Algorithms Revision Notes

Kevalee Shah

May 30, 2019

Contents

1 Big O Notation

- $T(n)$ is $\mathcal{O}(f(n))$ iff for some constants c, n_0 $T(n) \leq cf(n), \quad \forall n \geq n_0$.

  upper bound
- $T(n)$ is $\Omega(f(n))$ iff for some constants c, n_0 $T(n) \geq cf(n), \quad \forall n \geq n_0$.

  lower bound
- $T(n)$ is $\Theta(f(n))$ if it is $\mathcal{O}f(n)$ and $\Omega f(n)$

  Exact bound
- $T(n)$ is $o(f(n))$ if it is $\mathcal{O}(n)$ but not $\Theta f(n)$

  Upper bound but never equal to

2 Sorting Algorithms

2.1 Lowerbound for sorting

Condition 1: Number of Swaps
Consider the case such that each element is in the wrong place. E.g. 51234. Each swap must touch every position in the array and therefore there needs to

be at least $\frac{n}{2}$. As a result, the number of swaps is $\Theta(n)$

Condition 2: Number of Comparisons
There are $n!$ permutations of n items and therefore we need $\lg n!$. From Sterling's equation we know this is $\approx n \lg n$

$$\begin{aligned} \lg n! &= \lg 1 + \lg 2 + \cdots + \lg n \\ &\leq n \lg n \end{aligned}$$

2.2 Selection Sort

This algorithm always selects the min value of the remaining array

```
def selectionSort(a):
  for k from 0 to len(a) excluded:
  // assert array positions before k are sorted
  imin = k
  for j from imin to len(a) excluded:
    if a[j] < a[min]:
      imin = j
    swap(a[k], a[imin])
```

It is $\mathcal{O}(n^2)$ as in order to find the minimum element in the remaining array need to scan the whole array.

2.3 Insertion Sort

```
def insertionSort(a):
    for i from 1 included to len(a) excluded:
      j = i - 1
      while j>= 0 and a[j] > a[j + 1]:
        swap(a[j], a[j+1])
        j = j - 1
```

The worst case is when the list is in reverse order, and therefore for each of the outer loop iterations, the value being considered has to be moved all the way to the beginning.

$$\frac{1}{2}n(n-1) = \mathcal{O}(n^2)$$

2.4 Binary Insertion Sort

Insertion sort, but this algorithm is used to reduce the number of comparisons when finding where to insert the number.
First we need a function that returns the index to be inserted into.

```
def binarySearch(int[] a, int item, int low, int high):
  if(high <= low):
    if item > a[low]:
      return low + 1
    else:
      return low
  int mid = (low + high) / 2
  if item = a[mid]:
    return mid + 1
  if item > a[mid]:
    return binarySearch(a, item, mid + 1, high)
  else:
    return binarySearch(a, item, low, mid - 1)


def binaryInsertionSort(a):
  for k from 1 to len(a) excluded:
    i = binarysearch(a[0:k], a[k], 0, k-1)
    if i!= k: (to check k is not already in place)
      tmp = a[k]
      for j from k-1 down to i-1 excluded:
        a[j+1] = a[j] (in order to make space for a[k] to be
     swapped in)
      a[i] = temp
```

2.5 Bubble Sort

Swap adjacent elements if need be. Terminate when no swaps done in a pass.

```
def bubbleSort(a):
  repeat:
    swapsDone = false
    for k from 0 to len(a) - 1 excluded:
      if a[k] > a[k+1]:
        swap(a[k], a[k+1])
        swapsDone = true
  until swapsDone = false
```

Worst case is $\mathcal{O}(n^2)$
Advantage is that it is linear time for ordered input, compared to selection sort which is $\mathcal{O}(n^2)$.

2.6 Merge Sort

```
def mergeSort(a):
  if len(a) < 2:
    return a

  h = int(len(a) / 2)
  a_1 = mergesort(a[0:h])
  a_2 = mergesort(a[h:len(a)])

  a_3 = new array size len(a)
  i1 = 0
  i2 = 0
  i3 = 0
  while i1 < len(a_1) and i2 < len(a_2):
    a_3[i3] = smallest(a_1, i1, a_2, i2)
    i3 = i3 + 1
```

