1.1 Basics

Definition 1.1.1 (Graph) *An (undirected) graph is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an unordered pair of nodes (the endpoints of a). A graph is simple if it has no parallel arcs and no loops.*

Theorem 1.1.2 (Degrees)

1. *The sum of the degrees of all the nodes of a graph is twice the number of arcs, and therefore even.*

2. *The number of nodes with odd degree is even.*

3. *The degree of a node is the number of incident arcs, counting loops twice.*

4. *Each arc contributes twice to the total degree, one for each endpoint.*

Definition 1.1.3 (Full Subgraphs) *Has all the edges (of nodes present)*

1. *Any subset $X \subseteq nodes(G)$ induces a subgraph $G[X]$ of G, where $G[X]$ has nodes X and $G[X]$ contains all arcs of G which join nodes in X.*

2. *G' is a full (or induced) subgraph of G if $G' = G[X]$ for some $X \subseteq nodes(G)$.*

Definition 1.1.4 (Spanning Subgraph) *Has all the nodes*
If G' is a subgraph of G and $nodes(G') = nodes(G)$, we say that G' spans G.

Aside 1.1.5 (Representations)

Adjacency Matrices

- *count each loop twice*
- *are symmetric as arcs are undirected*
- *has size n^2*

Adjacency Lists

- *list loop once only*
- *has size $\leq n + 2m$*

1.2 Isomorphism and Planar

Definition 1.2.1 (Isomorphism) *Let G, G' be graphs.*
An isomorphism from G to G' is a bijection $f : nodes(G) \rightarrow nodes(G')$ together with a bijection $g : arcs(G) \rightarrow arcs(G')$ such that if $a \in arcs(G)$ has endpoints n1 and n2 then the endpoints of g(a) are f(n1) and f(n2).

- *The adjacency matrices of G and G' are the same, except that the rows and columns may have been reordered.*

Definition 1.2.2 (Automorphism) *An automorphism on a graph G is an isomorphism from G to itself.*

Method *Find how many places first node can map to, fix first node, find how many places second node can map to, repeat until last. Multiply all places.*

Definition 1.2.3 (Planar) *A graph is planar if it can be drawn so no arcs cross.*

- *A planar graph can always be redrawn so that all arcs are straight lines which don't cross.*

Theorem 1.2.4 (Kuratowski's Theorem) *A graph is planar iff it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.*

Theorem 1.2.5 (Euler's Formula) *For a connected planar graph,*

$$\#Faces = \#Arcs - \#Nodes + 2$$

Definition 1.2.6 (Dual Graph) *A dual graph is a planar graph where all the faces are nodes and neighbours are joined by an arc.*
dual graph of a map $\leftrightarrow$ a simple planar graph

Theorem 1.2.7 (Four Colour Theorem) *Every map can be coloured using at most four colours.*

Definition 1.2.8 (k-colourable) *A graph G is k-colourable if the nodes of G can be coloured using no more than k colours.*

Definition 1.2.9 (Bipartite) *G is bipartite if nodes(G) can be partitioned into sets X and Y in such a way that no two nodes of X are joined and no two nodes of Y are joined.*

- *A graph is bipartite iff it is 2-colourable.*

1.3 Path, Circuits and Connectedness

Definition 1.3.1 (Simple Path) *A path is simple if it has no repeated nodes.*

Definition 1.3.2 (Connected Graph) *A graph is connected if there is a path joining any two nodes.*

Definition 1.3.3 (Cycles) *A graph with no cycles is called acyclic.*

Definition 1.3.4 (Euler's Path) *An Euler Path is a path which uses each arc exactly once.*

- *A connected graph has an Euler path iff there are 0 or 2 odd nodes.*

Definition 1.3.5 (Euler's Circuit) *An Euler Circuit (or Euler Cycle) is a cycle which uses each arc exactly once. It is also an Euler's Path.*

