] = max(A[i-1], v[i] + A[j])
return A
```

Adjacency list representation

Each vertex has a list of its neighbors (In a directed graph, outbound neighbors only)

Breadth First Search (BFS)

```
//s is the source vertex
BFS{G,s)
    for each u in V
        u.color = white
        u.d = infinity
        u.pi = null

    s.d = 0
    s.color = grey
    Create a queue<vertex> Q contatining s
    while Q is not empty
        u = Q.remove()
        for each edge e =(u,v) in Adj[u]
            if v.color == white
                v.pi = u
                v.d = u.d + 1
                v.color = grey
```

```
                Q.add(v)
        u.color = black
```

Depth First Search (DFS)

```
DFS(G)
    for u in V
        u.color = white
        u.pi = null
    time = 0
    topNum = |v|
    for u in V
        if u.color == white
            DFS_Visit(u)

DFS_Visit(u)
    u.color = grey
    u.dis = ++time
    for each edge e=(u,v) in Adj[u]
        if v.color == white
            v.pi = u
            DFS_Visit(v)
    u.color = black
    u.fin = ++time
    u.top = topNum--
```

Generic MST Algorithm

```
A = Null Set
While A is not a spanning tree
    Find a cut (S, V-s) not crossed by any edge of A
    Let e=(u,v) be a light edge for (S,V-s)
    A = AU{e}
```

Prim's Algorithm

Input: G=(V,E) weights w, Root s

```
Prim(G,w,s)
    for each u in V
        u.d = infinity
        u.pi = null
```

```
    s.d = 0
    Set = null
    wmst = 0
    Create a priority queue Q with all vertices (priority = u.d)
    while Q is not empty
        u = Q.remove()
        Set = Set ∪ {u}
        wmst = wmst + u.d
        for each v in adj[u]
            if v is in Q and w(u,v) < v.d
                v.d = w(u,v)
                v.pi = u
    return wmst
```

Kruskal's Algorithm

```
A = null
for each edge e=(u,v) in sorted order of weight
    ru = Find(u)
    rv = Find(v)
    if ru != rv
        Union(ru,rv)
        A = A ∪ {(u,v)}
```

Disjoint Set ADT (Union-Find)

Find(u) - If u and v are in the same subset, Find(u) = Find(v)

```
Find(u)
    return u.rep
```

Union(u,v) - Merge subsets conatining u, containing v into a single subset

```
Union(u,v)
    for x in V
        if x.rep == v.rep
            x.rep = u.rep
```

Makeset(u) - Create a singleton set {u} represented by u

Submethods for dynamic programs used below

```
Initialize(s)
    for u in V
        u.d = 0
        u.pi - null
    s.d = 0

Relax(u,v)
    if v.d > u.d + w(u,v)
        v.d = u.d + w(u,v)
        v.pi = u
```

Bellman-Ford algorithm

Let $d_k(u)$ = Length of a shortest path from s to u that uses at most k edges

Recurrence

$$d_k(u) = \begin{cases} 0 & \text{if } u = s \text{ and } k = 0 \\ \infty & \text{if } u \neq s \text{ and } k = 0 \\ min(d_{k-1}(u), d_{k-1}(p) + w(p,u)) & \text{if } k > 0 \end{cases}$$

Dynamic program

This program uses the Initialize and Relax methods described above

```
BF_shortestPath(G,s)
Initilize(s)
for k = 1 to |v|-1
    for each edge e=(u,v) in E
        Relax(u,v)
    for each edge e=(u,v) in E                  // These lines
        if v.d > u.d + w(u,v)                   // are for verification
            raise exception "Negative cycle"    // purposes only and are not needed
```

DAG-shortest paths

Input: A directed, acyclic graph G
Output: u.d = $\delta(s,u)$ for all u in V

```
DAG(G,s)
    Initialize(s)
    Find a topological ordering of G
```

```
    for each u in V in topological order
        for each edge e=(u,v) in adj[u]
            Relax(u,v)
```

Dijkstra's algorithm

This only works when there are no negative edge weights

