CMPSCI 311: Introduction to Algorithms

Lecture 17: Network Flows II

Akshay Krishnamurthy

University of Massachusetts

Last Compiled: April 4, 2018

Defining Flows

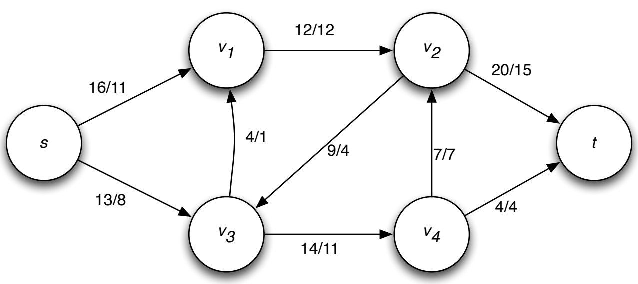
- Flow network
  - Directed graph
  - Source node s and target node t
  - Edge capacities $c(e) \geq 0$
- Flow
  - Capacity Constraints: $0 \leq f(e) \leq c(e)$ on each edge
  - Conservation Constraints:

$$f^{in}(s) = 0 \ , \quad f^{out}(t) = 0 \ , \quad \forall v \in V \setminus \{s,t\} \ f^{in}(v) = f^{out}(v)$$

where $f^{in}(v) = \sum_{e \text{ in to } v} f(e)$ and $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$

- Max flow problem: find a flow of maximum value $v(f) = f^{out}(s)$

Capacity/Flow



Residual Graph

Residual graph: data structure to identify opportunities to push more flow on edges with leftover capacity or undo flow on edges already carrying flow.

Original edge $e = (u, v) \in E$

- Flow $f(e)$
- Capacity $c(e)$

Forward residual edge

- $e = (u, v)$
- *residual capacity* $c(e) - f(e)$

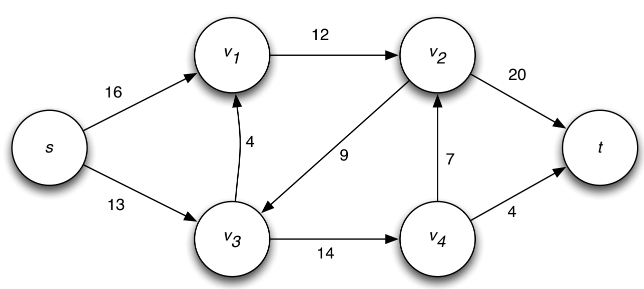
Backward residual edge

- if $f(e) > 0$, create edge $e' = (v, u)$
- *residual capacity* $f(e)$

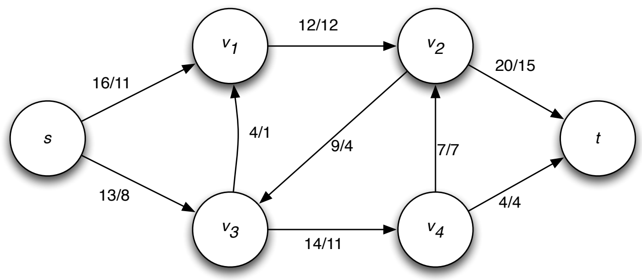
Residual Graph

Residual graph G_f with respect to flow f = graph of all forward and backward residual edges with positive residual capacity.

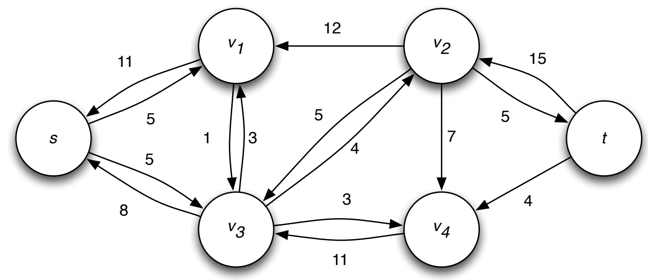
Capacity



Capacity/Flow



Residual Graph



Augmenting Path

Revised Idea: use paths in the *residual* graph to augment flow

Augment(f, P)
  Let $b =$ bottleneck(P, f) ▷ least residual capacity in P
  for edge $e = (u, v)$ in P **do**
    if e is a forward edge **then**
      $f(e) = f(e) + b$ ▷ increase flow on forward edges
    else
      $f(e) = f(e) - b$ ▷ decrease flow on backward edges
    end if
  end for

Ford-Fulkerson Algorithm

Repeatedly find augmenting paths in the residual graph and use them to augment flow!

Ford-Fulkerson(G, s, t)
  ▷ Initially, no flow
  Initialize $f(e) = 0$ for all edges e
  Initialize $G_f = G$
  ▷ Augment flow as long as it is possible
  while there exists an s-t path P in G_f **do**
    $f =$ Augment(f, P)
    update G_f
  end while
  return f

Ford-Fulkerson Analysis

- Step 1: argue that F-F returns a flow
- Step 2: analyze termination and running time
- Step 3: argue that F-F returns a maximum flow
- We did steps 1 and 2 last time, so just need to consider step 3.

Step 3: F-F returns a maximum flow

We will prove this by establishing a deep connection between flows and cuts in graphs: the max-flow min-cut theorem.

