1 A Brand New Problem

- We’re going to start looking at a new optimization problem.
- Imagine that you are building a skyscraper.
- You need to ship giant steel beams from a foundry to the construction site. Each beam requires its own flat-bed trailer truck to transport.
- The two locations are connected by a network of (one-way) roads.
- Each road can support only a certain number of trucks traveling on it per hour without collapsing. This max rate is the road’s capacity.
- Your goal is to find a strategy to get the beams from the foundry to the job site at the highest possible rate, while not exceeding the capacity of any road.

Let’s see a simple example.

- Consider this network of roads:

  - If we send 20 beams/hour north, we can get them to the site via the zig-zag route $s \rightarrow u \rightarrow v \rightarrow t$.
  - Under this strategy, we cannot send any beams per hour south on the road $s \rightarrow v$, because the road $v \rightarrow t$ is already at capacity.
  - Can you find a way to deliver the beams faster?
  - *Idea:* at $u$, send 10 of the beams directly to $t$ and the rest to $v$.
  - Now we can accommodate 10 beams/hour from $s$ directly to $v$, for a total of 30 beams/hour delivered.
  - Note that there is no way to move more than 30 beams/hour total on the roads connecting directly to the job site, so this result is the best we can do.
Let’s formalize our problem a little bit.

- Let $G = (V, E)$ be a weighted, directed graph (the roads).
- Designate two vertices in $G$ as the source $s$ and the sink $t$.
- Each edge $e$ has an integer-valued capacity $c(e)$.
- A flow in $G$ is an assignment of rates $f(e)$ to each edge $e$ of $G$ such that
  1. $0 \leq f(e) \leq c(e)$,
  2. at each vertex $v$ other than $s$ and $t$, flow is conserved. In other words, let $f_{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$, and let $f_{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$. Then for all $v \not\in \{s, t\}$, $f_{\text{in}}(v) = f_{\text{out}}(v)$.
- (The second property insists that trucks cannot just appear or disappear on the way from the foundry to the job site!)
- The goal is to find a flow on $G$ such that $f_{\text{in}}(t)$ is maximized.
- (Equivalently, $f_{\text{out}}(s)$ is maximized, because flow into $t$ equals flow out of $s$ – you can’t gain or lose flow between them due to conservation.)

This problem is known as the maximum flow problem.

2 Designing an Algorithm

- How can we find a maximum flow?
  - Idea: pick a simple (i.e. acyclic) path from $s$ to $t$, and push as much flow along it as possible. Repeat until no more flow can be pushed.
  - The maximum amount of flow that we can push on a path equals the minimum capacity of any edge on the path (i.e., the bottleneck).
  - After we push this much flow down a path, we can remove the amount of capacity used by that flow from the edges on the path.
  - The resulting residual graph records how much capacity is left to use on each edge.
  - Wash, rinse, repeat. When no path with minimum capacity $> 0$ connects $s$ to $t$ in the residual graph, we are done.

This algorithm certainly produces a feasible flow. If we can push flow along some path in the current residual graph, can argue inductively that it can be pushed along that path in the original graph after all our previous pushes.

- We haven’t specified how to choose among alternative paths.
- Unfortunately, we can’t just choose arbitrarily!
• *Example:* our original solution chose the path \( s \rightarrow u \rightarrow v \rightarrow t \) and pushed the max possible amount (20 units) on this path. The residual graph does not connect \( s \) to \( t \).

• Our better, alternate solution is equivalent to picking the northern path \( s \rightarrow u \rightarrow t \) first and pushing 10 units, then pushing another 10 units on \( s \rightarrow u \rightarrow v \rightarrow t \) and finally 10 units on the southern path \( s \rightarrow v \rightarrow t \).

• One approach (greedy) would be to try to find a rule for how to pick the first path. Unfortunately, I don’t know how to do this so as to guarantee an optimal flow.

We need a new insight.

• It would be nice if we could redirect some existing flow to free up capacity. This would let us “back out” of a bad choice.

• For example, in our first solution, if we get stuck, we could say “move 10 units of flow from edge \( uv \) to edge \( ut \)” to free up capacity on edge \( vt \).

• But our residual graph loses any information about our previous pushes! What can we do?

• *Idea:* augment the residual graph with *back edges* that remember the flow added so far.

• For each edge \( uv \) in the original graph, add a back edge \( vu \), initially with capacity 0.

• When we push \( k \) units of flow along an \( uv \), we reduce \( uv \)'s capacity by \( k \) in the residual graph, and we increase \( vu \)'s capacity by \( k \).

• Conversely, when we push flow along \( vu \), we reduce \( vu \)'s capacity and increase that of \( uv \).

• In general, when we push flow along some edge \( e \), we reduce \( e \)'s capacity and increase that of its *mirror edge* \( \bar{e} \).

What’s the big idea?

• A back edge says “if you can redirect up to \( k \) units of flow coming into \( u \) along some other path to \( t \), you could free up to that much capacity on edge \( uv \).”

