

15-451 Assignment 05

Karan Sikka

ksikka@cmu.edu

Collaborated with Dave Cummings and Sandeep Rao.

Recitation: A

October 10, 2014

1: Cell Towers

Profit = Actual Revenue - Cost

Profit = Total Possible Revenue - Lost Revenue - Cost

LR + Cost = TPR - Profit

Since the total possible revenue is some constant, minimizing the LHS of the above equation maximizes the profit.

So finding the min (lost revenue + cost) is the right thing to do. Now lets see how to do it.

In part (a) we will construct a graph.

In part (b) we will show that the finding the min cut correctly finds the cell towers to build which min (lost revenue + cost)

In part (c) we will give an algorithm to find the min cut.

In part (c) we will show that it's efficient.

(a) Construct a bipartite graph (nodes are partitioned into A and B with no edges within A or B) as follows:

There is a node in A for every revenue-generating pair of cell towers.

Add a source node.

Create edges from the source node to every node in A with capacity of the revenue generated by the pair of cell towers represented by the node in A.

For each node in A, for each cell tower in the pair represented by that node, connect that node to the cell tower in B.

Give that edge an infinite capacity.

There is a node in B for every cell tower.

Add a target node.

Create edges from every node in B to the target with capacity of the cost of building the tower represented by the node in B.

(b) Notice that the min cut will never contain an edge going from A to B, because those edges have infinite capacity.

Therefore the node representing a pair of cell towers in A will be in the same vertex set as the individual cell towers in B after the cut is performed.

If the min cut contains an edge from B to target, we say that we built that cell tower of the node in B, and incurred a financial cost of the amount of that capacity.

If the min cut contains an edge from source to A, we say that at least one of the cell towers in pair of the node in A will not be built, and we lost revenue of the amount of that capacity.

Min-cut will minimize the cost + the lost revenue

The edges of the min cut from B to target will tell you which cell towers to build.

(c) You can perform Edmond-Karp #1 until your last residual graph when the algorithm terminates. On the residual graph, find all vertices reachable from the source. This is one set in the min cut, the other set has the nodes not in the first set.

This is correct because the algorithm pushes max flow until some edges have residual capacity 0 and they disappear, disconnecting the graph. This max flow value is exactly equal to the min cut value. Therefore the vertices not reachable must have been separated by the edges of the min cut.

Find the edges between the sets, look for edges where one node is the target, and the other node in the edge is a cell tower you should build.

(d)

The number of nodes and edges in the graph is no more than polynomial in the number of cell towers and revenue pairs due to the way we constructed the graph.

The Edmond-Karp algorithm is polynomial in nodes and edges.

Therefore this algorithm runs in polynomial time and is “efficient”.

2: Eliminating Negative Edges

(a) Original graph had no negative cycles. We introduce a node s , create an edge from s to every other node, and give edge weight 0.

A new negative cycle would have to include s . However that can't be because there are no edges going into s . Therefore there are no negative cycles in the new graph.

(b)

Case the shortest path to v goes through u .

$$\Phi(v) = \text{shortest_path_from_u_to_v} + \Phi(u)$$

$$\text{and we know that } \text{shortest_path_from_u_to_v} \leq \text{len}(u, v)$$

$$\implies \Phi(v) \leq \text{len}(u, v) + \Phi(u) \implies \Phi(v) - \Phi(u) \leq \text{len}(u, v).$$

Case the shortest path to v does not go through u .

Then the shortest path from s to v is even shorter than the shortest path that goes through u , so $\Phi(v)$ is smaller than was shown in the previous case, so the original claim holds.

(c) Consider a path of nodes v_1, v_2, \dots, v_k . When you sum the new edge lengths, you see all of the Φ terms cancel out except

you are left with $\Phi(v_1) - \Phi(v_k)$. So the length of a path in the new graph is $\text{old_path_len} + \Phi(v_1) - \Phi(v_k)$

For each node, run djikstra's single source shortest path algorithm.

We claim that the resulting shortest paths will be the true shortest paths in the original graph.

This is because the shortest path from some u to some v is going to be the path among all paths from u to v minimizing $\text{old_path_len}(u, v) + \Phi(u) - \Phi(v)$ where $\Phi(u) - \Phi(v)$ is a constant quantity for a given u, v .

