Pragmatic Programming

Session 9 - Flows and Cuts

Max Nilsson

January 25th

Flows and Cuts

- The setting is a directed graph `int g[mxn][mxn]` with associated *capacities* `int cap[mxn][mxn]` with a source node `0` and a sink node `n`.

Flows and Cuts

- The setting is a directed graph `int g[mxn][mxn]` with associated *capacities* `int cap[mxn][mxn]` with a source node `0` and a sink node `n`.
- In the **min-cut problem** we want to find the subset S of the graph, such that $0 \in S$ and $n \notin S$, which minimizes the outgoing capacities of S. Let this value be D.

Flows and Cuts

- The setting is a directed graph `int g[mxn][mxn]` with associated *capacities* `int cap[mxn][mxn]` with a source node `0` and a sink node `n`.
- In the **min-cut problem** we want to find the subset S of the graph, such that $0 \in S$ and $n \notin S$, which minimizes the outgoing capacities of S. Let this value be D.
- In the **max-flow problem** we want to find a flow matrix `int flow[mxn][mxn]` which is both feasible (the flow of each edge is non-negative and not greater than its capacity, and the inflow of each non-sink/source node is equal to its outflow) and maximizes the outflow of the source node. Let this value be P.

Flows and Cuts

- The setting is a directed graph `int g[mxn][mxn]` with associated *capacities* `int cap[mxn][mxn]` with a source node `0` and a sink node `n`.
- In the **min-cut problem** we want to find the subset S of the graph, such that $0 \in S$ and $n \notin S$, which minimizes the outgoing capacities of S. Let this value be D.
- In the **max-flow problem** we want to find a flow matrix `int flow[mxn][mxn]` which is both feasible (the flow of each edge is non-negative and not greater than its capacity, and the inflow of each non-sink/source node is equal to its outflow) and maximizes the outflow of the source node. Let this value be P.
- The max-flow problem is a linear program, which is unbounded (also easy to verify that Slater's condition holds) and so by strong duality we have that $P = P^*$.

Flows and Cuts

- The setting is a directed graph `int g[mxn][mxn]` with associated *capacities* `int cap[mxn][mxn]` with a source node `0` and a sink node `n`.
- In the **min-cut problem** we want to find the subset S of the graph, such that $0 \in S$ and $n \notin S$, which minimizes the outgoing capacities of S. Let this value be D.
- In the **max-flow problem** we want to find a flow matrix `int flow[mxn][mxn]` which is both feasible (the flow of each edge is non-negative and not greater than its capacity, and the inflow of each non-sink/source node is equal to its outflow) and maximizes the outflow of the source node. Let this value be P.
- The max-flow problem is a linear program, which is unbounded (also easy to verify that Slater's condition holds) and so by strong duality we have that $P = P^*$.
- Furthermore, it holds that $P^* = D$, even though the dual problem of max-flow is not technically equal to the min-cut problem!

The Max-Flow Min-Cut Theorem

- It should be utterly obvious (as Peter D. Lax puts it in his Functional Analysis book) that $P \leq D$. We will now sketch why $P = D$. Given a max-flow, we can with this algorithmic construct the subset S yielding the minimum cut.

The Max-Flow Min-Cut Theorem

- It should be utterly obvious (as Peter D. Lax puts it in his Functional Analysis book) that $P \leq D$. We will now sketch why $P = D$. Given a max-flow, we can with this algorithmic construct the subset S yielding the minimum cut.
- Initialize $S = \{0\}$. While there exists $u \in S$ and $v \notin S$ such that `g[u][v] == 1` and either `flow[u][v] < cap[u][v]` or `flow[v][u] > 0`, then include v in S.

The Max-Flow Min-Cut Theorem

- It should be utterly obvious (as Peter D. Lax puts it in his Functional Analysis book) that $P \leq D$. We will now sketch why $P = D$. Given a max-flow, we can with this algorithmic construct the subset S yielding the minimum cut.
- Initialize $S = \{0\}$. While there exists $u \in S$ and $v \notin S$ such that `g[u][v] == 1` and either `flow[u][v] < cap[u][v]` or `flow[v][u] > 0`, then include v in S.
- We can now verify that the cut of S equals

$$\begin{aligned}
\text{cut}(S) &:= \sum_{i \in S, j \notin S} \texttt{cap[i][j]} \\
&= \sum_{i \in S, j \notin S} \underbrace{\texttt{flow[i][j]}}_{=\texttt{cap[i][j]}} - \underbrace{\texttt{flow[j][i]}}_{=0} \\
&= \text{flow}(0)
\end{aligned}$$

