Optimisation Methods Notes

Vishnu

Saturday 25th May, 2019

Contents

1 What is Optimisation?

1.1 Combinatorial Optimisation

Most problems can be represented as (R,C) where:

R is a **finite** set of configurations, each of which is a possible solution to the problem. They effectively represent the search space of the problem, and in most cases won't be explicit.e.g.
S=1,2,3,. . . ,100 and S $= x|x \in N, x \leq 100$ are the same space, but the second is much more compact

C is the cost function, which tells us the value of each configuration. It takes in a configuration and outputs a real number.

An optimisation problem is to find the configuration with the lowest cost (i.e.arg $\min_R(r)$).If we wanted to find the configuration with the highest cost, we could just multiply the costs by -1.

1.1.1 SAT Problem

Given a boolean CNF (a boolean consisting of 'AND' with 'OR' as sub-problems (e.g. (a OR b) AND (c or d))), find a configuration of truth values for each variable that satisfies the CNF (i.e. the whole CNF results in true).
The cost function is simply 1 if the CNF is true, and 0 otherwise. X-SAT refers to a problem where the longest sub-problem's length is X. Any X > 2 can be converted to a 3-SAT by adding extra variables. 2-SAT is P complexity, 3-SAT is NP-Complete (Explained in 1.2.2).

1.2 Time Complexity

1.2.1 Algorithms

Measuring the running time of algorithms is futile: it depends on the hardware used, the background processes running on the computer, etc. Instead it's assumed that each step takes a constant amount of time, and the growth in the number of steps based on the size of the input determines the time complexity of an algorithm.
Again, rather than comparing the exact number of steps, an algorithm is abstracted to the largest influence on the number of steps. e.g. If the algorithm takes $4n^2+2n+3$ steps (n is the input size), we represent only consider the n^2. This is because, for large enough input values, the lower powers have little impact on the final value. e.g. The difference between n^2 and $n^2 + n$ at n = 1000 is 0.1%.

To represent the time complexity of an algorithm (f(x), the growth in the number of steps with the growth in the input size), we use the following three notations:

Big-O(O) An algorithm is O(g(x)) if there are some constants c and d such that $c*g(x) \geq f(x)$ when $x > d$. This gives an upper bound on the performance of an algorithm, so the **Worst Case**, and is the notation mostly used.

Big-Omega(Ω) An algorithm is Ω(g(x)) if there are some constants c and d such that $c * g(x) \leq f(x)$ when $x > d$. This gives an lower bound on the performance, so the **Best Case**.

Big-Theta(Θ) An algorithm is Θ(g(x)) if there are some constants c_1, c_2 and d such that $c_1 * g(x) \leq f(x) \leq c_2 * g(x)$ when $x > d$. This gives a tight bound on the performance, so the **Average Case**.

O and Ω could be very loose but still technically correct. e.g f(x) = n is O(n^{10}), but that doesn't tell us anything useful. Since we mostly expect large input values, we use d to remove problems at smaller values: e.g. $5n > n^2$ for $n < 5$, but n^2 grows quicker,

$$O(logn) < O(n)(Linear) < O(n * logn) < O(n^x)(Polynomial) < O(X^n)(Exponential)$$
$$O(n!) \approx O(n^5).$$

1.2.2 Problems

P denotes the class of problems for which there exists a **Polynomial-Time** algorithm to solve them. These are computationally 'easy'. e.g. Shortest Path with Positive Weights, Linear Programming

NP problems can't be solved, but a solution can be verified, in polynomial time. Alternatively, they can be solved in polynomial time by a **Non-deterministic Turing Machine** (a type of automaton with memory). Finding a polynomial solution to one of these is an ongoing problem. e.g. Travelling Salesman Problem, Boolean SAT

Chess is NP-Hard (as hard/harder than every other NP problem), but is EXP-Complete (an exponential-time algorithm), since verification is exponential. A problem is NP-Complete if it is harder than/as hard as every other NP problem, and is also an NP problem itself.

1.3 Travelling Salesman Problem

The input is a graph (V,E) with the cost of each edge $\in E$ known. The objective is to find the path that visits every vertex $\in V$ exactly once with a minimal cost. Some problems may require the path to start and end at the same node.