The real length of the path from u to v is $\text{new_path_len}(u, v) - \Phi(u) + \Phi(v)$

The cost of running djikstra's n times using the fibonacci heaps version is $O(nm + n^2 \lg(n))$

3: Color Me Red, Color Me Blue

For convenience, we will call a $(k, n - k)$ partition which minimizes split edges a “minimizing k-partition”.

We will consider a function `kPart` as follows:

Given a node n , value i ($1 \leq i \leq k$) and bit b ,
The function returns `NONE`, or `SOME (A, B, S)`.

`kPart(n, k, b) = NONE` signifies that no k-partition of vertices in the subtree rooted at n exists, which happens when $(k > |n|)$.

`kPart(n, k, b) = SOME(A, B, S)` signifies that A, B is some minimizing k-partition the nodes of the subtree rooted at n ,

such that $b = 1 \implies n \in A$ and $b = 0 \implies n \in B$ and

S is the set of split edges

We create a memo table for the computation which we assume is available to the function.

The table can be accessed by $T[n][k][b]$ where n is a node, k is an int from 0 to k , b is a bit.

We initialize the table with `NULLs`, and compute the values in the following recursive function.

The procedure to compute `kPart(r, k, b)` is as follows:

(Check if value has already been computed.)

If $T[r][k][i] \neq \text{NULL}$,

 return $T[r][k][i]$

(If $k=0$, A is empty, B is rest of nodes.)

If $k = 0$,

$T[r][k][i] = (\{\}, \text{all_nodes}(b), \{\})$

(If $k > n$, no set A can be formed...)

If $k > n$,

$T[r][k][i] = \text{DNE}$

If both $r \rightarrow \text{left}$ and $r \rightarrow \text{right}$ are `NULL`,

 If $k = 1$

$T[r][k][i] = (\{r\}, \{\}, \{\})$

 Else

$k > n$ and this case would have been caught earlier

If only $r \rightarrow \text{left}$ is `NULL`,

$T[r][k][i] =$ Take the answer (of the form (A, B, S)) with the minimum $|S|$ over all of the following cases, where each case represents a possible coloring of r and $r \rightarrow \text{right}$ that might yield a minimizing k-partition:

 Case 1: $r \in A, r \rightarrow \text{right} \in A \implies$

$A_r, B_r, S_r = \text{kPart}(r \rightarrow \text{right}, k - 1, 1)$

 answer is $(\{r\} \cup A_r, B_r, S_r)$

 Case 2: $r \in A, r \rightarrow \text{right} \in B \implies$

$A_r, B_r, S_r = \text{kPart}(r \rightarrow \text{right}, k - 1, 0)$

 answer is $(\{r\} \cup A_r, B_r, S_r \cup \{(r, r \rightarrow \text{right})\})$

 Case 3: $r \in B, r \rightarrow \text{right} \in A \implies$

$A_r, B_r, S_r = \text{kPart}(r \rightarrow \text{right}, k, 1)$

 answer is $(A_r, \{r\} \cup B_r, S_r \cup \{(r, r \rightarrow \text{right})\})$

 Case 4: $r \in B, r \rightarrow \text{right} \in B \implies$

$A_r, B_r, S_r = \text{kPart}(\text{r->right}, k, 0)$
 answer is $(A_r, \{r\} \cup B_r, S_r)$

If only r->right is NULL,

Logic is symmetric to above case.

If r->left and r->right are not NULL,

$T[r][k][i] =$ Take the answer (of the form (A, B, S)) with the minimum $|S|$ over all of the following cases, where each case represents a possible coloring of r , r->right , and r->left that might yield a minimizing k -partition:

(Note that in each case, the Left and Right subtrees can each be partitioned in at most $k+1$ ways such that the sum of their resulting A sets should be $k-1$ if r in A or k otherwise.)