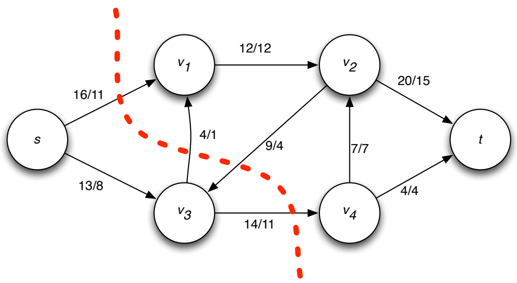
- An s-t cut (A, B) is a partition of the nodes into sets A and B where $s \in A$, $t \in B$
- Capacity of cut (A, B) equals

$$c(A, B) = \sum_{e \text{ from } A \text{ to } B} c(e)$$

- Flow across a cut (A, B) equals

$$f(A, B) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Example of Cut



Capacity is 34 and flow across cut is 19.

Flow Value Lemma

First relationship between cuts and flows

Lemma: let f be any flow and (A, B) be any s-t cut. Then

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Basic idea of proof is to use conservation of flow: all the flow out of s must leave A eventually.

Corollary: Cuts and Flows

Really important corollary of flow-value lemma

Corollary: Let f be **any** s-t flow and let (A, B) be **any** s-t cut. Then $v(f) \leq c(A, B)$.

Proof:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= c(A, B) \end{aligned}$$

Implies that if there's a flow f^* and cut (A^*, B^*) with $v(f^*) = c(A^*, B^*)$, then f^* is a max flow and (A^*, B^*) is a min cut.

F-F returns a maximum flow

Theorem: The s-t flow f^* returned by F-F is a maximum flow.

- Since f^* is the final flow there are no residual paths in G_{f^*}.
- Let (A^*, B^*) be the s-t cut where A^* consists of all nodes reachable from s in the residual graph.
- By Lemma, we know:

$$v(f) = f(A^*, B^*) = \sum_{e \text{ out of} A^*} f(e) - \sum_{e \text{ into} A^*} f(e)$$

.

Max Flow-Min Cut Proof

- Any edge out of A^* must have $f(e) = c(e)$ otherwise there would be more nodes than just A^* that are reachable from s.
- Any edge into A^* must have $f(e) = 0$ otherwise there would be more nodes than just A^* that are reachable from s.
- Therefore

$$\begin{aligned} v(f) = f(A^*, B^*) &= \sum_{e \text{ out of} A^*} f(e) - \sum_{e \text{ into} A^*} f(e) \\ &= \sum_{e \text{ out of} A^*} c(e) \\ &= c(A^*, B^*) \end{aligned}$$

First Application of Network Flows: Bipartite Matching

- Given an undirected graph $G = (V, E)$, a subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- The maximum matching problem is to find the matching with the most edges.
- We'll design an efficient algorithm for maximum matching in a bipartite graph. Recall, a graph is bipartite if the nodes V can be partitioned into two sets $V = L \cup R$ such that all edges have one endpoint in L and one endpoint in R.

Formulating it as a a network flow problem

- Given an instance $G = (L \cup R, E)$ of maximum matching, create a directed graph with nodes $L \cup R \cup \{s, t\}$
- For each undirected edge $(i, j) \in E$, add a directed edge from $i \in L$ to $j \in R$ with capacity 1.
- Add an edge with capacity 1 from s to each of the nodes in L
- Add an edge with capacity 1 from each of the nodes in R to t.
- Claim: The size of the maximum matching in G equals the value of the maximum flow in G'

Proof of Claim

- Any matching in G has size at most the maximum flow in G':
  - Can easily extend a matching in G of size k into a flow in G' of value k
- Any flow in G' has size at most the maximum matching in G
  - Consider the maximum flow f in G'. We may assume $f(e)$ is integral for each e.
  - Consider set of edges from L to R that have $f(e) = 1$, this is a matching because each node in L and R has at most one unit of flow in or out respectively.

Second Application of Network Flows: Image Segmentation

- Using an expensive camera and appropriate lenses, you can get a "bokeh" effect on portrait photos in which the background is blurred and the foreground is in focus.

- But using cheap cameras in phones and appropriate software you can fake this effect...

Formulating the problem

- Input:
  - Let V be the set of pixels in the images and let E be pairs of neighboring pixels.
  - For each pixel i, you have a likelihood $f_i \geq 0$ that it is in the foreground and a likelihood $b_i \geq 0$ that it is in the background.
  - For each $(i, j) \in E$, let p_{ij} be a penalty you pay for labeling one as foreground and one as background.
- Goal: You want to partition V into foreground pixels F and background pixels B such that you maximize

$$score(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E: i \in F, j \in B} p_{ij}$$

- Observation: Define

$$score'(F, B) = \sum_{i \in V} f_i + \sum_{j \in V} b_j - score(F, B)$$

  - Maximizing $score(F, B)$ is same as minimizing $score'(F, B)$

Turning the problem into a network flow problem

- Define the directed graph G where
  - Pixels, V, are nodes of G
  - Between each pair of neighboring pixels i and j, add an edge in each direction with capacity p_{ij}.
  - Add node s with an edge to each pixel j with capacity f_i
  - Add node t with an edge from each pixel j with capacity b_i
- We can rewrite $score'(F, B)$ as:

$$\begin{aligned} score'(F, B) &= \sum_{i \in V} f_i + \sum_{j \in V} b_j - score(F, B) \\ &= \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E: i \in F, j \in B} p_{ij} \\ &= c(F, B) \end{aligned}$$

- So finding minimum cut in G is equivalent to maximizing the image segmentation score.