This can be converted to a decision problem: given a length X, is there a path with cost $< X$? Alternatively, each vertex can be given as an (x,y) point in space, and the euclidean distance ($\sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2}$) between points is used as the cost. The classic problem is NP-Hard, the decision problem is NP-Complete.

2 Branch and Bound

2.1 Partial Configurations

A configuration is a possible solution to a problem: so a partial configuration is a partial solution. For example, a partial configuration of a TSP is a path that doesn't contain every vertex. A partial configuration can be extended (e.g. adding another vertex) to make another configuration, either complete or partial.
Every complete configuration can only be obtained from **one** partial configuration in NP problems. In EXP problems like chess, this doesn't hold. The initial partial configuration (for TSP a path with no vertices) can be extended to every other configuration.

2.2 Branching

The same as extending, this procedure generates new configurations from a partial configuration. This depends on the problem, but it always takes one step: that is, it only generates the neighbourhood of the configuration. Again using TSP as an example, extending is only adding one vertex: if two were added at a time, some configurations might be missed.
By making every possible branch starting from the initial configuration (state), the entire search space of the problem can be traversed. However, blindly branching will take a very long time.

2.3 Bounding

Rather than branching every possibility, this uses the cost function to decide whether or not to follow a branch. The bounding procedure keeps a track of the best cost found so far, and when branching checks the best cost obtainable from the generated state or its descendents. If this cost is worse (higher) than the current best cost, the algorithm doesn't follow the branch, since it can't provide a better configuration.
When calculating the cost, the algorithm normally ignores validity to get a faster result. For example in TSP, the cost of a partial configuration is:

$$bound(PC) = cost(PC) + \text{min. cost edge from all nodes not in the PC} \tag{1}$$

Obviously, this might not be the cost a valid path: but it represents a lower bound. For a complete configuration, this is just the cost of the path. Creating a minimum valid path would take as long as generating all the descendents from the state, so defeats the purpose.

2.4 Algorithm

```
A to_visit = [initial_state], best_bound = $\infty$
B curr = Null, best = Null
C while to_visit is not empty:
    1 curr = state from to_visit with lowest bound
    2 remove curr from to_visit
    3 if $bound(curr) \geq best\_bound$ : (If the best unvisited state is worse than the current best, the search is over)
          break
    4 else:
        i. children = branch(curr)
        ii. for child in children:
            I. if $bound(child) < best\_bound$ :
                $\alpha$. if child is a complete configuration: (We've found a new best solution)
                      best = child, best_bound = bound(child)
                $\beta$. else: (We've found a branch that might lead to a best solution)
                      to_visit.add(child)
```

This algorithm is greedy, in that it always chooses the node with the lowest cost at each iteration. However, since it keeps a track of all possible nodes it won't miss an optimal solution. Until it finds a complete configuration it will add all branches to *to_visit*, as the initial bound is infinite.

3 Shortest Path Algorithms

3.1 Graphs

A graph is normally represented as (V,E) for **Vertices** and **Edges** respectively . **Weighted graphs** have a cost/weight associated with each edge, and in **Directed graphs** each edge has a direction, a start node and an end node. In directed graphs, a **Cycle** is a set of edges that return to the same node (e.g. (a,b),(b,c),(c,a)).
A **Fully Connected graph** has a direct connection between every node. In these graphs, there are $\frac{v*(v-1)}{2}$ edges, where v is the number of vertices. In **Connected graphs**, where all the nodes are connected directly or indirectly, there are at least $v - 1$ edges, where v is the number of vertices.

3.2 Paths

In a directed graph, a path is an ordered list of edges: the **source** of a path is the start node of the first edge, and the **sink/destination** is the end node of the last edge. In a weighted directed graph, where the weight of an edge e is given by w(e), the weight of a path P is:

$$\sum_{e \in P} w(e) \tag{2}$$

For two points a and b, we define the **Shortest Path Weight** as:

$$\delta(a,b) = \begin{cases} \text{Weight of Path with Lowest Weight} & \text{, if there is a path(s) between a and b} \\ \infty & , otherwise \end{cases} \tag{3}$$

The shortest path is the path with this weight. Note that by this definition there can be **multiple shortest paths**, as long as each path has the lowest weight.
By definition, the weight of a path with a negative cycle is $-\infty$, since any path can just repeat the cycle infinitely. Most shortest path algorithms stop or break when they encounter a negative cycle, since the path is 'longer' in that it takes more edges, and the concept of $-\infty$.