Case 1: $r \in A, \text{r->left} \in A, \text{r->right} \in A$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k - 1$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 1)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 1)$
 answer is $(A_l \cup A_r \cup \{r\}, B_l \cup B_r, S_l \cup S_r)$

Case 2: $r \in A, \text{r->left} \in A, \text{r->right} \in B$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k - 1$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 1)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 0)$
 answer is $(A_l \cup A_r \cup \{r\}, B_l \cup B_r, S_l \cup S_r \cup \{(r, \text{r->right})\})$

Case 3: $r \in A, \text{r->left} \in B, \text{r->right} \in A$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k - 1$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 0)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 1)$
 answer is $(A_l \cup A_r \cup \{r\}, B_l \cup B_r, S_l \cup S_r \cup \{(r, \text{r->left})\})$

Case 4: $r \in A, \text{r->left} \in B, \text{r->right} \in B$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k - 1$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 0)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 0)$
 answer is $(A_l \cup A_r \cup \{r\}, B_l \cup B_r, S_l \cup S_r \cup \{(r, \text{r->left}), (r, \text{r->right})\})$

Case 5: $r \in B, \text{r->left} \in A, \text{r->right} \in A$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 1)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 1)$
 answer is $(A_l \cup A_r, B_l \cup B_r \cup \{r\}, S_l \cup S_r \cup \{(r, \text{r->left}), (r, \text{r->right})\})$

Case 6: $r \in B, \text{r->left} \in A, \text{r->right} \in B$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k$ and take the one with minimum $|S|$:

$A_l, B_l, S_l = \text{kPart}(\text{r->left}, i, 1)$
 $A_r, B_r, S_r = \text{kPart}(\text{r->right}, j, 0)$
 answer is $(A_l \cup A_r, B_l \cup B_r \cup \{r\}, S_l \cup S_r \cup \{(r, \text{r->left})\})$

Case 7: $r \in B, \text{r->left} \in B, \text{r->right} \in A$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i + j = k$

and take the one with minimum $|S|$:

$$A_l, B_l, S_l = \text{kPart}(\mathbf{r} \rightarrow \text{left}, i, 0)$$

$$A_r, B_r, S_r = \text{kPart}(\mathbf{r} \rightarrow \text{right}, j, 1)$$

answer is $(A_l \cup A_r, B_l \cup B_r \cup \{r\}, S_l \cup S_r \cup \{(r, \mathbf{r} \rightarrow \text{right})\})$

Case 8: $r \in B, \mathbf{r} \rightarrow \text{left} \in B, \mathbf{r} \rightarrow \text{right} \in B$

Compute the following answers over all integral i, j s.t. $i, j \geq 0$ and $i+j = k$

and take the one with minimum $|S|$:

$$A_l, B_l, S_l = \text{kPart}(\mathbf{r} \rightarrow \text{left}, i, 0)$$

$$A_r, B_r, S_r = \text{kPart}(\mathbf{r} \rightarrow \text{right}, j, 0)$$

answer is $(A_l \cup A_r, B_l \cup B_r \cup \{r\}, S_l \cup S_r \cup \{\})$

return $T[r][k][i]$

Correctness

The proof of correctness is by structural induction.

The algorithm correctly returns (A, B, S) of the minimizing k-partition of a leaf node.

The inductive cases consider all possible colorings of the root, the left child, and right child, compute the minimizing k-partition in each case, and takes the answer with the min $|S|$ of all of those.

Note that there are no edges between nodes in the left subtree and nodes in the right subtree due to the definition of a tree.

Therefore there are no other partitions to consider which can be a minimizing k-partition.

Runtime analysis

We're going to put coins on nodes/root of subtrees.

A function call spends the coins on the node/root of subtree to do work (like take a min), to perform recursive calls that have been memoized, but not to perform unmemoized recursive calls. The claim is that recursive calls will pay for themselves using the money on their root.

For a given node n and k value, the function will spend at most $8(k+1)$ (which is less than $8(n+1)$) coins at most twice (once when b is 1, once when b is 0).

After spending this money, the function is memoized and some other function call pays for it.

There are at most $n+1$ possible values of k , so the money diminished from that node is at most $(n+1) * 2 * 8 * (n+1)$ coins total.

There are n nodes so the total number of coins you need to fund the operation is at most $n * (n+1) * 2 * 8 * (n+1) = O(n^3)$