Subtleties to do with memory management.
A disadvantage is that it can't be sorted in place - need $\frac{n}{2}$ extra space.
Analysis:

$$
\begin{aligned}
f(n) &= 2f(\frac{n}{2}) + kn \qquad \text{(for merging)} \\
\text{Let } n &= 2^m \\
f(n) &= f(2^m) \\
&= 2f(2^{m-1}) + k \cdot 2^m \\
&= 2(2(f(2^{m-2}) + k \cdot 2^{m-1})) + k \cdot 2^m \\
&= 2^2 f(2^{m-2}) + 2k \cdot 2^m \\
&\vdots \\
&= 2^m f(1) + mk \cdot 2^m \\
&= k_0 n + n \lg nk \qquad \text{as } m = \lg(n) \qquad = \mathcal{O}(n)
\end{aligned}
$$

2.7 Quicksort

- This has an average time complexity of $\mathcal{O}(n \lg n)$ but in the worst case it can be $\mathcal{O}(n^2)$
- A huge advantage is that it is in place
- However it is not stable

The following conditions must be true during the algorithm:

- `a[ibegin : ileft]` $\leq$ `pivot`

- `a[iright : iend]` $>$ `pivot`
- `ileft < iright`

Keep on moving ileft to the right and iright to the left until the conditions no longer hold true
Then:

- If they have stopped and have met - swap iright and pivot. Recursively sort the two subarrays formed.
- Else - it means that `a[ileft] > pivot` and `a[iright - 1]` $\leq$ `pivot`. Therefore swap them and continue moving ileft and iright

Analysis:
If a pivot is chosen so that the array splits in two, then we have the recurrence relation $f(n) = 2f(\frac{num}{2}) + kn$. This is $\mathcal{O}(n \lg n)$.
In the worst case the pivot splits the array into arrays of size 1 and $n-1$. Then we have the recurrence $f(n) = f(n-1) + kn$ which is $\mathcal{O}(n^2)$.
Some variations of the algorithm take the pivot at random. Others use insertion sort for smaller subarrays.
The actual reccurence relation for the average case is:

$$f(n) = kn + \frac{1}{n}\sum_{i=1}^{n} f(i-1) + f(n-i) = \mathcal{O}(n \lg n)$$

```
def quicksort(A, p, r):
  if p < r:
    q = partition(A, p, r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)


def partition(A, p, r):
  pivot = A[r]
  i = p-1
  for j from p to r-1:
    if A[j] <= pivot:
      i = i + 1
      swap(A[i], A[j])

  swap(A[i+1], A[r])

  return i + 1
```

2.8 Heap Sort

Max heap property: A[parent(i)] $\geq$ A[i]
To do heap sort, first need to heapify the array, and then need to repeatedly get the max to put at the end of the array.

- Heapify takes $\lg n$ time
- Heapsort takes $n \lg n$
- Build max heaptakes n time

Key facts:

- n-element heap has height $\lfloor \lg n \rfloor$
- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h
- $\lfloor \frac{n}{2^h} \rfloor$ nodes of height $\geq$ h

```
def buildMaxHeap(A):
  for i from A.length/2 to 1:
    maxHeapfiy(A, i)
```

Proof that `buildMaxHeap` is $\mathcal{O}(n)$

- $\lfloor \frac{n}{2} \rfloor$ can move down 1 level
- $\lfloor \frac{n}{4} \rfloor$ can move down 2 levels - but one level already counted above, therefore can move down one more level
- $\lfloor \frac{n}{8} \rfloor$ can move down 3 levels - but two already counted above, therefore can move down one more level

$$\sum_{n=1}^{\lg n} \frac{n}{2^h} \leq n \sum_{n=1}^{\infty} 2^{-h} \leq n \cdot \left(\frac{1}{2} + \frac{1}{4} + \cdots \right) \leq n$$
$$\therefore \mathcal{O}(n)$$

2.9 Counting Sort

If we know that the numbers are in a certain range, e.g. $0 - n - 1$ then we have an array of size n, and initalise it to 0. Then we go through the list and increment array when the index appears. It is then easy to get an ordered list from this. An advantage is that it is linear time and stable, but a disadvantage is that it is not inplace.

2.10 Bucket Sort

Relies on knowing that data lies in a range. Requires a uniform distribution. As the assumption is that the buckets distribute the data evenly. Go through the list and put the data in buckets. The sort the buckets, e.g. insertionSort. Then put the data back together. If it is not evenly distributed then the worst case is $\mathcal{O}(n^2)$ otherwise it is linear time.