```
dijkstra(G,s)
    Set = null
    Initialize(s)
    Create a priority queue Q of all vertices using u.d as the priority
    While Q is not empty
        u. = Q.remove()
        Set = Set ∪ {u}
        for each e=(u,v) in Adj[u]
            Relax(u,v)
    return S
```

Floyd-warshall's algorithm

$d_k(u,v) = Length of a shortest path from u to v in which its internal nodes can only be chosen from {1,2,...,k}
Output: d = $\delta(s, u)$

Recurrence

$$d_k(u,v) = \begin{cases} w(u,v) & \text{if } k = 0 \\ min(d_{k-1}(u,v), d_{k-1}(u,k) + d_{k-1}(k,v)) & \text{if } k > 0 \end{cases}$$

Dynamic program

```
for all u,v in E
    D[u,u] = 0
    D[u,v] = infinity
    for k = 1 to |v|
        for u in V
            for v in V
                D[u,v] = min(D[u,v], D[u,k] + D[k,v])
```

Ford-Fulkerson

1.

```
for each edge(u,v) in E
    f(u,v)=0
```

2. Find the residual networks $G_f = (v, E_f)$ of this flow

```
for each edge (u,v) in E
    Add edge (u,v) to E_f with residual capacity c[f] = c(u,v) - f(u,v)
    Add edge (u,v) to E_f with residual capacity c[f] = f(u,v)
    Parallel edges need to be aggregated into a single edge
    Drop edges with 0 capacity
```

3. Find a path p from s to t in G_f (use BFS)
   if no such path exists, output f as max flow

4. Let $C_f(p) = min(c_f(e))$

```
for each edge(u,v) in p
    if(u,v) is a foward edge in G
        f(u,v) += cf(p)
    else
        f(v,u) -=cf(p)
Go back to step 2
```

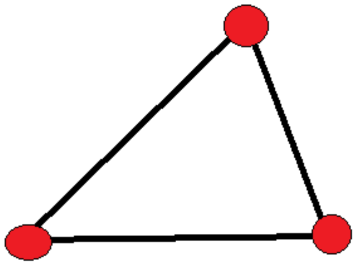
Below is a different way of looking at the problem

Input: given G=(V,E) with capacity c, source s, sink t, and path p
Output: compute a flow f from s to t of maximum value

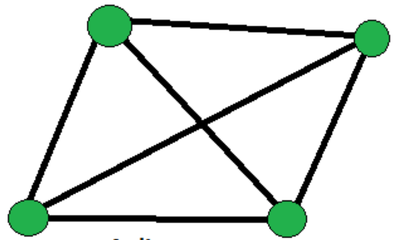
```
f(u,v) = 0 for all edges
while there is a path p from s to t in Gf such that cf(u,v) > 0 form all edges (u,v) in p
    Find cf(p) = min(cf(u,v)) across all u,v in p
    for each edge e=(u,v) in p
        f(u,v) = f(u,v) + cf(p)
        f(v,u) = f(v,u) - cf(p)
```

Clique examples

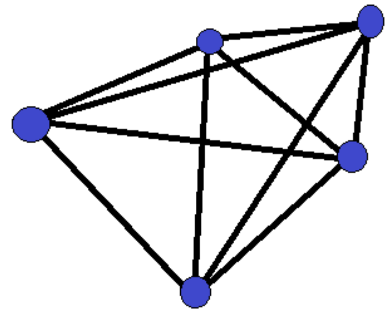
A subgraph where every node is connected to every other node in the subgraph



3-clique



4-clique



5-clique

Vertex cover examples

A subset of vertices in an undirected graph where each of the vertices in the entire graph connects to at least one of the vertices in the subset

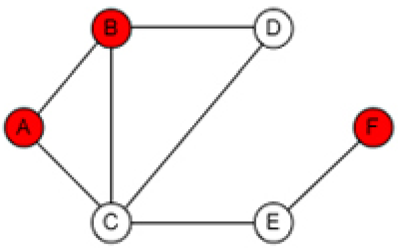


Figure 1

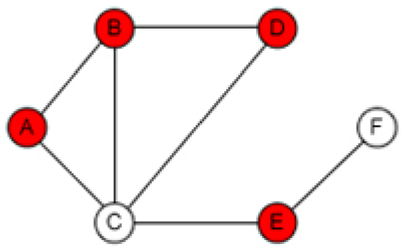


Figure 2

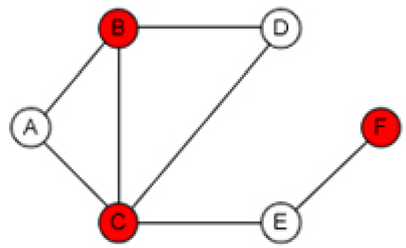


Figure 3