Introduction to Network Flow

1 Problem

Network flow is an advanced branch of graph theory. The problem resolves around a special type of weighted directed graph with two special vertices: the **source** vertex, which has no incoming edge, and the **sink** vertex, which has no outgoing edge. By convention, the source vertex is usually labelled s and the sink vertex labelled t.

The story usually go like this: there are a bunch of junctions (nodes in the graph) and a bunch of pipes (edges in the graph) connecting the junction. The pipe will only allow water to flow one way (the graph is directed). Each pipe has also has a **capacity** (the weight of the edge), representing the maximum amount of water that can flow through the pipe. Finally, we pour an infinite amount of water into the source vertex. The problem is to find the **maximum flow** of the graph - the maximum amount of water that will flow to the sink. Below is an example of a network flow graph:

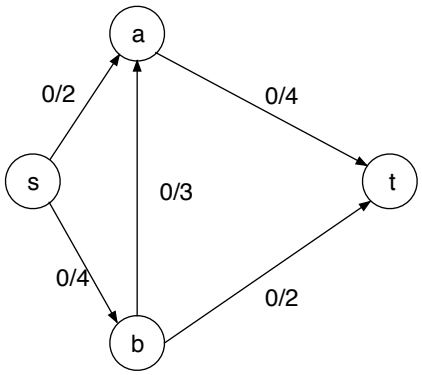

Figure 1: A simple network flow graph. There are currently no water flowing.

It is fairly easy to see that the maximum flow in the above figure is 6. We can flow 2 units of water from $s \to a \to t$, 2 units of water from $s \to b \to a \to t$, and 2 units of water from $s \to b \to t$. This gives a flow of 6 and since all incoming edge to the sink are saturated, this is indeed the maximum flow. Note that in the example, not all pipe are saturated with water.

2 Ford Fulkerson Method

It was easy to see the solution in the above example, but how do we find the solution in general? One idea is to keep finding path from s to t along pipes which still has some capacities remaining and push as much flow from s to t as possible. We will then terminate once we can't find any more path. This idea seem to work since it is exactly how we found the maximum flow in the example. However, there is one problem - we cannot guarantee which path we'll find first. In fact, if we picked the wrong path, the whole algorithm will go wrong. For example, what happens if the first path we found was $s \to b \to a \to t$. If we push as much flow as possible, then we end up with the following:

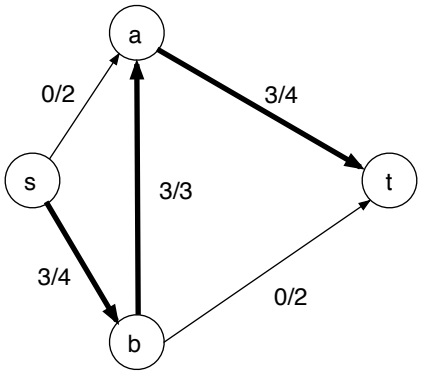

Figure 2: We have pushed 3 flow along the path $s \to b \to a \to t$

Now we have run into a problem: our only options left are to push 1 unit of water along the path $s \to a \to t$ and 1 unit of water along $s \to b \to t$. After that, we won't be able to find any more path from s to t! Yet, we have only found 5 flow, which is not he maximum flow. Thus, the idea is not optimal since it depends on how we picked our path. While we can try to find a "good" path-picking algorithm, it would be nice if the algorithm is not dependent on the paths we chose.

A crucial observation is that there is actually another path from s to t other than the two that we mentioned above! Suppose we redirect 2 units of water from $b \to a \to t$ to $b \to t$, this will decrease the amount of water running through the pipe (a, t) to 0. Now we have a path from $s \to a \to t$ in which we can flow 2 units of water! The graph now looks as follows:

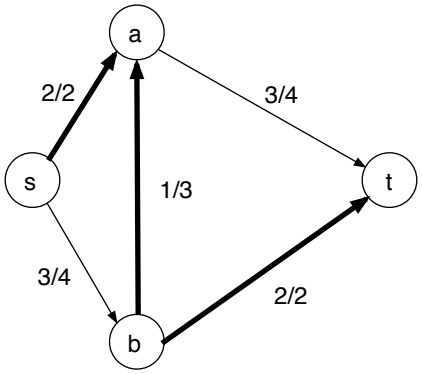

Figure 3: The bolded edge indicate which pipe has their flow adjusted.

It is now easy to see that we can push 1 unit of water along $s \to b \to a \to t$ to obtain the maximum flow of 6. If we carefully study the above figure, what we have essentially done is to flow 2 unit of water along $s \to a$, and then **push back** 2 unit of water from $a \to b$, and finally redirect the pushed back flow along $b \to t$ to the sink. So the key to completing the algorithm is the idea of **pushing back flow** - if we have x units of water flowing in the pipe (u, v), then we can pretend there is a pipe (v, u) with capacity x when we are trying to find a path from s to t.