2.11 Radix Sort

Similar to bucket sort, but sort from Least Significant to Most Significant digit. Each pass use a stable algorithm to sort by the digit. The number of passes is the number of digits.

2.12 Stable

Stable algorithms are when the order of repeated elements doesn't change. Stable algorithms are:

- insertion sort
- selection sort
- merge sort

The unstable algorithms are:

- heapsort
- quicksort

3 Dynamic Programming

3.1 Features

- Many choices that have a scoare that needs to be minimised or maximised
- No of choice are exponential and therefore brute force doesn''t work
- Optimal solution made up of optimal solutions of subproblems
- Optimal solution to sub problems are required to solve many higher level problems - which is why divide and conquer alone doesn't work

4 Data Types

4.1 Abstract Data Types

It is a specification of list of operations that are needed for the datatype.
`precondition` - what must be true before the method is called
`behaviour` - imperative what the method must do `precondition` - what will be true after the method has been called

4.2 Stack

Follows the Last In First Out principle. Supports: `.isEmpty()`, `.push(item x)`, `.pop()` and `.top()`. Two ways of implementing a stack:

1. Have an array of values and an index pointing to the op of the stack. If you add to array you increment the index and if you pop then decrement
2. Linked list - to push you add an element to the head, and to pop you remove the head.

Has many uses, e.g. reverse polish notation $3x4 + 5 \implies 3\,4\,\times 5+$

4.3 Queue

Obeys the First In First Out principle. Supports: `.isEmpty()`, `.put(item x)`, `.get()`, `.first()`. Can also have `deque()` data structure where you can add and remove from both ends.

20mm

4.4 Dictionaries

They support: `.set(k, v)`, `.isEmpty()`, `.get(k)`, `.delete()`.
Issues to do with addressing. Could hold data in a list and scan to find a key. In order to add a key, value pair, could just add to the front, but risks repeats, or could scan the whole list first to check if key exists, but that is slow. If the keys can be ordered, can use a binary search tree. Then searching is $\Theta(\lg n)$
Important: If a node in a BST has two children, it's successor has no left children.
Deleting in a binary search tree can be difficult:

- Deleting a leaf is easy
- Deleting a node with one child - replace with child
- Deleting a ndoe with two children - replace with successor, who has max one child and therefore method 2 can be used to solve that

4.5 Red-Black Trees

- Every node is black or red
- The root is black
- All leaves are black - and they contain no key/value pair
- Red nodes have two black children
- From every node the path to leaves contain the same number of black nodes.

    All paths have a node count of between b and $2b$

4.5.1 Rotations

4.6 Insertion

There are three cases for inserting a node:

1. Case 1: Uncle is Red
   Swap colours of $p\,u\,g$

    g is root and therefore cannot be red, change it back to black

    g has red parents, therefore cannot be red - case 2 or 3 where $g = n'$

2. Case 2: Uncle is Black ZigZag
   Rotate pn edge so that p is n's child

    Might result in a case 3

3. Case 3: Uncle is Black Straight
   Swap colours of $p\,g$, then rotate pg edge

4.7 B Trees

Used for when you are storing on disk and dont want to many disk searches as they are slow, and therefore want a dense, not very deep tree.

- Internal Node: has key, payload and children
- Leaf node: has no key, payload or children
- Eahc key in a node and node holds associated payload
- All leaf nodes are at the same distance from root
- All internal nodes have max $2t$ children, and min t children (except for the root)
- A node has c children it is has $c-1$ keys.

4.8 Hash Tables

General case of dictionary. A hash function maps a key onto an integer - index of the array where the key value pair is stored. However sometimes hash functions for differenct keys map onto the same integer.

- Chaining - have a linked list at the index point, and just chain on all keys that map there
- Open Addressing - keep on probing until you find a free space
  - Linear Probing - $h(k)+j \mod m \mod m$. This means that key that lands on the same cell will then have the same probing sequence. This leads to primary clustering
  - Quadratic Probing - $h(k) + cj + dj^2$. This still leads to secondary clustering when the hash function maps two keys to the same cell, but gives a different probing sequence if they had different $h(k)$ values to start off with
  - Double hashing is the best option, but it is costly computing two hash functions. $h(k) + j \cdot i(k) \mod m$

5 Graphs General Terms

- Forest - undirected acyclic graph
- Tree - connect forest
- DAG - directed graph with no cycles
- Density - $\frac{E}{V^2}$
- Adjacency List - way of representing graphs that takes $\mathcal{O}(V+E)$ space. Good for sparse graphs
- Adjacency Matrix - way of representing graphs that takes $\mathcal{O}(V^2)$ space. Good for dense graphs