- *A connected graph has an Euler circuit iff every node has even degree.*

Definition 1.3.6 (Hamiltonian's Path) *An Hamiltonian Path is a path which uses each node exactly once. HP implies the graph is connected.*

Definition 1.3.7 (Hamiltonian's Circuit) *An Hamiltonian Circuit is a Hamiltonian Path which returns to the start node. HC implies the graph is connected with every node having degree ≥ 2.*

1.4 Trees and Directed

Theorem 1.4.1 *A tree with n nodes has n-1 arcs.*

Definition 1.4.2 (Spanning Tree)

- *A nonrooted tree T is the spanning tree for graph G if (1) T is a subgraph of G and (2) $nodes(T) = nodes(G)$.*

- *Every connected graph has a spanning tree.*

Definition 1.4.3 (Directed Graph) *A directed graph is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an ordered pair of nodes (the endpoints of a).*

- *sum of indegrees of all nodes = sum of outdegrees = number of arcs*

- *A directed graph is strongly connected if for any $x, y \in nodes(G)$ there is a path from x to y.*

2.1 Graph Traversal

Algorithm 2.1.1 (Depth-First Search) $O(n+m)$
Can adapt code to detect cycles.
If a is any arc of G with endpoints x and y, then either x ancestor of y or y ancestor of x.

```
procedure dfs(x):
    visited[x] = true; print x
    for y in adj[x]:
        if not visited[y]:
            parent[y] = x
            dfs(y)
```

Algorithm 2.1.2 (Breadth-First Search) $O(n+m)$

```
procedure bfs(x):
    visited[x] = true; print x
    enqueue(x, Q)
    while not isempty(Q):
        y = dequeue(Q)
        for z in adj[y]:
            if not visited[z]:
                visited[z] = true; print z
                parent[z] = y
                enqueue(z, Q)
```

Algorithm 2.1.3 (Topological Sort) *When performing DFS on a DAG, when we exit a node x we have already exited all nodes reachable from x.*

```
procedure dfsts(x):
    entered[x] = true
    for y in adj[x]:
        if entered[y]:
            if not exited[y]:
                abort #cycle
        else:
            parent[y] = x
            dfsts(y)
    exited[x] = true
    ts[index] = x ; index = index - 1
```

2.2 Weighted Graphs

Definition 2.2.1 (Weighted Graph) *A weighted graph (G, W) is a simple graph G together with a weight function $W : arcs(G) \rightarrow \mathbb{R}^+$.*

Algorithm 2.2.2 (Prim's Algorithm)
Classic $O(n^2)$: *$O(n)$ to find minimum fringe node and $O(n)$ iterations of loop*
Better for dense graphs

```
# Choose any node start as the root
tree[start] = true
for x in adj[start]:
    fringe[x] = true
    parent[x] = start
    weight[x] = W[start, x]

while fringe nonempty:
    Select fringe node f s.t. weight[f] is minimum
    fringe[f] = false
    tree[f] = true
    for y in adj[f]:
        if not tree[y]:
            if fringe[y]:
                if W[f , y] < weight[y]:
                    weight[y] = W[f , y]
                    parent[y] = f
            else:
                fringe[y] = true
                weight[y] = W[f , y]
                parent[y] = f
```

Priority Queue $O(m \log n)$: *$O(m)$ decrease keys $O(\log n)$ for decrease key*
Better for sparse graphs

```
Q = PQcreate()
for x in nodes(G):
    key[x] = infinity; parent[x] = null
    insert(Q, x)
decreaseKey(Q, start, 0)

while not isEmpty(Q):
    f = getMin(Q)
    tree[f] = true
    for y in adj[f]:
        if not tree[y]: # so y in Q
            if W[f, y] < key[y]:
                decreaseKey(Q, y, W[f, y]); parent[y] = f
```

