CS361B - Advanced Algorithms

Spring 2014

Contents

1 A scaling algorithm for the Shortest Path Problem

Scaling is an iterative technique where we start by solving the problem using very coarse rounding of the input data (e.g. look only at the most significant bit) and at each phase we refine our solution to take into account more bits in the input.

1.1 Definitions and Descriptions

Shortest Path Problem Given a Directed Graph $G = (V, E)$ with a weight (or cost or length) function $c : E \to R$ on the edges and a special s (the "source"), the goal is to find the shortest path from s to every node in G with respect to c.

Dijkstra's shortest path algorithm can be implemented to run in $O(m + n \log n)$ time using Fibonacci heaps.

Min-Cost Flow Problem Given a directed graph $G(V, E)$ with non-negative capacity $\text{cap}(uv)$, arbitrary cost $c(uv)$ on each edge uv, and special nodes s ("source") and t ("sink"), the goal is to find max flow from s to t of minimum cost. The cost of a flow f is defined to be $\sum_{e \in E} c(e)f(e)$.

Relationship between Shortest Path and Min Cost Flow problems Observe that shortest path problem can be viewed as a trivial case of min-cost flow problem. To find a shortest path from s to t, we can solve a min-cost flow problem,requiring the min-cost flow of value 1, where the capacities are 1 on each edge and where the cost of each edge is equal to its length. (How would you use single min-cost flow computation to find distances from s to *all* the nodes ?)

If your "black box" min-cost flow solver produces fractional solutions, then one can decompose the resulting flow into paths and use any of these paths. Observe that length of each one of the paths should be the same since otherwise the black box has not produced a min-cost flow solution.

1.2 Scaling implementation of Dijkstra's algorithm

In the following discussion we will assume that the reader is familiar with the standard heap-based implementation of Dijkstra's shortest paths algorithm.

In order to implement Dijkstra's algorithm, we need a data structure that supports the following operations:

- Insert node with a specific key
- Decrease key of a specific node
- Extract a node with smallest key

The key used in the above operations is our current upper bound on the distance of the node from s. Initially the distance is infinity, and each time we relax a node, we update the upper bounds on distances for all of its neighbors.

We will start by considering the case where all the costs are either 0 or 1. In this case, n can serve as infinity, and all distances are between 0 and $n-1$. We will implement the data structure as an array of $n+1$ cells, with cell k pointing to a linked list of nodes that have current bound $d(u)$ on distance equal to k.

Initially, source is linked to cell 0, while all other nodes to cell n. Each time we reduce a key of some node, this node "moves left", to a linked list of nodes that have smaller key values.

We also assume that there is an underlying data structure defining the graph. In particular, this data structure allows us to list all the neighbors of a node in constant time per node and find these neighbors in the above mentioned array.

Dijkstra's algorithm works by extracting node v with the smallest $d(v)$, current upper bound on distance, assigning this upper bound as the real distance to this node, and updating upper bound for every neighbor u of v to $\min(d(u), d(v) + c(vu))$. This operation is called "relaxation of node v".

Following observations can be made:

- When relaxing node at distance d, i.e.this node was fetched from the linked list attached to cell d of the array, all the neighbors of this node can move left up to cell d, but cannot move to cells left of d. This is due to the fact that distances are non-negative.
- When relaxing node at distance d, cells with index smaller than d are empty.
- Nodes always move left, never right.

From these observations, it is easy to see that the total amount of time spent on dealing with the data structure is bounded by $O(m)$ for time we spent on relaxing each edge plus $O(n)$ time we spent moving right in the array, looking for the next non-empty cell. Thus, the total running time of Dijkstra's shortest path algorithm will be $O(m)$ in this case.

Now consider the case where the lengths are just non-negative integers and we are guaranteed that maximum shortest path is limited by $n-1$. Observe that the above implementation works in this case without any modifications with $O(m)$ running time.

Before we can use the above implementation to solve the shortest path problem efficiently, we need to introduce the concept of Reduced Costs.

Reduced Cost For every edge uv, we transform the cost of this edge as follows:

$$c_p(uv) = p(u) - p(v) + c(uv)$$

Where p is an arbitrary function from $V \to R$.