3.3 Shortest Path Properties

- Any subpath (subset) of a shortest path is also a shortest path
- The concatenation of two shortest paths is not necessarily a shortest path
- A shortest path can have at most v-1 edges, where v is the number of vertices
  - If there are more than v-1 edges, this means there's a vertex has been visited more than once, so there's a cycle
  - negative-cost cycles aren't allowed by most algorithms, positive-cost cycles won't exist in a shortest path since they add weight, zero-cost cycles can be removed for no change
  - Therefore there are no cycles in a shortest path, so there can't be more than v-1 edges.

3.4 Multiplicative Path Weights

For certain applications, rather than a sum of edge costs the product is used (e.g converting between various currencies). This can be converted to a summation by converting each edge weight to a logarithm. i.e if w(a,b) is x, set it to $\ln x$. This converts the product to a sum, since $\sum_{i=0}^{n} ln(x_i) = ln(\prod_{i=0}^{n} x_i)$, so standard techniques can be used.

3.5 Single Source

Any path has a source and end (sink). A single-source is a graph where one node s is designated as the source, and the shortest path to each node is calculated from it. Each node v in such a graph has the following two attributes:

v.d is the shortest path estimate, the current upper bound on the shortest path from s to v. This is initialised to ∞ for all nodes, then s.d is set to 0.

v.π is the node before v on the current shortest path from s to v. This is initialised to Nil for all nodes, since a path hasn't been defined yet.

At the end of an algorithm, v.d $= \delta(s, v)$ for all v, and $v.\pi$ is the predecessor on the shortest path for each vertex.

3.5.1 Relaxation

The process of reducing the upper bound of a node, this is used by the following algorithms to generate the shortest paths. Where u and v are nodes, (u,v) is the edge from u to v and w is the cost function for an edge:

$$\begin{aligned} &\text{If v.d} > \text{u.d+w(u,v):} \\ &\quad v.d = u.d + w(u, v) \\ &\quad v.\pi = u \end{aligned} \tag{4}$$

For this, u.d$\neq \infty$. So in the first step of an algorithm, only the nodes connected from s can be initialised. Also, if the bound on source node of an edge (u.d in the example) is reduced, that means the edges from it can be further reduced: so care has to be taken to re-relax edges if required.

3.6 Shortest Paths Tree

A tree based on a graph G and rooted at s such that, for every other node v, the path from s to v in the tree is the shortest path from s to v in G. Unlike the traditional definition, the tree only contains one path from s to v.
If a node doesn't appear in the tree then there's no path from s to v in G. Using this approach to store shortest paths is $O(v)$ space, while using a list would be $O(v^2)$.

3.7 Bellman-Ford Algorithm

We've seen in previous sections that for a graph (v,e), the shortest path between any two nodes/vertices has at most v-1 edges. We've also seen that relaxing an edge reduces the bound on the destination node, and this relaxation spreads out from the source node. Therefore, if we relax every edge in the graph v-1 times, we guarantee that the bound on every node is equal to the shortest path weight.

Bellman_Ford(V,E):

```
A v.d = $\infty$, v.$\pi$ = Nil $\forall$ v $\in V$

B s.d = 0

C for i in range(0, v-1): (range(a,b) takes values a to b-1)

        for (a,b) in E:
            relax(a,b)

D for (a,b) in E:

        if (a,b) can be relaxed
            return False

E return (V,E)
```

Since every shortest path is less than v-1, if an edge can still be relaxed after step C that means there's a negative cycle: this would give us a weight of $-\infty$, so the algorithm just rejects it by returning false.
This algorithm is **O(VE)** - it loops through all the edges V times in total (v-1 times in step C, 1 time in step D). It returns the graph, with with d and π set for each vertex.