Algorithm 2.2.3 (Kruskal's Algorithm) $O(m \log n)$

```
sets = UFcreate(n) # initialise UF with singletons {1}...{n}
F = {}
while not isEmpty(Q):
    (x, y) = deleteMin(Q)
    x' = find(sets,x) ; y' = find(sets,y)
    if x' != y': # no cycle
        add (x, y) to F
        union(sets,x', y')
```

Aside 2.2.4 (Union-Find)

Naive Implementation *Maintain array leader of nodes*
Find $O(1)$, Union $O(n)$

Non-binary Trees *Each set stored as a tree, merge sets by appending trees*
Find $O(n)$, Union $O(1)$

Weighted Union *Append tree of lower size to tree of greater size*
Find $O(\log n)$, Union $O(1)$

Algorithm 2.2.5 (Path Compression)

```
proc cfind(x):
    y = parent[x]
    if y == x: #x is the root
        root = x
    else:
        root = cfind(y)
        if root != y:
            parent[x] = root
    return root
```

Algorithm 2.2.6 (Djikstra's Algorithm)

- *If x is a tree or fringe node (other than start) then parent[x] is a tree node.*

- *If x is a tree node (other than start) then distance[x] is the length of shortest path, and parent[x] is its predecessor along that path.*

- *If f is a fringe node then distance[f] is the length of the shortest path where all nodes except f are tree nodes. Furthermore, parent[f] is its predecessor along that path.*

Classic $O(n^2)$

```
Input: Weighted graph (G, W) together with start & finish
Output: Length of shortest path from start to finish

tree[start] = true
for x in adj[start]:
    fringe[x] = true
    parent[x] = start
    distance[x] = W[start, x]

while not tree[finish] and fringe nonempty:
    Select a fringe node f s.t. distance[f] is minimum
    fringe[f] = false
    tree[f] = true
```

```
while not tree[finish] and fringe nonempty:
    Select a fringe node f s.t. distance[f] is minimum
    fringe[f] = false
    tree[f] = true

    for y in adj[f]:
        if not tree[y]:
            if fringe[y]:
                if distance[f] + W[f, y] < distance[y]:
                    distance[y] = distance[f] + W[f, y]
                    parent[y] = f
            else:
                fringe[y] = true
                distance[y] = distance[f] + W[f, y]
                parent[y] = f

return distance[finish]
```

Definition 2.2.7 (Heuristic Function)
Consistent:

1. *for any adjacent nodes x, y we have $h(x) \leq W(x, y) + h(y)$*

2. *$h(finish) = 0$*

Admissible: *For any node x we have $h(x) \leq$ weight of the shortest path from x to finish*

Priority Queue $O(m \log n)$

```
Q = PQcreate()
for x in nodes(G):
    key[x] = infty; parent[x] = null
    insert(Q, x)
decreaseKey(Q, start, 0)

while not tree[finish] and not isEmpty(Q):
    f = deleteMin(Q)
    tree[f] = true
    for y in adj[f]:
        if not tree[y]: # so y in Q
            if key[f]+W[f, y] < key[y]:
                decreaseKey(Q, y,key[f]+W[f, y]); parent[y] = f
```

Algorithm 2.2.8 (A* Algorithm) *Same invariants as Djikstra's Algorithm*
Classic:

```
Input: Weighted graph (G, W) together with start, finish
and consistent heuristic function h
Output: Length of shortest path from start to finish

tree[start] = true ; travelled[start] = 0
estimate[start] = travelled[start] + h[start]
for x in adj[start]:
    fringe[x] = true
    parent[x] = start
    travelled[x] = W[start, x]
    estimate[x] = travelled[x] + h[x]

while finish not a tree and fringe non-empty:
    Select a fringe node x s.t. estimate[x] is minimum
    fringe[x] = false
    tree[x] = true

    for y in adj[x]:
        if not tree[y]:
            if fringe[y]:
                if travelled[x] + W[x, y] < travelled[y]:
                    travelled[y] = travelled[x] + W[x, y]
                    estimate[y] = travelled[y] + h[y]
                    parent[y] = x
            else:
                fringe[y] = true
                travelled[y] = travelled[x] + W[x, y]
                estimate[y] = travelled[y] + h[y]
                parent[y] = x

return travelled[finish]
```