6 Graph Traversal

6.1 Depth First Search

Recursive way of doing depth first search:

```
def dfs_recurse(g,s):
  for v in g.vertices:
    v.visited = False
  visit(s)

def visit(v):
  v.visted = True
  for w in v.neighbours:
    if w.visited = False:
      visit(w)
```

Depth first search using a stack:

```
def dfs_stack(g, s):
  for v in g.vertices:
    v.visited = false
  s.visited = true
  toExplore = Stack([s])

  while toExplore != empty:
    v = toExplore.pop
    for w in v.neighbours:
      if w.visited = false:
        w.visited = true
        toExplore.push(w)
```

The time complexity for this is $\mathcal{O}(V+E)$ as it first needs to set every vertex to not seen, and the second half of the code is run for every edge in the graph.

6.2 Breadth First Search

We visit every node at distance d before moving onto nodes at distance $d+1$.

```
1  def bfs(g, s):
2    for v in g.vertices:
3      v.visited = false
4    s.visited = true
5    toExplore = Queue([s])
6
7    while toExplore != empty:
8      v = toExplore.pop
9      for w in v.neighbours:
10       if w.visited = false:
11         w.visited = true
12         toExplore.push(w)
```

Identical to `dfs_stack` but with a queue instead.
The code can also0 be adapted to give a path between two nodes, if it exists - just need to keep track of from what previous node the next node was visited, and then can back-track.
The same analysis as dfs shows the runtime for this is $\mathcal{O}(V + E)$

7 Djikstra's Alogrithms

If each edge in a graph has a cost associated with it, we want to find the shortest path from $s \to t$. Djikstra's algorithm is used for this.

```
1  def djikstras(g, s):
2    for v in g.vertices:
3      v.distance = infinity
4    s.distance = 0
5    toExplore = PriorityQueue([s], lambda v : v.distance)
6
7    while toExplore != empty:
8      v = toExplore.popmin()
9      # Assert 1: v.distance is the true distance from s to v
10     # Assert 2: v is never put back into the PriorityQueue
11     for w in v.neighbours:
12       dist_w = v.distance + cost(v to w)
13       if dist_w < w.distance:
14         w.distance = dist_w
15         if w in toExplore:
16           toExplore.decreaseKey(w)
17         else:
18           toExplore.push(w)
```

Important to remember that cost of an edge must be positive. Using a fibonacci heap to implement the priority queue, we get that the time complexity for djikstra's is:

- Line 8 is done for each node and therefore takes $\lg v \times v = \mathcal{O}(V \lg V)$
- Lines 12-18 are done for every edge and therefore takes $1 \times E = \mathcal{O}(E)$
- In total the complexity is $\mathcal{O}(E + V \lg V)$

We need to prove that this algorithm is correct - that is terminates and when it does that the v.distance for $v \in$ g.vertices is the true distance from $s \to v$. Furthermore the two assertions are two.

7.1 Proof of Assertion 1

Suppose this assertion fails at some point in execution. Let the vertex for this fails be v. Consider the shortest path from $s \to v$. This can be written as

$$s = u_1 \to \cdots \to u_i \to \cdots \to u_k = v$$

Let u_i be the first vertex which has not been popped. If they have all been popped then let $u_i = v$. Then,

$$\begin{aligned} dist(s \to v) &< v.dist \\ &\leq u_i.dist \\ &\leq u_{i-1}.dist + cost(u_{i-1} \to u_i) \\ &= dist(s \to u_{i-1}) + cost(u_{i-1} \to u_i) \\ &\leq dist(s \to v) \end{aligned}$$

This is a contradiction and therefore the assertion must be true.

7.2 Proof of Assertion 2

Once a vertex has been popped, assertion 1 guarantees thaat v.distance is dist(s to v). The only way v can be put back into toExplore, is if there is a shorter path to v. This means assertion 1 would be incorrect, which is not true, therefore assertion 2 is correct.

8 Bellman Equation

We have a graph in which the vertices represent the states an 'agent' can be in. Edges represent actions that take it from one state to another. Edges have a weight associated with them. Goal is to best sequence of weights to achieve some end state.