3.8 Improvements

- Edges only need to be updated when their source nodes are updated: the current algorithm updates every edge every time, this can be reduced by keeping a track of which nodes get updated.
- Most shortest paths aren't going to be v-1 edges long: the loop can terminate early if there are no edges to relax at a generation. A shortest path of length k will finish relaxing every node in it in generation k.
- The negative cycle check can be done in the first loop: by keeping a track of the updated nodes, if there are any nodes to relax at iteration v then there's a negative cycle

3.9 Dijkstra's Algorithm

An alternate single-source shortest-paths algorithm that is faster than the Bellman-Ford, **but can't handle negative edge weights**. If there are negative edge weights/cycles, the algorithm doesn't raise a flag: it finishes executing, but the answers produced will be incorrect.

3.10 Lemmas Used

Triangle Inequality: $\delta(u, v) \leq \delta(u, x) + w(x, v)$.i.e. the shortest path between two points can't be further relaxed

Upper Bound: $v.d \geq \delta(s, v)$ at every point of the algorithm

No Path: If there is no path between S and V, $v.d = \delta(s, v) = \infty$

Convergence: Once $v.d = \delta(s, v)$, there is no change in v.d

3.11 Algorithm

In the below, N.adj is the list of nodes that are connected to N by an outgoing edge (i.e. N→n), and extract_min(Q) returns the node with the lowest upper bound in Q.
Dijkstra(V,E):

```
A v.d = $\infty$, v.$\pi$ = Nil $\forall$ v $\in V$, s.d = 0
B S = $\emptyset$, Q = V
C while Q $\neq \emptyset$:
    1 N = extract_min(Q)
    2 S.add(N)
    3 for n in N.adj:
        relax(N,n)
D return (V,E)
```

Each edge is only relaxed once, since each node is only considered once. As the nodes are reached in a greedy order and there are no negative edge weights, this means that the nodes are reached as fast as possible and thus there shouldn't be a shorter path. The algorithm returns the graph, with with d and π set for each vertex.

3.12 Proof of Correctness

If we can prove that for any u ∈ S, u.d = $\delta(s,u)$, then at the end all nodes are in s so all have the shortest path.

- We can prove this by induction:
- At S=∅, trivially true.
- At S=$\{s\}$, s.d = $\delta(s,s) = 0$

Hypothesis We assume that for S=X, all nodes in S and the edges from them have been relaxed (this is what the algorithm does at each step)

- At S = X+$\{u\}$, a proof by contradiction:
  - Assume the contradiction, that u.d $\neq \delta(s,u)$
    - u can't be s, since s.d = $\delta(s,s) = 0$
    - There must be a path s→u, otherwise s.d = $\delta(s,s) = \infty$
  - Let y be the first node on s→u that's NOT in S (this might be u), and x be y's predecessor (so x ∈ S)
    - Since y is before or equal to u, $\delta(s,y) \leq \delta(s,u)$
    - We know that x.d and (x,y) have been relaxed, by the inductive hypothesis
    - By convergence, since x.d and (x,y) have been relaxed, y.d is relaxed as well, so y.d = $\delta(s,y)$
    - **Since the algorithm chooses the lowest bound each time**, u.d $\leq$ y.d
  - Combining the abve, this gives us $u.d \leq y.d = \delta(s,y) \leq \delta(s,u)$
  - $u.d$, by definition, can't be $< \delta(s,u)$. Therefore, $u.d = \delta(s,u)$
  - This contradicts our assumption, so it can't hold. Therefore we've proved that $u.d = \delta(s,u)$ if u is the node with the lowest bound

3.13 Implementation and Performance

The algorithm's performance depends on the implementation of Q:

- If Q is a regular list, extract_min is O(V) every time (V times across the algorithm), and updating the bound on a vertex is O(1) (E times across the algorithm) so the algorithm is $\boldsymbol{O(V^2 + E)} \approx O(V^2)$

- If Q is a binary min-heap, extract_min is O(log V) every time (V times across the algorithm), and updating the bound on a vertex is O(log V) (E times across the algorithm) so the algorithm is $\boldsymbol{O((V+E)\log V)} \approx O(E \log V)$

Therefore, this depends on if the graph has more edges or vertices. A dense graph has more edges so a regular list works best, a sparse one has more vertices so the min-heap is better. Dense/Sparse can be determined by $\frac{v^2}{\log v}$: E. E is usually lower, so the min-heap is more commonly used.