Lemma 1.1. *The shortest path computed in the reduced cost graph is the same as the shortest path in the original graph. More generally the ranking of source-destination paths by cost is preserved by the reduced-cost transformation.*

Proof: Consider any path from s to some node t. Compute the sum of the costs along the edges of the path. Note that all the p's except $p(s)$ and $p(t)$ cancel out. A constant term of $p(s) - p(t)$ is added to the weight of each such path. Thus the ranking of source-destination paths by cost is preserved by the reduced-cost transformation. Specifically the shortest path is preserved. ▮

Scaling Shortest Path Algorithm

Consider binary representation of weights and let $c^i(uv)$ be the weight resulting from considering i most significant bits of the original weight $c(uv)$. Clearly, $c^0(uv) = 0$ and $c^{\lceil \log C \rceil}(uv) = w(uv)$, where C is the maximum weight of an edge. Let $d^i(u)$ be distance from s to u with respect to c^i, i.e. only i most significant bits of the costs.

Initialize $p(u) = 0$ for all nodes $u \in V$ and $i = 0$. This implies $c_p^0(uv) = p(u) - p(v) + c^0(uv) = 0$ for all edges $uv \in E$. Also, $d^0(u) = 0$ for all u.

To go from i to $i + 1$:

1. Set potentials $p(u) = d^i(u)$ and compute reduced costs $c_p^i(uv)$ with respect to these potentials.
2. Using definition of $d^i(u)$, we have $c_p^i(uv) = p(u) + c^i(uv) - p(v) = d^i(u) + c^i(uv) - d^i(v) \geq 0$. Moreover, observe that if edge uv is on a shortest path (from s to *any* node), $d^i(v) = d^i(u) + c^i(uv)$, implying that $c_p^i(uv) = 0$.
3. Set $p(v) = 2p(v)$ for all v, i.e. double the values of $p(v)$. Now consider $c_p^{i+1}(uv)$. Since $c^{i+1}(uv) = 2c^i(uv)$ or $2c^i(uv) + 1$, we get that:
   (a) $c_p^{i+1}(uv) \geq 0$ for all uv.
   (b) if uv is on some shortest path, $c_p^{i+1}(uv)$ is either 0 or 1.

   This implies that the lengths of paths that were shortest with respect to c^i are at most $n - 1$ with respect to c_p^{i+1}. Thus, length of any shortest path with respect to c_p^{i+1} is at most $n - 1$.
4. Use modified Dijkstra as explained above to find shortest paths from s to all nodes with respect to c_p^{i+1}. Note that these paths are also shortest with respect to non-reduced costs c^{i+1}.
5. Compute $d^{i+1}(u)$ for all nodes.

Analysis of running time

Each instance of Dijkstra's algorithm runs in time $O(m)$ as discussed in the previous section. Assuming the costs are integer with C_{max} denoting the largest absolute value of the cost, there are $\log C_{max}$ iterations and the total time is bounded by $O(m \log C_{max})$. Observe that single iteration still runs in $O(m)$ time even if the bound on longest distance is m and not $n - 1$. This allows us to change the scaling parameter from 2 to m/n, more precisely closest power of 2 to this ratio. This improves the running time to $O(m \log_{m/n} C)$. In particular, for dense graphs where $m = \Omega(n^2)$, this leads to $\log n$ improvement in the running time. See [3] for more discussion of scaling algorithms.