• More precisely, suppose we have a path \( s \rightarrow v \rightarrow u \rightarrow t \) in the graph, where \( sv \) and \( ut \) are forward edges, and \( vu \) is a back edge (as in the picture).
• When we push some new flow from $s$ to $t$ along path $s \rightarrow v \rightarrow u \rightarrow t$, we really mean “push the new flow along the path from $s$ to $v$, redirect some old flow that previously followed $u \rightarrow v$ along the path from $u$ to $t$, and push the new flow along the path that the old flow previously took out of $v$.”

• **Example:**

To summarize, we now have the following algorithm:

```plaintext
BUILDFLOW(G)
    $G_r \leftarrow G$ > current residual graph
    add a back edge with capacity 0 for each $e \in G_r$
    set $f(e) \leftarrow 0$ for all $e \in G_r$.
    while $G_r$ contains a simple path $p$ with min capacity $k > 0$ do
        for each edge $e$ in $p$ do
            reduce $c(e)$ by $k$ in $G_r$
            increase $c(\tilde{e})$ by $k$ in $G_r$
    return flow recorded in back edges of $G_r$
```

3 **Proving Feasibility**

The strategy used by BUILDFLOW is a cute idea, but we’d better check that it is correct, in the sense that it always results in a feasible flow.

• **Lemma:** suppose $G$ is an instance of maximum flow. The sum of all pushes made by BUILDFLOW on the original graph $G$ is a feasible flow.

• **Pf:** by induction on the number of pushes.

• Let $c_i(e)$ be the capacity of edge $e$ in residual graph $G_i$ after $i$ pushes. We claim that the flow in $G$ after $i$ pushes is feasible, and that

  1. For each forward edge $e$, the total flow along $e$ in $G$ is $c(e) - c_i(e)$.
  2. For each back edge $e$, the total flow along its corresponding forward edge $\tilde{e}$ in $G$ is $c_i(e)$.

• **Bas:** When $i = 0$, no edges have any flow, and so the claimed properties are all trivially true.

• **Ind:** Now suppose we have already pushed $i - 1$ times. By the IH, the two claimed properties and overall feasibility hold.

• Suppose the $i$th push moves $k > 0$ units of flow along some path $p$ in $G_{i-1}$. 

4
• It must be that every edge \( e \in p \) has \( c_{i-1}(e) \geq k \).

• For each forward edge \( e \), there are only \( c(e) - c_{i-1}(e) \leq c(e) - k \) units of flow on \( e \) in \( G \) already (by property (1)), so \( e \)'s capacity constraint is not violated by the push.

• For each back edge \( e \), the corresponding forward edge \( \tilde{e} \) already has \( c_{i-1}(e) \geq k \) units of flow (by property (2)), so redirecting \( k \) units of flow away from \( \tilde{e} \) does not result in a negative flow.

• Hence, the \( i \)th push does not violate any capacity constraints.

• Next, we argue that the flow in \( G \) remains conserved after the \( i \)th push.

• We may divide \( p \) into runs of adjacent forward and back edges.

• The first and last runs in \( p \) are forward, because \( p \) is acyclic – we never direct flow into \( s \) or out of \( t \).

• For each forward run, we push \( k \) units into its tail and out of its head, so conservation holds.

• For each back run, we redirect \( k \) units away from its head (into the next fwd run) and supply \( k \) units to its tail (replacing the flow lost to the next fwd run), so conservation holds.

• Finally, we check that each iteration preserves properties (1) and (2).

• For each edge \( e \) on \( p \), we reduce its capacity by \( k \) and increase its opposite edge’s capacity by \( k \).

• We can easily check that, for both forward and back edges, the impact of using \( p \) on the capacities \( c_i \) matches the changes actually made to the flow in \( G \).

• No other edges in \( G \) or \( G_i \) change. QED

4 How Good is This Algorithm?

• The above algorithm is due to Ford and Fulkerson (1956).

• We’ve checked that it always returns a feasible flow.

• We would like to prove that it always returns an optimal flow.

• We’ll get there, but meanwhile, we are already in a position to analyze its running time.

• Fact: Let \( C = \sum_{e \text{ out of } s} c(e) \). BuildFlow terminates in at most \( C \) iterations.

• Indeed, each iteration augments the flow by some amount, since it finds a path of minimum capacity \( > 0 \).

• At each step, the capacities start off integers, and so the min capacity on the chosen path is integer; hence, after the updates, all the capacities are still integers.
• But then the total flow at each iteration increases by at least 1 unit.

• The total flow cannot exceed $C$, so the algorithm terminates in at most $C$ iterations. QED

• **Cor:** $\text{BUILDFLOW}$ can be implemented in time at most $\Theta(|E|C)$.

• **Pf:** WLOG, assume that every vertex in $G$ is connected to both $s$ and $t$ by some path. (Else, we never push any flow through it!)

• Each iteration can be implemented using a BFS or DFS to identify a path of nonzero capacity from $s$ to $t$ in the current residual graph.

• For a connected graph, each of these algorithms takes time $\Theta(|E|)$.

• Once we have a path $p$, we do constant work per edge of $p$, hence $O(|E|)$ work, to update the residual graph. QED

• More to come!