3.14 Johnson's Algorithm

Both of the above algorithms give the shortest path to each node from a single source node. To get this for each node, we could run Dijkstra/Bellman_Ford once for each node as the source. But the former can't handle negative weights, and the latter would be $O(V^2E)$.

3.14.1 Re-Weighting

An easy fix to the problem is removing all the negative edge weights. Adding a constant M to each edge to make all the weights positive would fix that problem, but then it would add M to the cost of a path for each edge in it: since negative weights give a shorter path weight with more edges, this will change the results.

The new weight function ($\hat{w}$) needs to preserve the shortest paths, and be positive for every edge. This can be achieved by giving each vertex v a value h(v), and defining $\hat{w}$ as:

$$\hat{w}(a,b) = w(a,b) + h(a) - h(b) \tag{5}$$

Using this equation, the weight of a path P would be:

$$\hat{w}(s,v) = \sum_{(a,b)\in P} w(a,b) + h(s) - h(v) \tag{6}$$

This is because, other than the source/sink, every node n on the path will have an incoming edge (so -h(n)) and an outgoing edge (+h(n)). Therefore, for any path between the nodes, the added values will be the same: this preserves the shortest paths. Calculating h(v) to remove negativity is shown in the next section.

3.14.2 Calculating h(v)

Since the aim is to remove negative weights, we need to show that for an edge (a,b) that $h(a) - h(b) \geq 0$. h(v) is computed by creating a new node, say s, connecting it to every other node with a zero-cost edge, then setting h(v) = $\delta(s,v)$.
Briefly, since s has access to every other node (and thus every possible path in the graph), $\delta(s,v)$ is the shortest path into v from any other node. Since shortest is by path weight, we can say this is the most negative path weight leading into v. If this is subtracted from any incoming path weight, it'll end up as positive. A more mathematical proof:

$$\begin{aligned}
\text{By the triangle inequality, } \delta(s,v) &\leq \delta(s,u) + w(u,v) \\
\text{Therefore, } \delta(s,v) - \delta(s,u) &\leq w(u,v) \\
\hat{w}(u,v) &= w(u,v) + h(u) - h(v) \\
&= w(u,v) + \delta(s,v) - \delta(s,u) \\
&\geq 0 \quad \forall\, u,v
\end{aligned}$$

Since we compute h(v) using Bellman-Ford, s has access to every path so it can also check if there's a negative cycle and terminate. Using Dijkstra wouldn't work, since there would be negative edges.

3.15 Algorithm

Since the algorithm has to store the shortest path for each algorithm, it uses a VxV matrix. Normally the algorithm stores either d or π for each vertex for each source, this pseudocode stores both. In the below pseudocode, G.V and G.E are the vertices/edges for a graph G.

Johnson(G,W):

```
A G' = (G.V,G.E) (G's is a copy of G, which we use to generate node weights)
B G'.V.add(s) (Add the extra node)
C for v in V: (The original list of vertices)
    1 G'.E.add((s,v))
    2 w(s,v) = 0 (Adding zero-cost edges)
D G" = Bellman_Ford(G'.V, G'.E) (G" is the graph with the shortest paths from s)
E If Bellman_Ford returned false:
      return False
F for (a,b) in E: (Setting the new weights)
      w(a,b) = w(a,b) + G".V[a].d - G".V[b].d    (G".V[v].d = h(v))
G Results = []
H for v in V: (Run Dijkstra for)
    1 Set v as source
    2 $G_v$ = Dijkstra(G.V,G.E)
    3 Results.append($G_v.V$) (Add the vertices from the solved graph to the list of
      results)
I return Results
```

All the weights will be off by the node values: they can be fixed before step H.2. The weight of the shortest path to node v from node u can be fixed by adding h(u)-h(v): essentially reversing the effects of the re-weighting.

4 Flow Networks

Flow networks are directed graphs with two additions:

- Each edge (u,v) has a capacity c(u,v)
- Two distinct nodes are marked as the source and sink

Usually, optimising a flow network is finding the maximum flow: the amount of units that can travel from the source to the sink simultaneously. Units travel along the edges, but an edge can only support up to its capacity simultaneously. Other problems may be the minimum flow that hits a certain target, or having multiple types of flow at the same time.
Most of the below algorithms assume that if an edge (a,b) exists, then the edge (b,a) also exists. If it's not present in the given graph, it's given a default capacity of zero.