This is the concept of **residual graph**. The residual graph of a network flow is essentially the network graph except that for every edge (u, v) that currently carries x unit of water, there is an edge (v, u) with capacity x in the residual graph. The following figure shows the residual graph after finding our first path:

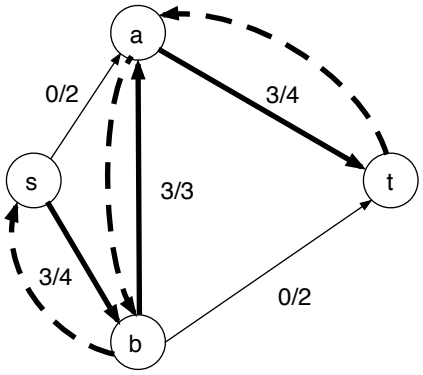

Figure 4: The dashed edges are the edges we added in the the residual graph. The capacity of the dashed edge is the same as the amount of water carried by the solid edge in the opposite direction.

So here is an algorithm to find the maximum flow: First construct the residual graph (in the beginning, the residual graph is the same as the network graph). While there exists a path from s to t in the residual graph, we flow water through one of the path (a path in the a residual graph is called an **augmenting path**). We then adjust the residual graph accordingly. Once we can no longer find any more path in the residual graph, we have found the maximum flow. This algorithm known as the **Ford Fulkerson Method** and its correctness is dependent on the following theorem, which we shall not prove:

Theorem 2.1. *If the residual graph contains no more augmenting path, then we have found the maximum flow.*

2.1 Time Complexity and Edmonds-Karp Algorithm

To be technical, the Ford Fulkerson Method is not an algorithm since it does not specify how to find the augmenting paths. For example, we can use either BFS or DFS to find the augmenting paths. In fact, without further knowledge on how to find the augmenting path, the best bound we have on the time complexity is $O(E * f)$, where E is the number of edges in the graph and f is the maximum flow. This is based on the observation that it takes $O(E)$ time to find an augmenting path and every augmenting path increases the flow by at least 1. Finally, it should be noted that the Ford Fulkerson Method does not guarantee that it will terminate - there are some special cases involving irrational capacity where we will keep finding augmenting path with smaller capacities. However, if it does terminate (as it will in the case of integer capacities), it will return the correct answer.

If we use BFS to find the augmenting path, then it is known as The **Edmonds-Karp Algorithm**. In this case, we can actually prove that the algorithm will terminate in $O(VE^2)$ time, a much tighter bound than what we have before.

3 Implementation of Edmonds-Karp

```
int capacity[128][128];    // adjacency matrix for the residual graph
bool seen[128];            // seen array for BFS
int parent[128];           // for tracing the augmenting path
int n;                     // number of vertices
int SRC = n + 1;           // the source vertex
int SNK = n + 2;           // the sink vertex

// return true iff an augmenting path exists in the residual graph
bool find_augmenting_path() {
  memset(parent, -1, sizeof(parent));
  memset(seen, 0, sizeof(seen));

  queue<int> q;
  q.push(SRC);
  seen[SRC] = true;
  while (!q.empty()) {
    int cur = q.front(); q.pop();
    if (cur == SNK) return true;

    for (int i = 0; i <= SNK; ++i) {
      if (!seen[i] && capacity[cur][i]) {
        seen[i] = true;
        parent[i] = cur;
        q.push(i);
      }
    }
  }
  return false;     // did not find a path to the sink
}

// find how much flow we can push along the augmenting path
int trace_augmenting_path() {
  int cur = SNK;
  int prev = parent[SNK];
  int flow = cap[prev][cur];
  while (prev != -1) {
    flow = min(flow, cap[prev][cur]);
    cur = prev;
    prev = parent[cur];
  }
  return flow;
}
```

```
// adjust the residual graph after pushing f flow via the augmenting path
void adjust_capacity(int f) {
  int cur = SNK;
  int prev = parent[SNK];
  while (prev != -1) {
    capacity[prev][cur] -= f;
    capacity[cur][prev] += f;
    cur = prev;
    prev = parent[cur];
  }
}

// the main function to solve max flow
int max_flow() {
  int maxflow = 0;
  while (find_augmenting_path()) {
    int flow = trace_augmenting_path();
    maxflow += flow;
    adjust_capacity(flow);
  }
  return maxflow;
}
```

4 Ford Fulkerson with DFS

We can also use DFS to find augmenting path in the Ford Fulkerson method. While there are not guaranteed improvement on the time complexity, it does make the code a lot shorter since we can keep track of how much flow to push in the augmenting path and adjust the residual graph in the same recursive call. Implementation as follow (uses the same variable as above):

```
int find_augmenting_path(int cur, int curflow) {
  if (cur == SNK) return curflow;
  seen[cur] = true;

  for (int i = 0; i <= SNK; ++i) {
    if (!seen[i] && capacity[cur][i]) {
      // find how much flow we can push through in augmenting path
      int amt = min(curflow, capacity[cur][i]);

      // recursively try to find the augmenting path
      int flow = find_augmenting_path(i, amt);

      if (flow > 0) {
        // found augmenting path
        capacity[cur][i] -= flow;
        capacity[i][cur] += flow;
        return flow;
      }
    }
  }
  return 0;
}

int max_flow() {
  int maxflow = 0, flow = 0;
  do {
    memset(seen, 0, sizeof(seen));
    flow = find_augmenting_path(SRC, INF);
    maxflow += flow;
  } while (flow > 0);
  return maxflow;
}
```