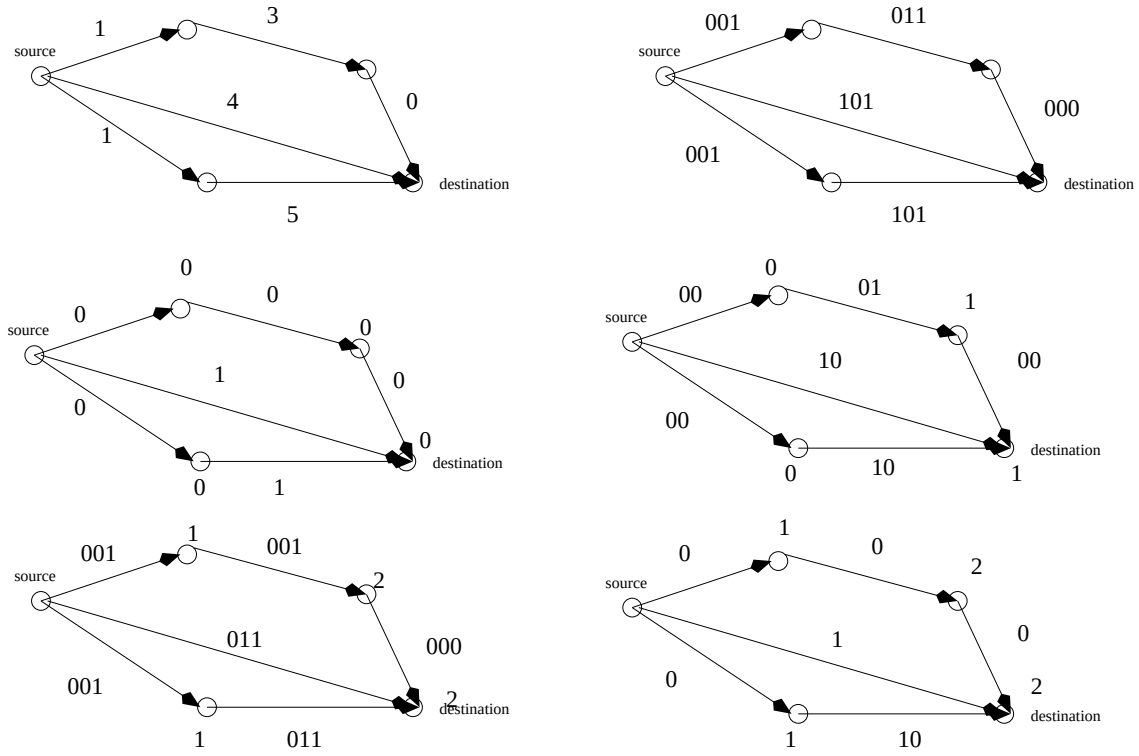


Figure 1: *Working of the Shortest Path Scaling Algorithm. The figure in the top-left is the graph for which the shortest path is being computed. The figure in the top-right is the same graph with the costs in binary. The figures in the mid-left,mid-right, bottom-left, show the inputs to the Dijkstra's algorithm for the first three iterations respectively. The vertices are labeled with the shortest path that is computed for them in the corresponding iteration. The figure in the bottom-right is state when the loop exits. Note that edges on the shortest-path to any vertex have 0 reduced cost.*

2 Min-cost Flow

In this section we reduce the min-cost flow problem to the min-cost circulation problem. We give two alternative characterizations of optimal circulation, both using the concept of residual graphs. The first characterization is based on a global property of the residual graph - no negative-cost cycles. The second is based on the local notions of node potentials and reduced costs of edges, and is derived from an LP formulation of the problem. Namely, a circulation is optimal iff there exist potentials such that the reduced costs on every edge of the residual graph for the circulation are non-negative.

2.1 Problem definition

Definition 2.1. Min-cost flow *is a max flow with minimum cost.*

This means that a solution must have the maximum possible flow, and of all possibilities with that flow, it has the minimum possible cost. Solving this problem is a good example of how considering duality can lead to insights into the solution of optimization problems. For this discussion it is assumed that we are only dealing with a cost that is per unit flow.

We need to optimize two different things: first the amount of flow, and then the cost given the amount of flow. To solve the problem, we will reduce the min-cost max flow problem to a min-cost circulation problem.

A circulation problem requires that every node obey flow conservation, meaning that each node has the same amount of flow entering as leaving. In a min-cost flow, on the other hand, the source, s, and the sink, t, are special nodes and do not obey flow conservation. That is, no flow enters s but flow leaves from s, and flow enters t but no flow leaves from t.

To figure out how to convert min-cost flow to a circulation problem, we'll need to first discuss residual graphs.

2.2 Residual graphs

Definition 2.2. *Given a graph G and a flow f, the* **residual graph** *G_f is a graph where for each edge uv in G, there are two edges in the residual graph:*

- *One edge is from u to v in G_f with capacity equal to the capacity of uv in G minus the flow through uv in G.*
- *The other edge is from v to u in G_f with capacity equal to the flow through uv in G.*

Any edge with capacity 0 in the residual graph is removed.

This means that the capacity of an edge in a residual graph, the residual capacity, is the amount of extra flow, in addition to the flow that has already been pushed over the edge, that we can now push before exceeding the original capacity.

[0]Section edited by Jeremy Pack

For example, say the edge uv has initial capacity 5 and cost 7. Now we push a flow of 2 along uv. In the residual graph, uv will have a capacity of $5 - 2 = 3$, which represents the possibility of pushing up to 3 more units of flow along uv. However, now we also have the possibility of removing the 2 units of flow we have already pushed. This is represented by creating an edge vu with capacity 2. Since pushing a unit of flow along vu is equivalent to removing a unit from uv, the cost for vu is -7. The residual graph does not include those edges with residual capacity of 0.