The Max-Flow Min-Cut Theorem

- It should be utterly obvious (as Peter D. Lax puts it in his Functional Analysis book) that $P \leq D$. We will now sketch why $P = D$. Given a max-flow, we can with this algorithmic construct the subset S yielding the minimum cut.
- Initialize $S = \{0\}$. While there exists $u \in S$ and $v \notin S$ such that `g[u][v] == 1` and either `flow[u][v] < cap[u][v]` or `flow[v][u] > 0`, then include v in S.
- We can now verify that the cut of S equals

$$\begin{aligned}\text{cut}(S) &:= \sum_{i \in S, j \notin S} \texttt{cap[i][j]} \\ &= \sum_{i \in S, j \notin S} \underbrace{\texttt{flow[i][j]}}_{=\texttt{cap[i][j]}} - \underbrace{\texttt{flow[j][i]}}_{=0} \\ &= \text{flow}(0)\end{aligned}$$

- What remains to show is that the algorithm actually yields a cut, i.e. that $0 \in S$ and $n \notin S$.

The Max-Flow Min-Cut Theorem

- It should be utterly obvious (as Peter D. Lax puts it in his Functional Analysis book) that $P \leq D$. We will now sketch why $P = D$. Given a max-flow, we can with this algorithmic construct the subset S yielding the minimum cut.
- Initialize $S = \{0\}$. While there exists $u \in S$ and $v \notin S$ such that `g[u][v] == 1` and either `flow[u][v] < cap[u][v]` or `flow[v][u] > 0`, then include v in S.
- We can now verify that the cut of S equals

$$\begin{aligned}\text{cut}(S) &:= \sum_{i \in S, j \notin S} \texttt{cap[i][j]} \\ &= \sum_{i \in S, j \notin S} \underbrace{\texttt{flow[i][j]}}_{=\texttt{cap[i][j]}} - \underbrace{\texttt{flow[j][i]}}_{=0} \\ &= \text{flow}(0)\end{aligned}$$

- What remains to show is that the algorithm actually yields a cut, i.e. that $0 \in S$ and $n \notin S$.
- Question: Can you construct efficiently the max-flow from the min-cut?

How to Compute the Max-Flow

- The standard algorithm to use is the **preflow-push algorithm**.

How to Compute the Max-Flow

- The standard algorithm to use is the **preflow-push algorithm**.
- The algorithm keeps track a preflow at each node, which it constantly tries to push out to neighbouring nodes with smaller height. If this is not possible, it increases the height of the node. The initialization is that the source node pushes out a maximum saturating push and increases its fixed height to n.

How to Compute the Max-Flow

- The standard algorithm to use is the **preflow-push algorithm**.
- The algorithm keeps track a preflow at each node, which it constantly tries to push out to neighbouring nodes with smaller height. If this is not possible, it increases the height of the node. The initialization is that the source node pushes out a maximum saturating push and increases its fixed height to n.
- I have implemented this algorithm, with parallelization, in `C`, `C++`, `Java`, `Rust` and `Clojure`. Usually I take a shower after thinking about this time in my life.

How to Compute the Max-Flow

- The standard algorithm to use is the **preflow-push algorithm**.
- The algorithm keeps track a preflow at each node, which it constantly tries to push out to neighbouring nodes with smaller height. If this is not possible, it increases the height of the node. The initialization is that the source node pushes out a maximum saturating push and increases its fixed height to n.
- I have implemented this algorithm, with parallelization, in `C`, `C++`, `Java`, `Rust` and `Clojure`. Usually I take a shower after thinking about this time in my life.
- The time complexity is $\mathcal{O}(n^3)$ for a graph that is complete.

The Min-Cost-Max-Flow Problem

- Now we add edge weights `int weight[mxn][mxn]` and we generalize the max-flow problem to finding the maximum flow that minimizes

$$\sum_{i,j} \texttt{flow[i][j]} \cdot \texttt{weight[i][j]}.$$

The Min-Cost-Max-Flow Problem

- Now we add edge weights `int weight[mxn][mxn]` and we generalize the max-flow problem to finding the maximum flow that minimizes

$$\sum_{i,j} \texttt{flow[i][j]} \cdot \texttt{weight[i][j]}.$$

- Setting all the weights to 1 gives the standard max-flow problem.

The Min-Cost-Max-Flow Problem

- Now we add edge weights `int weight[mxn][mxn]` and we generalize the max-flow problem to finding the maximum flow that minimizes

$$\sum_{i,j} \texttt{flow[i][j]} \cdot \texttt{weight[i][j]}.$$

- Setting all the weights to 1 gives the standard max-flow problem.
- Suppose first that all weights are non-negative. There is seemingly no easy way to extend the preflow-push algorithm to this framework. But the (often) less efficient Edmonds-Karp algorithm (with at best $\mathcal{O}(n^4)$ complexity for the max-flow problem) can easily be extended.