4.1 Flow

The flow f(u,v) of an edge (u,v) is the amount of units passing through the edge: it's abstracted to a function that maps edges to real numbers. Flow is subject to the following:

1. f(a,b) $\leq$ c(a,b) $\forall$ (a,b) - Flow for an edge is always less than/equal to capacity
2. For all nodes except source/sink, incoming flow = outgoing flow (net flow on a node = 0)
3. If f(a,b) > 0, then f(b,a) = 0

A **saturated edge** is one where f(a,b) = c(a,b). The **value of a flow** in a network is defined as the net flow from the source/net flow into the sink.
A flow can be decomposed into **paths or cycles**, which start from the source and end at the sink. There can be at most m paths/cycles, where m is the number of edges.

4.2 Flow-Feasibility Problems

A variation on flow networks, rather than a single source/sink each vertex has a value d(v). Usually $\sum_{v \in V} d(v) = 0$.

- If $d(v) > 0$, it's a **Supply Node**. Supply nodes have an outgoing flow (negative).
- If $d(v) < 0$, it's a **Demand Node**, with a demand of abs(d(v)). Demand nodes have an incoming flow (positive).
- If $d(v) = 0$, it's a **Transitional Node**.

These problems try to find a flow such that d(v) = 0 for all nodes, and can be reduced to max. flow problems.

4.2.1 Reduction to Max. Flow Problems

1. Make a source node and connect it to each supply node s with an edge of capacity d(s).
2. Make a sink node and connect it to each demand node x with an edge of capacity d(x).
3. If the max. flow saturates the edges from the source/into the sink, that's a feasible flow.

4.3 Max Flow

More of a general approach than an algorithm, this uses the concept of residual capacity to iteratively build the optimal flow. The following algorithms use the ideas here.

4.4 Residual Capacity

The residual capacity for an edge (a,b) is defined as:

$$c_r(a,b) = \begin{cases} c(a,b) - f(a,b) & ,iff(a,b) > 0 \\ c(a,b) + f(b,a) & ,iff(b,a) > 0 \end{cases} \tag{7}$$

By the definition of a flow, if $f(u,v) > 0$, then f(v,u)=0. The residual capacity of an edge with a flow is how much more flow it can take (capacity-flow), and the backward edge has how much flow can be reversed (capacity + reverse edge flow).
A **Residual Edge** is an edge with residual capacity > 0. The **Residual Network** of a flow f is the flow network made of residual edges after applying f. This network can be used to find flows not included in f, and these can be 'added' to f.

4.5 'Adding' Flows

By using the residual network, there might be a flow which improves the current one: this flow is 'added' to the current flow, altering it and increasing the overall flow. Assuming there are two flows F_1 and F_2, adding them together is done edge-by-edge to produce the flow F'. Assuming that $F_1(u,v) > 0$ (and therefore $f(u,v) = 0$):

- If $F_2(u,v) > 0$, then $F'(u,v) = F_1(u,v) + F_2(u,v)$
  - Both flows are in the same direction for this edge, just add them together
- If $F_2(v,u) > 0$, then $F'(u,v) = max(0, F_1(u,v) - F_2(v,u))$ and $F'(v,u) = max(0, F_2(v,u) - F_1(u,v))$
  - The flows run in opposite direction, so think of them as colliding
  - The larger flow direction is set to the difference of the flows
  - The smaller flow direction is set to 0

4.6 Algorithm

```
A F = Zero Flow (f(a,b) = 0 for all (a,b))
B while True:
    1 N = Residual Network of F
    2 F' = Flow in N
    3 If F' does not exist:
        break
    4 F = F.add(F')
C return F
```

4.7 Ford_Fulkerson Method

4.7.1 Augmenting Path

Building on the definition of a residual network, a **Residual Path** is a path (collection of consecutive edges) in the residual network. An **Augmenting Path** is a residual path from the source to the sink: essentially, it's a part of a flow in the residual network. The residual capacity of this augmenting path is the lowest residual capacity of its edges. i.e for an augmenting path p:

$$c_r(p) = \min_{(a,b)\in p} c_r(a,b) \tag{8}$$

4.7.2 Path Flow

Given a residual network R and an augmenting path P, the path flow for any edge (u,v) is:

$$f_P(u,v) = \begin{cases} c_r(p) & \text{, (u,v) is on P} \\ 0 & , otherwise \end{cases} \tag{9}$$

For the path flow, the only flow in the network is on the residual path.