The residual graph is thus a representation of how much the flow can change in either direction along each edge. Now that we have defined residual graphs, we can define augmenting paths:

Definition 2.3. *An* **augmenting path** *is a path from s to t in a residual graph.*

2.3 Reduction to min-cost circulation

Definition 2.4. *A* **pseudoflow** *is a flow that satisfies capacity constraints but not necessarily conservation constraints.*

Definition 2.5. *A* **circulation** *is a pseudoflow that satisfies conservation constraints everywhere, that is: for every node, the flow into the node is equal to the flow out of the node.*

To reduce to a min-cost circulation problem, we first add an edge from t to s. For now, assume the edge's capacity is sufficiently large, and its cost is sufficiently negative to make any solution to the min-cost circulation problem the same as that for the min-cost flow problem (with the flow into s and out of t in the min-cost flow being represented along the edge between them in the min-cost circulation).

Lemma 2.6. *Given a directed graph, G, containing nodes s and t, the min-cost flow problem can be solved as follows. Add an edge ts to G, whose cost is sufficiently negative as to overcome the expense of the most expensive $s - t$ path in G, and whose capacity is sufficiently large as not to be saturated under the max flow. Then solve the min-cost circulation problem on the resulting graph.*

Proof: Let f be the flow obtained from the solution f_c to min-cost circulation by removing the edge ts. If f is not an optimal solution to min-cost flow, then either f is not a maximum flow, or f is a maximum flow with cost greater than the minimum cost.

Suppose that f is not a maximum flow. Then there is an augmenting path from s to t in G_f. Since the cost of the edge ts is sufficiently negative, the cycle obtained by combining the augmenting path with this edge has negative cost. Now the circulation f_c can be modified by sending flow around the cycle because the edges on the augmenting path are not saturated and the edge ts has a sufficiently large capacity. This produces a new circulation with a smaller cost than f_c, which is a contradiction.

Suppose that f is a maximum flow with cost greater than that of the min-cost flow f^*. Construct a new circulation f_c^* from f^* by sending the flow that goes from s to t back along the edge ts. Both f_c^* and f_c incur the same cost along the edge ts, since the amount of flow on this edge in both circulations is the value of a maximum flow. The circulation f_c^* incurs a lower cost on the other edges in the graph than f_c, however, and so f_c^* has a smaller cost than f_c. This is again a contradiction, so the solution to the min-cost circulation must also solve the min-cost flow. ∎

2.4 Negative cost cycles

Lemma 2.7. *A circulation f^* is a min-cost circulation iff there are no negative-cost cycles remaining in the residual graph, G_{f^*}.*

Proof:

Only if: A min-cost circulation cannot contain negative-cost cycles in its residual graph, because its cost could then be trivially lowered just by augmenting along one of those cycles.

If: Consider a circulation f', which is not optimal but for which there remain no negative-cost cycles in the residual graph. Consider also an optimum circulation f^*. Consider the pseudoflow $f^* - f'$. First, we know it must be a circulation. This is because f^* and f' are both circulations, and thus conservation constraints are satisfied everywhere. So, when we subtract f' from f^*, we remove the same amount of flow going in to each node as going out, and conservation constraints are still satisfied in the result.

Second, we know $f^* - f'$ must be feasible in $G_{f'}$. To see this, let $cap(uv)$ be the capacity of edge uv in G, and $cap_{f'}(uv)$ the capacity of edge uv in $G_{f'}$. Then for every edge uv for which $f'(uv) \leq f^*(uv)$ we have $f^*(uv) \leq cap(uv) \rightarrow f^*(uv) - f'(uv) \leq cap(uv) - f'(uv) = cap_f(uv)$. Thus $f^* - f'$ satisfies capacity constraints for this edge. A similar proof for the case $f^*(uv) < f'(uv)$ is left as an exercise. Finally, flow $f^* - f'$ must have negative cost, since $cost(f^*) = cost(f') + cost(f^* - f')$ and f' is not optimal. However, since one can decompose any circulation into $O(m)$ cycles (also left as an exercise), at least one negative-cost cycle must be used by $f^* - f'$ since it is a flow of negative cost. But this means that negative-cost cycles exist in $G_{f'}$, which contradicts our assumptions. ∎

Note that we can check for negative cost cycles using the Bellman-Ford algorithm.