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ weight(i \to j) & \text{if there is an edge } i \to j \\ \infty & \text{otherwise} \end{cases}$$

Let $M_{ij}^{(l)}$ be min weight from i to j in l steps. Then,

$$\begin{aligned} M_{ij}^{(1)} &= W_{ij} \\ M_{ij}^{(l)} &= \min_k \left\{ W_{ik} + M_{kj}^{(l-1)} \right\} \end{aligned}$$

We can also write this as,

$$M - ij^{(l)} = (W_{i1} + M_{1j}^{(l-1)}) \wedge (W_{i2} + M_{2j}^{(l-1)}) \wedge \cdots \wedge (W_{in} + M_{nj}^{(l-1)})$$

This looks a lot like matrix multiplication, but with + instead of × and ∧ (meaning min) instead of +.
Problem Statement: Consider a directed graph with no negative weight cycles. Compute the min weight path between every pair of vertices.
The algorithm we use is then:

```
Let M^(1) = W
Compute M^(n-1) using M^(l) = W ⊗ M^(l-1)
Return M^(n-1)
```

We know this algorithm is correct as we have defined $M_{ij}^{(l)}$ to be the min path for all paths $\leq l$. We need to show that $n-1$ is enough to find all min paths. If the graph has n nodes, if there is a path longer than $n-1$ edges, then there must be a cycle. If there is a cycle from the definition of the graph we know it doesnt have a negative weight and therefore would only add to the min weight. As a result the min weight path is $\leq n-1$ edges.
The runtime for this algorithm is the time taken for matrix multiplication for every vertex. Therefore this is $V^3 \times V = \mathcal{O}(V^4)$. However using the trick of squaring instead of multiplying, we can reduce this to $\mathcal{O}(V^3 \lg V)$:
E.g. $V = 10$

$M_{(1)} = W$

$M_{(2)} = M_{(1)} \bigotimes M_{(1)}M_{(4)} \quad = M_{(2)} \bigotimes M_{(2)}M_{(8)} = M_{(4)} \bigotimes M_{(4)}M_{(16)} \quad = M_{(8)} \bigotimes M_{(8)} = M_{(9)}$ no neg w

9 Bellman-Ford

If we have found a path from $s \to u$ and there is an edge $u \to v$, we have a path $s \to v$. The min path found so far is stored in a variable called `minweight`:

$$\text{if } v.minweight > u.minweight + weight(u \to v):$$
$$v.minweight = u.minweight + weight(u \to v)$$

This updating is called *relaxing* the edge $u \to v$

Problem Statement: Given a directed graph, where each edge has a weight. (i) If there are no negative weight cycles reachable from s. For every vertex compute the min-weight path. (ii) Report that there is a neg weight cycle.

```
def bf(g,s):
  for v in g.vertices:
    v.minweight = ∞
  s.minweight = 0

  repeat len(g.vertices) - 1 times:
    # relax all edges
    for (u,v,c) in g.edges:
      min(u.minweight + c, v.minweight)
      # Assert: v.minweight >= true min weight from s to v
```

```
11
12  for(u,v,c) in g.edges:
13    if u.minweight + c < v.minweight:
14      throw "Negative Cycle"
```

The runtime for this algorith is $\mathcal{O}(VE)$ - as all edges are iterated over V times.

Proof of Assertion: $w(v)$ denotes the true minweight for node v. $w(v) = -\infty$ if there is a path that includes a neg weight cycle. The algorithm only updates v.minweight if there is a valid path to v. Therefore assertion true.

9.1 Proof Case 1

Pick any vertex v and consider the min weight path

$$s = u_0 \to u_1 \to \cdots \to u_k = v$$

Consider the loop:

- u_0.minweight is correct as it is equal to 0
- After one iteration u_1.minweight is correct as if there was a lower weight path to u_1 then the path we've assumed to be the minweight path is not the minweight path
- After two iterations u_2.minweight is correct
- $\cdots$

By the time we get to the final iteration, all vertices have the correct minweight and therefore an exception is never thrown

9.2 Proof Case 2

Suppose there is a negative weight cycle reachable from s

$$s \to v_0 \to v_1 \to \cdots \to v_k \to v_0 \qquad \text{totalweight}(v_0 \to v_k) < 0$$

If the algorithm terminates without throwing an exception, then all edges must pass the test on line 13:

$$\begin{aligned} v_0.minweight + weight(v_0 \to v_1) &\geq v_1.minweight \\ v_1.minweight + weight(v_1 \to v_2) &\geq v_2.minweight \\ &\vdots \\ v_k.minweight + weight(v_k \to v_0) &\geq v_0.minweight \end{aligned}$$

Putting these together gives

$$\begin{gathered} v_0.minweight + weight(v_0 \to v_1) + \cdots + weight(v_k \to v_0) \geq v_0.minweight \\ weight(v_0 \to v_1) + \cdots + weight(v_k \to v_0) \geq 0 \end{gathered}$$

This cannot be true as we assumed that it was a cycle of negative weight and therefore the exception will be thrown.