Priority Queues: *g represents shortest distance to node*

```
Q = PQcreate()
for x in nodes(G):
    g[x] = infty; key[x] = infty; parent[x] = null;
    insert(Q,x)
g[start] = 0 ; decreaseKey(Q, start, g[start] + h[start

while not tree[finish] and not isEmpty(Q):
    x = deleteMin(Q)
    tree[x] = true
    for y in adj[x]:


        if not tree[y]:
            if g[x] + W[x, y] < g[y]:
                g[y] = g[x] + W[x, y]
                decreaseKey(Q, y, g[y] + h[y])
                parent[y] = x
```

Algorithm 2.2.9 (Warshall's Algorithm) *Find all possible paths.* $O(n^3)$

```
input A
copy A into B (array of Booleans)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            B[i,j] = B[i,j] or (B[i,k] and B[k,j])
return B
```

Algorithm 2.2.10 (Floyd's Algorithm) *All Path Shortest Path.* $O(n^3)$

```
input A
set B[i,j]
    | i = j      = 0
    | arc[i,j]  = A[i,j]
    | otherwise = infty
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            B[i,j] = min(B[i,j], B[i,k] + B[k,j])
return B
```

Algorithm 2.2.11 (Bellman-Held-Karp Algorithm) *TSP.* $O(n^2 2^n)$
Input (G, W)
Choose any start $\in nodes(G)$
for $x \in Nodes \setminus \{start\}$:
  $C[\emptyset, x] = W[start, x]$
 # Process sets S in increasing order of size.
for $S \subseteq Nodes \setminus \{start\}$ with $S \neq \emptyset$:
  for $x \in Nodes \setminus (S \cup start)$:
    # Find C[S, x]
    $C[S, x] = infty$
    for $y \in S$:
      $C[S, x] = min(C[S \setminus \{y\}, y] + W[y, x], C[S, x])$

opt = infty
for $x \in Nodes \setminus \{start\}$:
  $opt = min(C[Nodes \setminus \{start, x\}, x] + W[x, start], opt)$
return opt

3.1 Searching

Aside 3.1.1 (Unordered List) *Linear Search: $W(n) = n$.*

Aside 3.1.2 (Ordered List) *Binary Search: $W(n) = 1 + \lfloor \log n \rfloor$*

- *Binary tree of depth d has $n \leq 2^{d+1} - 1$ nodes*
- *Minimality of Binary Search = Depth of Tree = $\lceil \log(n+1) \rceil$*

3.2 Orders

Definition 3.2.1 *Let $f, g : \mathbb{N} \to \mathbb{R}^+$.*

- *f is $O(g)$ iff $\exists m \in N, \exists c \in \mathbb{R}^+$ such that*

$$\forall n \geq m.[f(n) \leq c.g(n)]$$

- *f is $\theta(g)$ iff f is $O(g)$ and g is $O(f)$*

Theorem 3.2.2 (Master Theorem)

$$T(n) = aT(n/b) + f(n)$$

Where $E = \frac{\log a}{\log b}$,

1. *If $n^{E+\epsilon} = O(f(n))$ for some $\epsilon > 0$, $T(n) = \Theta(f(n))$ — first level*
2. *If $f(n) = \Theta(n^E)$, $T(n) = \Theta(f(n) \log n)$ — every level*
3. *If $f(n) = O(n^{E-\epsilon})$ for some $\epsilon > 0$, $T(n) = \Theta(n^E)$ — base level*

3.3 Sorting

Definition 3.3.1 (Balanced Trees)

- *Total path length of a tree is the sum of the depths of all leaf nodes*
- *A tree of depth d is balance if every leaf is at depth d or d-1*
- *If a tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing the total path length.*