The Min-Cost-Max-Flow Problem

- Now we add edge weights `int weight[mxn][mxn]` and we generalize the max-flow problem to finding the maximum flow that minimizes

$$\sum_{i,j} \texttt{flow[i][j]} \cdot \texttt{weight[i][j]}.$$

- Setting all the weights to 1 gives the standard max-flow problem.
- Suppose first that all weights are non-negative. There is seemingly no easy way to extend the preflow-push algorithm to this framework. But the (often) less efficient Edmonds-Karp algorithm (with at best $\mathcal{O}(n^4)$ complexity for the max-flow problem) can easily be extended.
- Furthermore, with some technical details, one could also handle negative weights and non-symmetric capacities (`cap[i][j]` $\neq$ `cap[j][i]`) in $\mathcal{O}(n^4)$ complexity.

Application into Matching

- A **Matching** of a graph G is a set of edges such that no two edges share a node. We want to find a perfect matching, i.e. a matching such that each node is an endpoint of an edge of the matching. If a perfect matching does not exist (for instance if n is odd) then we want to find a **maximum cardinality matching**.

Application into Matching

- A **Matching** of a graph G is a set of edges such that no two edges share a node. We want to find a perfect matching, i.e. a matching such that each node is an endpoint of an edge of the matching. If a perfect matching does not exist (for instance if n is odd) then we want to find a **maximum cardinality matching**.
- If the graph is bipartite (a graph whose nodes that can be divided into disjoint sets, such that each edge goes from one set to another) then we can connect the graph with a source and a sink and solve it as a max-flow problem in $\mathcal{O}(n^3)$.

Application into Matching

- A **Matching** of a graph G is a set of edges such that no two edges share a node. We want to find a perfect matching, i.e. a matching such that each node is an endpoint of an edge of the matching. If a perfect matching does not exist (for instance if n is odd) then we want to find a **maximum cardinality matching**.
- If the graph is bipartite (a graph whose nodes that can be divided into disjoint sets, such that each edge goes from one set to another) then we can connect the graph with a source and a sink and solve it as a max-flow problem in $\mathcal{O}(n^3)$.
- If the edges has an associated weight, we can solve the **assignment problem** (a maximum cardinality matching with the lowest total weight) as a min-cost-max-flow problem in $\mathcal{O}(n^4)$.

Application into Matching

- A **Matching** of a graph G is a set of edges such that no two edges share a node. We want to find a perfect matching, i.e. a matching such that each node is an endpoint of an edge of the matching. If a perfect matching does not exist (for instance if n is odd) then we want to find a **maximum cardinality matching**.
- If the graph is bipartite (a graph whose nodes that can be divided into disjoint sets, such that each edge goes from one set to another) then we can connect the graph with a source and a sink and solve it as a max-flow problem in $\mathcal{O}(n^3)$.
- If the edges has an associated weight, we can solve the **assignment problem** (a maximum cardinality matching with the lowest total weight) as a min-cost-max-flow problem in $\mathcal{O}(n^4)$.
- **Question:** Can we utilize the bipartite structure in a better way? **Yes**, in the unweighted case there is *the Hopcroft–Karp algorithm* in $\mathcal{O}(n^{2.5})$ and in the weighted case there is *the Hungarian algorithm* in $\mathcal{O}(n^3)$.

Application into Matching

- A **Matching** of a graph G is a set of edges such that no two edges share a node. We want to find a perfect matching, i.e. a matching such that each node is an endpoint of an edge of the matching. If a perfect matching does not exist (for instance if n is odd) then we want to find a **maximum cardinality matching**.
- If the graph is bipartite (a graph whose nodes that can be divided into disjoint sets, such that each edge goes from one set to another) then we can connect the graph with a source and a sink and solve it as a max-flow problem in $\mathcal{O}(n^3)$.
- If the edges has an associated weight, we can solve the **assignment problem** (a maximum cardinality matching with the lowest total weight) as a min-cost-max-flow problem in $\mathcal{O}(n^4)$.
- **Question:** Can we utilize the bipartite structure in a better way? **Yes**, in the unweighted case there is *the Hopcroft–Karp algorithm* in $\mathcal{O}(n^{2.5})$ and in the weighted case there is *the Hungarian algorithm* in $\mathcal{O}(n^3)$.
- **Question:** What happens if the graph is not bipartite? We can still solve the weighted problem in $\mathcal{O}(n^4)$ using *the Edmonds' blossom algorithm*.

This Week

- Skim read chapter 13 in Sannemo[1] and section 12.3 in Laaksonen[2].

[1] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

[2] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.

This Week

- Skim read chapter 13 in Sannemo[1] and section 12.3 in Laaksonen[2].
- Solve half of this weeks 9 problems (truncated).

[1] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

[2] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.