10 Johnson's Algorithm

Problem Statement: Given a directed graphwhere each edge has a weight. (i)if no neg weight cycles then compute the min weight path between every pair of nodes. (ii) if there is a neg weight cycle then detect that is the case.

1. Create a helper graph, by adding vertex s to the graph, which has an edge to every vertex in the graph with weight 0
2. Using Bellman-Ford compute the minimum weight from $s \to v$. Let this be d_v. $d_v < 0$ if there is an edge with negative weight that results in a min weight path lower than 0
3. If Bellman-Ford detects a neg cycle, then terminate.
4. After B-F is complete, remove node s and recompute the edge weights:

   $$w'(u \to v) = d_u + w(u \to v) - d_v$$

   i.e. The shortest distance from $s \to u$ + original edge weight - the shortest distance from $s \to v$

   $w'(u \to v) \geq 0$

   Proof: $d_v \leq d_u + w(u \to v)$ - due to the relaxation equation
5. Run Djikstra on the new graph, V times once from every vertex.

   Claim: min weight paths in the new graph are the same as in the original graph

   Proof:
   Pick any two vertices p, q and consider any path between them. $p = v_0 + \cdots v_k = q$. Consider the weight of the path in the tweaked graph:

   $$\begin{aligned} &= d_p + w(v_0 \to v_1) - d_{v_1} + d_{v_1} + w(v_1 \to v_2) - d_{v_2} + \cdots \\ &= d_p + w(v_0 \to v_1) + w(v_1 \to v_2) + \cdots + w(v_{k-1} \to v_k) - d_q \\ &= textweightinoriginalgraph + d_p - d_q \end{aligned}$$

   This shows that since $d_p - d_q$ is the same for every path from $p \to q$ the ranking of paths is the same for the tweaked graph vs the original graph.

11 Prim's

Minimum Spanning Tree - tree the connects all the vertices and has the minimum weight out of all the spanning trees.
Prim's greediily builds up a MST, using the frontier to explore idea similar to Djikstra. Pick a vertex arbitarily to start off with. Look at all the edges between the tree that we've built, and the adjacent vertices that aren't part of the tree. Pick the edge with the lowest weight.

```
def prims(g, s):
  for v in g.vertices:
    v.distance = \infty
    v.in_tree = false
```

```
  s.come_from = False
  s.distance = 0
  toExplore = PriorityQueue([s], lambda v : v.distance)

  while toExplore != empty:
    v = toExplore.popmin()
    v.in_tree = true
    # Let t be the graph made of vertices with in_tree=true
    # and edges (w-w.come_from, for w in g.vertices
   excluding s)
    # Assert: t is part of an MST for g
    for (w, edgecost) in v.neighbours:
      if (not w.in_tree) and edgeweight < w.distance:
        if w in toexplore:
          toExplore.decreaseKey(w)
        else:
          toExplore.push(w)
```

The runtime for this algorithm is the same as for Djikstra, therefore $\mathcal{O}(E + V \lg V)$

12 Kruskal

This is another algorithm used to find a minimum spanning tree. It uses a data structure called the DisjointSet. It can be used to keep track of a collection of disjoint set - sets with no common elements. Initally each vertex is in its own partition. Each edge is considered in turn, in min weight order, and if it connects two elements in different paritions, then the edge is added to the MST.

```
def kruskal(g):
  tree_edges = []
  partition = DisjointSet()
  for v in g.vertices:
    partition.addSingleton(v)
  edges = sorted(g.edges, sortKey = lambda u,v, edgeweight:
   edgeweight)

  for(u, v, edgeweight) in edges:
    p = partition.get(u)
    q = partition.get(v)
    if p != q:
      tree_edges.append((u,v))
      partition.merge(p, q)
      # Let f be the forest made of up edges in tree_edges
      # Assert: f is part of the MST
      # Assert: f has one connected component per set in
   partition

  return tree_edges
```