Aside 3.3.2 (Decision Tree)

- *Binary tree of depth d has $l \leq 2^d$ leaves*
- *Sorting decision tree must have $n!$ leaves*
- *Sorting by comparison must perform $\lceil \log(n!) \rceil$ comparisons*
- *Average of $\lfloor \log(n!) \rfloor$ comparison (balanced trees of depth d or d-1)*

Algorithm 3.3.3 (Insertion Sort) *Insert $L[i]$ into $L[0..i-1]$ in correct position. Then $L[0..i]$ is sorted. $W(n) = n^2$.*

Algorithm 3.3.4 (Merge Sort) *Divide roughly into 2. Sort each half. Merge the two halves. $W(n) = n - 1 + W(\lceil \frac{n}{2} \rceil) + W(\lfloor \frac{n}{2} \rfloor) = n \log n - n + 1 = n \log n$.*

Algorithm 3.3.5 (Quicksort) *Split around first element then sort two sides recursively. $W(n) = \frac{n(n-1)}{2}$. $A(n) = n - 1 + \frac{1}{n} \sum_{s=1}^{n}(A(s-1) + A(n-s)) = n \log n$.*

- *left $< i \leq j + 1$*
- *$j \leq$ right*
- *if left $\leq k < i$ then $L[k] \leq d$*
- *if $j < k \leq$ right then $L[k] > d$*

```
Algorithm Split(left,right): # left < right
d = L[left] # pivot
i = left + 1 ; j = right
while i <= j:
    if L[i] <= d:
        i=i+1
    else:
        Swap(i, j) ; j = j - 1
Swap(left, j) ; return j
```

Algorithm 3.3.6 (Heapsort) *$O(n \log n)$: n elements, $\log n$ deleteMax*

```
Build Max Heap H out of array E

SchemeHeapSort(H):
    for i = n to 1:
        E[i] = deleteMax(H)
    deleteMax(H):
        copy element at last node into root node
        remove last node
        fixMaxHeap(H)

ArrayHeapSort(E,n):
    heapsize = n
    while heapsize > 1:
        swap(1, heapsize)
        heapsize--
        fixMaxHeap(1, heapsize)
```

Algorithm 3.3.7 (Heaps) *Build Heap = $\Theta(n)$*

```
buildMaxHeap(H):
    if H not a leaf:
        buildMaxHeap(left subtree of H)
        buildMaxHeap(right subtree of H)
        fixMaxHeap(H)

fixMaxHeap(H):
    if H is not leaf:
        largerSubHeap = left/right subheap with larger root
        if root(H).key < root(largerSubHeap).key:
            swap elements at root(H) and root(largerSubHeap)
            fixMaxHeap(largerSubHeap)

insert(Q,x): # heap as array indexed at 1
    heapsize = heapsize + 1
    E[heapsize] = x
    percolateUp(heapsize)

percolateUp(c):
    if c>1:
        parent = floor(c/2)
        if E[c].key > E[parent.key]:
            swap(c, parent)
            percolateUp(parent)
```

4 Complexity

4.1 P and NP

Definition 4.1.1 (Decision Problem) *A decision problem D is decided by an algorithm A if for any input x, A returns 'yes' or 'no' depending on $D(x)$*

Definition 4.1.2 (Cook-Karp Thesis) *A problem is tractable iff it can be computed within polynomially many steps in worst case ($W(n) \leq$ some $p(n)$).*

Theorem 4.1.3 (Polynomial Invariance Thesis) *If a problem can be solved in p-time in some reasonable model of computation, then it can be solved in p-time in any other reasonable model of computation.*

Definition 4.1.4 (P) *A decision problem $D(x)$ is in the complexity class P (polynomial time) if it can be decided within time $p(n)$ in some reasonable model of computation, where n is the input size $|x|$.*

- *Superpolynomial parallelism is unreasonable*
- *Unary numbers are unreasonable*