2.5 LP Formulation

Min-cost circulation can be expressed as a Linear Program:

Primal:

$$\begin{array}{lrcll} \forall uv \in E: & f(uv) & \geq & 0 & \\ & f(uv) & \leq & cap(uv) & \text{(capacity constraint)} \\ \forall v \in V: & \sum f(vw) - \sum f(uv) & = & 0 & \text{(conservation constraints)} \end{array}$$

$$\text{maximize} - \sum cost(uv) f(uv)$$

Dual:

1. Each edge capacity constraint in the primal corresponds to a variable $\ell(uv)$ in the dual; since the capacity constraints are inequalities, we have the constraint $\ell(uv) \geq 0$.
2. Each node circulation constraint in the primal corresponds to a variable $p(v)$ in the dual; since the circulation constraints are equalities, the $p(v)$ are unrestricted.
3. Each variable in the primal corresponds to an inequality in the dual. The left-hand side of the inequality can be obtained by looking at the matrix used in the primal–see a full LP review for more on this–and the right-hand side comes from the primal's objective function. This yields the following inequalities: $\forall uv \in E : \ell(uv) + p(u) + cost(uv) - p(v) \geq 0$.
4. The coefficient of each variable in the dual's objective function comes from the right-hand side of the corresponding equation in the primal, leading to the objective function: minimize $\sum \ell(uv) cap(uv)$.

So the final equations are:

$$\begin{array}{lrcl} \forall uv \in E: & \ell(uv) & \geq & 0 \\ \forall uv \in E: & \ell(uv) + p(u) + cost(uv) - p(v) & \geq & 0 \\ \text{minimize} \sum \ell(uv) cap(uv) & & & \end{array} \tag{1}$$

Definition 2.8. *The* **reduced cost**, $C_p(uv) = p(u) + cost(uv) - p(v)$.

Definition 2.9. $p(u)$ *is the* **node potential** *of a node* u.

Intuition Regarding p: The potentials p act to "straighten up" the edges of the graph. Consider the example of a triangle that has edge weights -10, 20, and 0 between uv, vw, and wu respectively. We then create potentials $p(u) = 0$, $p(v) = -10$, and $p(w) = 0$. (You can check that these are in fact the distances to the added node s.) These give a reduced cost graph with edge weights 0, 10, and 0. The potentials work to redistribute the negative cost edges, in this case uv, among the higher cost edges, in this case vw.

From here we may also note that the potentials are translationally independent, but not independent subject to multiplicative scaling.

This idea of relating node potentials to distance is a common one that appears in a variety of contexts.

The equations (1) can be written in terms of the reduced cost instead of the node potentials as follows:

$$\begin{array}{lrcl} \forall uv \in E: & \ell(uv) & \geq & 0 \\ \forall uv \in E: & \ell(uv) + C_p(uv) & \geq & 0 \\ \text{minimize} \sum \ell(uv) cap(uv) & & & \end{array} \tag{2}$$

Now, we can see that, given the reduced cost $C_{p^*}(uv)$ for an edge uv corresponding to the values p^* of the variables p in an optimal solution to the dual problem, the optimum $\ell^*(uv)$ can be derived by breaking into two cases, $C_{p^*} < 0$ or $C_{p^*} \geq 0$. Suppose first that it is negative.

From (2), for any edge uv, if $C_{p^*}(uv)$ is negative, then the active constraint on $\ell(uv)$ is that

$$\ell(uv) \geq -C_{p^*}(uv)$$

Now, the value of $\ell(uv)$ that minimizes the objective fuction, while satisfying the above constraint is, in this case, is

$$\ell^*(uv) = -C_{p^*}(uv)$$

On the other hand, if $C_{p^*}(uv) \geq 0$, then the active constraint on $\ell(uv)$ is $\ell(uv) \geq 0$. In this case, the objective function is minimized when

$$\ell^*(uv) = 0$$

Thus, for each edge uv, the value of ℓ^* is just

$$\ell^*(uv) = \max(-C_p^*(uv), 0) \tag{3}$$

Next we will show useful relations between optimum values of the primal and the dual.

2.6 Reduced costs and node potentials

Lemma 2.10. *If the optimum reduced cost of an edge is negative then the edge is saturated in optimal circulation.* $C_{p^*}(uv) < 0 \Rightarrow f^*(uv) = cap(uv)$.

Proof: From (3), we see that $C_{p^*}(uv) < 0$ means $\ell^*(uv) > 0$. Using complementary slackness, we see that this implies that $f^*(uv) = cap(uv)$. ∎

The contrapositive form of above lemma is that if an edge is not saturated in an optimal circulation, then the optimum reduced cost is non-negative. $uv \in G_{f^*} \Rightarrow C_{p^*}(uv) \geq 0$.