All DisjointSet operations take $\mathcal{O}(1)$ time and therefore the total time is:

$$\mathcal{O}(V) - \text{to add singleton vertices to partition}$$
$$\mathcal{O}(E \lg E) - \text{to sort edges}$$
$$\mathcal{O}(E) - \text{iterate through each edge to decide whether to add to tree edge}$$
$$= \mathcal{O}(V + E \lg E + E)$$
$$= \mathcal{O}(E \lg E)$$

$E \lg E$ can also be written as $E \lg V$ as:

$$V - 1 \leq E \leq \frac{1}{2}V(V-1)$$
$$\lg V - 1 \leq \lg E \leq \lg \frac{1}{2} + \lg V + \lg V - 1$$
$$\Theta(\lg E) = \Theta(\lg V)$$

To prove the correctness of the algorithms we use Prim's proof - when the algorithm merges partitions p, q consider the cut of all vertices into p vs $not - p$; the algorithm picks a min weight edge across this cut and so by the theorem we still have a MST

13 Topological Sort

We want to see if a graph has a total order. If a graph has a cycle then it cannot have a total order.
In a DAG a total ordering can always be found.

```
def toposort(g):
  for v in g.vertices:
    v.visited = False
    # v.colour = white
  totalOrder = []
  for v in g.vertices:
    if not v.visited:
      visit(v, totalOrder)
  return totalOrder

def visit(v, totalOrder):
  v.visited = True
  # v.colour = grey
  for w in v.neighbours:
    if not w.visited:
      visit(w, totalOrder)
  totalOrder.prepend(v)
  # v.colour = black
```

This is very similar to dfs and therefore the runtime is $\mathcal{O}(V + E)$

13.1 Correctness

Theorem: The algorithm terminates and returns the totalorder.
The colours can help us show the algorithm is correct.

Consider edge $u \to v$. We want to show that u appears before v. We know that each vertex is visited only once, and on that visit it is coloured grey, then some stuff happens, and then it is coloured black. Consider when u is coloured grey:

1. v is black - this means that v has already been added to the list and therefore when u is prepended, it will appear before v
2. v is white - this means that v has still not been visited. Therefore we will call visit on v. Once that call returns, after v has been added to the list, we are back to case 1, and u appears before v
3. v is grey - This means there was an earlier call to v which we are now inside. This means we are on a path from $v \to u \to v$ which is a cycle, and not allowed in a DAG and therefore it is impossible for v to be grey.

14 Networks and Flows

Definitions:

- Flow - set of edge labels such that $0 \leq f(u \to v) \leq c(u \to v)$
- Conservation of flow - $\sum_{u:u\to v} f(u \to v) = \sum_{w:v\to w} f(v \to w)$
- Value of flow - $value(f) = \sum_{u:s\to u} f(s \to u) - \sum_{u:u\to s} f(u \to s)$

  This is the net flow out of s
- Cut - Partition of vertices into two sets such that $V = S \cup \overline{S}$, with $s \in S, t \in \overline{S}$
- Capacity of Cut - $capacity(S, \overline{S}) = \sum_{u\in S, v\in\overline{S}:u\to v} c(u \to v)$

14.1 Max-Flow Min-Cut Theorem

Theorem - For ant flow f and cut $(S, \overline{S})$,

$$value(f) \leq capacity(S, \overline{S})$$

Therefore,

$$maxflow \leq mincut$$

The theorem states that it is possible to find this bound

Proof

$$
\begin{aligned}
value(f) &= \sum_u f(s \to u) - \sum_u f(u \to s) \\
&= \sum_{v \in S} \left(\sum_u f(s \to u) - \sum_u f(u \to s) \right) \\
&= \sum_{v \in S} \sum_{u \in S} f(v \to u) + \sum_{v \in S} \sum_{u \notin S} f(v \to u) \\
&\quad - \sum_{v \in S} \sum_{u \in S} f(u \to v) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) \\
&= \sum_{v \in S} \sum_{u \notin S} f(v \to u) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) \\
&\leq \sum_{v \in S} \sum_{u \notin S} f(v \to u) \\
&\leq \sum_{v \in S} \sum_{u \notin S} c(v \to u) \\
&= capacity(S, \overline{S})
\end{aligned}
$$