4.7.3 Augmentation

When adding the path flow $F_p$from an augmenting path P, the flow value is constant for every edge: its the capacity of the augmenting path, say q. This can be used to speed up addition. If an edge (a,b) $\in$ P (a$\rightarrow$b is from the source to the sink), f is the current flow and F is the resulting flow then:

- F(a,b) = f(a,b) + maxq-f(b,a),0
  - If f(a,b) ¿ 0, then the flows were in the same direction and f(b,a) = 0: so they're just added
  - if f(a,b) = 0, then the flows were in the opposite direction, and f(b,a) $\geq$ 0: f(a,b) is set to the result of removing the original flow from the augment
- F(b,a) = maxf(b,a)-q,0
  - The flow in the augmenting path is in the opposite direction: if they collided f(b,a) is set to the difference, if f(b,a) was 0 it stays at 0

4.7.4 Algorithm

```
A F = Zero Flow *(f(a,b) = 0 for all (a,b))*
B while True:
    1 N = Residual Network of F
    2 $F_p$ = Augmenting path in P
    3 If F' does not exist:
        break
    4 F = F.augment($F_p$)
C return F
```

The advantage over the basic method is its easier to find augmenting paths than full flows.

4.7.5 Performance

For a graph (V,E): constructing the residual network is $\boldsymbol{\Theta(E)}$ (checking edge by edge), finding an augmenting path is **O(E)** (For example, a breadth-first search), and augmenting is **O(V)** (each vertex appears max. once).
This means each iteration is normally O(E) (most graphs are sparse, $\frac{v^2}{\log v}$:E). The number of iterations depends on which augmenting path is selected.

The bound on the number of iterations is the size of the maximum flow ($|f*|$). This is obtained when the flow is comprised entirely of distinct paths (i.e size of flow = number of paths in flow). The Ford_Fulkerson method discovers one path at a time, so the overall runtime will be $O(|f*| * E)$.

4.7.6 Edmonds_Karp Method

A variation of the Ford_Fulkerson method: select the shortest (lowest number of edges) augmenting path each time. This can be found using a Breadth-First Search or similar. The performance is $\boldsymbol{O(VE^2)}$, with the number of iterations being **O(VE)**. This is obtained by there being at most V-1 paths in any network, and any path length repeating at most E times (the proof isn't required).

4.8 Cuts in a Flow Network

A **Cut** is splitting the nodes in a network into two groups: group S contains the source, group T contains the sink.

- The **Capacity** of a cut is $\sum c(u,v)$ where u ∈ S, v ∈ T, (u,v) ∈ E : the capacity of the edges going from S to T
- The **Net Flow** of a cut is $\sum f(a,b) - \sum f(c,d)$ where a,c ∈ S, b,d ∈ T, (a,b),c,d ∈ E : the flow from S to T minus the flow from T to s

For any cut, flow(cut) = flow(sink) ≤ capacity(cut) (equal when there is no flow from S to T). Therefore, **max. flow in a network ≤ min. capacity of all cuts**.

4.8.1 Max-Flow Min-Cut Theorem

1. Max. flow in a network ≤ min. capacity of any cut in the network
2. The following three statements are equivalent:
    (a) F is a max. flow in G
    (b) There's no augmenting path in the residual network G_F
    (c) For some cut (S,T) in G, c(S,T) = f(S,T) = F

   Proof:

a→b If there was an augmenting path, it could be used to increase the flow. The flow can't be increased, so there's no augmenting path

b→c No augmenting path means all the paths from source to sink are saturated, which means there are no reverse edges. All edges from S to T in a cut will be at capacity, all edges from T to S will be empty.

c→a When c(S,T) = F(S,T), we've shown that this is a max flow. Since f(S,T) = F, F is a max flow

This is also a proof for the Ford_Fulkerson algorithm: **when there are no augmenting paths left, a max. flow has been found**.