Theorem 4.1.5 (Function Composition) *Suppose that f and g are functions which are p-time computable. Then the composition $g \circ f$ is also p-time computable. f has only p-time to build the output so $|f(x)| \leq p(|x|)$.*

Definition 4.1.6 (NP) *A decision problem $D(x)$ is in NP (non-deterministic polynomial time) if there is a problem $E(x, y)$ in P and a $p(n)$ such that*

- *$D(x)$ iff $\exists y.E(x, y)$*
- *if $E(x,y)$ then $|y| \leq p(|x|)$ (E is poly balanced)*

Definition 4.1.7 (Many-One Reduction) *We say that D reduces to D' ($D \leq D'$) if there is a p-time computable function f such that $D(x)$ iff $D'(f(x))$*

- *Reduction is reflexive ($D \leq D$) and transitive ($D \leq D' \leq D'' \to D \leq D''$)*
- *If $D \leq D'$ and $D' \leq D$, then $D \sim D'$*
- *If $D \leq D'$ and $D' \in P$, then $D \in P$*
- *If $D \leq D'$ and $D' \in NP$, then $D \in NP$*

4.2 NP Complete

Definition 4.2.1 (NP-hard) *D is NP-hard if for all problems $D' \in NP$ we have $D' \leq D$. If $P \neq NP$, then $D \notin P$.*

Definition 4.2.2 (NP-complete) *D is NP-complete (NPC) if*

- *$D \in NP$*
- *D is NP-hard, where $D' \leq D$ for some know NPC problem D'*

All NPC problems are $\sim$ since they are both NP and NP-hard (harder than each other)

Theorem 4.2.3 (NP-hard Problems)

- *SAT — Cook-Levin 1971*
- *HamPath — SAT reduction*
- *HamCycle — HamPath reduction*
- *MTSP(D) — HamPath reduction*
- *TSP(D) — MTSP(D) reduction*
- *VRPC(D) — MTSP(D) reduction*
- *k-COL — SAT reduction*

(c) Unique. All STs have the same number of arcs, which is 6. Call the MST obtained T. Any other ST would have to use (at least) one of the arcs not used by T. These have weights $7, 8, 9$ and 10. Since no existing arc of T has weight > 6, this can only give a more costly ST.

9. (a) Guess an assignment of three colours to nodes and check in p-time that adjacent nodes have different colours.
More formally,
$$3\text{COL}(G) \iff \exists c.\text{VER-3COL}(G, c)$$
where $\text{VER-3COL}(G, c)$ iff c is a 3-colouring of G, and $\text{VER-3COL}(G, c) \in \text{P}$. Furthermore, $|c|$ is p-bounded in $|G|$.

10. (a) We can guess a list of nodes and then check in p-time that it is a Hamiltonian path and that it uses a.
More formally, define $\text{VER-HPA}(G, x, y, p)$ iff p is a Hamiltonian path in G which uses a. This is p-decidable and p-balanced since we can bound the size of p by the size of G. Also $\text{HPA}(G, x, y)$ iff $\exists p.\text{VER-HPA}(G, x, y, p)$.
(b) We have shown that $\text{HPA} \in \text{NP}$ in part (a). To show that HPA is NP-hard, we show $\text{HP} \leq \text{HPA}$.
Let $f(G)$ be G with two new nodes x, y, with x joined to every node of G, and with y joined to x by arc a. Then f is clearly p-time.
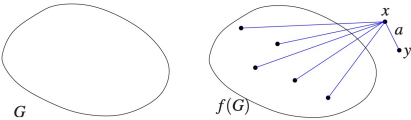
Suppose that G has an HP $x_1, \dots, x_n$. Then $f(G)$ has an HP $y, x, x_1, \dots, x_n$ using a.
Conversely, suppose that $f(G)$ has an HP using a. Clearly y must be an endpoint since it has degree one. The HP must be of the form $y, x, x_1, \dots, x_n$. Then $x_1, \dots, x_n$ is an HP of G.