14.2 Ford-Fulkerson

Problem Statement: Given a weighted directed graph g, with source s and sink t, find a flow $s \to t$ of maximum value.

```
def ford-fulkerson(g, s, t):
  # let f ble a flow initally empty
  for u → v in g.edges:
    f(u → v) = 0

  # Define a helper graph for finding augmenting path
  def find_augmenting_path():
    for each pair of vertices v, w in g:
      if f(v → w) < c(v → w): h has an edge v → w with the
    label forwards
      if f(v → w) > 0: h as an edge (v → w) with the label
    backwards
    if h has a path from s to t:
      return path with labels
    else:
      # There is a set of vertices reachable from c
      # This is called the cut associated with f
      return none

  while True:
    p = find_augmenting_path()
    if p is none:
      break
    else:
      let the vertices of p be s = v_0, v_1, ⋯, v_k = t
      δ = ∞ # amount to augment the flow by
```

```
        for each edge (v_i, v_{i+1}) along p:
          if the edge has label forward:
            δ = min(δ, c(v_i, v_{i+1}) − f(v_i, v_{i+1}))
          else:
            δ = min(δ, f(v_i, v_{i+1})
        # assert: δ > 0
        for each edge (v_i, v_{i+1}) along p:
          if the edge has label forward:
            f(v_i, v_{i+1}) = f(v_i, v_{i+1}) + δ
          else:
            f(v_i, v_{i+1}) = f(v_i, v_{i+1}) − δ
        # Assert: f is still a flow
```

The while loop is guaranteed to terminate if the capacities are of integer value. Then we can only increment and decrement the flow by integer values, and the flow cannot exceed the capacity nor decrease below 0. The while loop is run f^* times, where f^* is the value of the max flow.
Runtime also depends on implementation of finding the augmenting path. If using dfs then the runtime is $\mathcal{O}(f^*(V+E))$. As all vertices can be reached from s, then we know $E \geq V-1$ and therefore we can write this as $\mathcal{O}(f^*E)$

14.2.1 Proof

Theorem The algorithm terminated and the flow is f^*. THen let S^* be the cut associated with the flow.

1. the value of f^* is equal to the capacity of the cut
2. f^* is a max flow

Proof of 1
$f^*(w \to v) = 0$ for all $v \in S^*$ and $w \notin S^*$ therefore we have

$$\sum v \in S \sum_{u \notin S} f(v \to u) = \sum v \in S \sum_{u \notin S} f(v \to u)$$

We also know that $f^*(v \to w) = c(v \to w)$ therefore $f{*}^* = capacity(S*, \overline{S*})$

Proof of 2
From the max-flow min-cut theorem we know that $value(f) \leq capacity(S, \overline{S})$.
Let $S = S^*$ - the final cut
Therefore the maximum flow is

$$\max_{all\ flows\ f} value(f) \leq capacity(S^*, \overline{S^*})$$

However from part 1 we know that f^* achieves this bound and therefore we have that f^* is the max flow. This cut is called the bottleneck cut

15 Matchings

Bipartide Graph: Vertices are split into two sets and all edges have one end in one set and the other end in another set.

Matching: A selection of some or all of the graph's edges such that no vertex is connected to more than one edge in this selection.
Size: Number of edges the matching contains
Maximum matching: Largest possible matching
e.g. Kidneys to people who need transplant

Problem Statement: Find a maximum matching in a bipartide graph
a

1. Add source vertex with edges to the left set
2. Add sink vertex with edges from the right set
3. All edges have capacity 1
4. All original edges are directed edges from left set to right set
5. Run Ford-Fulkerson from $s \to t$
6. Interpret flow as matching

16 Aggregate Analysis

Definitions:

- Aggregate Analysis: total worst case of a sequence of operations.
- Amortized Cost: Let there be a sequence of k operations, each with true costs $c_1, c_2, \cdots, c_k$. Suppose we invest amortized costs $a_1, a_2, \cdots a_k$ such that

  $$c_1 + c_2 + \cdots + c_j \leq a_1 + a_2 + \cdots + a_j \qquad \text{for } j \leq k$$

  $$\therefore \text{aggreate true cost} \leq \text{aggregate amortized cost}$$

  for a sequence of operations

16.1 Potential Functions

Φ is a function that maps possible states of the data structure to the real numbers ≥ 0.
Φ applied to the empty initial state is 0
For an operation with true cost c, with state before $\mathcal{S}_a nte$ and state after $\mathcal{S}_p ost$
Then we define the amortized cost as

$$c' = c + \Phi(\mathcal{S}_p ost) - \Phi(\mathcal{S}_a nte)$$