The importance of the above lemma comes from the ability given p^* to compute C_{p^*} for all edges as follows. If $C_{p^*} > 0$, we can throw the edge out.

Lemma 2.11. *If there is some flow through an edge in optimal primal, then the reduced cost of that edge in optimal dual is non-positive.* $f^*(uv) > 0 \Rightarrow C_{p^*}(uv) \leq 0$.

Proof: By complementary slackness, we see that $f^*(uv) > 0$ implies $\ell^*(uv) + C_{p^*}(uv) = 0$. Now, since $\ell^*(uv) \geq 0$, we have $C_{p^*}(uv) \leq 0$. ∎

The contrapositive form of above lemma is that if the optimal reduced cost of an edge is positive, then there is no flow in that edge in any optimal flow. $C_{p^*}(uv) > 0 \Rightarrow f^*(uv) = 0$.

The above lemmas give us a way to find optimal circulation f^*, given optimal solution to dual p^*. For every edge having positive C_{p^*}, we fix its flow at zero (i.e. remove the edge). For every edge with negative C_{p^*}, we saturate it (i.e. make minimum flow on it equal to maximum flow on it equal to the actual saturated flow).

Then, in this residual graph, using these changed capacity constraints, we cancel the excesses and deficits by adding s, t connected to excesses and deficits with cost 0 and then run max-flow.

Now, we will show a necessary and sufficient condition for the optimality of a given circulation f.

Theorem 2.12. f^* *is optimal* $\Leftrightarrow \exists p$ s.t. $\forall uv \in G_{f^*}, C_p(uv) \geq 0$.

Proof: ($\Leftarrow$). Suppose that we are given a p such that, for our circulation f^*, all the reduced costs on the residual edges are positive. Then, any cycle in the residual graph G_{f^*} will have $\sum C_p(uv) \geq 0$. This means that on any cycle in G_{f^*},$\sum cost(uv) \geq 0$. Thus, there are no negative cost augmenting cycles. Then Lemma 2.7 says that f^* is optimal.

($\Rightarrow$). For an optimal circulation f^*, we need to show a p exists exhibiting no negative reduced costs. We do this by constructing such a p in the following way.

We add an extra node s to the residual graph G_{f^*}. We add zero-cost edges from s to each node in G_{f^*}. Now, for each node v, we assign

$$p(v) = \text{dist}(s, v)$$

Notice that this distance can be at the most zero, but it could be a negative number.

We can be sure that such a distance exists, only if the graph we constructed contains no negative-cost cycles. This fact is true due to the following argument: Since f^* is optimal, from Lemma 2.7, we know that that G_{f^*} has no negative cycles. Now, since all of the added edges point away from s, there are no

cycles through {s} and so no new cycles have been introduced to G_{f^*}. Thus, in the constructed graph, there are no negative cost cycles.

Finally, we will show that for any edge $uv \in G_{f^*}$, $C_p(uv) \geq 0$. By triangle inequality,

$$\begin{aligned} p(v) &= \text{dist}(s, v) \\ &\leq \text{dist}(s, u) + \text{dist}(u, v) \\ &\leq \text{dist}(s, u) + cost(uv) \\ &= p(u) + cost(uv) \end{aligned}$$

From this, we immediately see that

$$C_p(uv) = p(u) + cost(uv) - p(v) \geq 0$$

Thus, we have constructed a set of potentials, such that every edge in the residual graph has non-negative reduced cost. ∎

3 Capacity Scaling Approach to the Minimum Cost Circulation Problem

We shall first present a "direct" capacity scaling weakly polynomial algorithm for the minimum cost circulation problem. We then present Orlin's [13] improvement of the technique, which is also weakly polynomial, but can be transformed into a strongly polynomial algorithm (see [13], [14]).

3.1 The Capacity Scaling Technique

Recall that the main idea in creating a scaling algorithm is to be able to update an existing optimal solution, after one of the parameters of the current problem is changed.

Consider the minimum cost circulation problem on the directed graph G. Suppose we have obtained an optimal circulation f and a set of optimal node potentials p for G. Consider the graph G', identical to G, except that for one edge vw, $cap(vw \in G') = cap(vw \in G) + 1$. We wish to find an optimal circulation f' with respect to this new graph via f and p.

Let G'_f be the residual graph corresponding to the circulation f in G' and let G_f be the residual graph corresponding to the circulation f in G . Since f does not saturate the edge $vw \in G'$, we have $vw \in G'_f$. Let us first try to eliminate cases in which the current solution continues to be optimal (see fig 3.1).

case(i) : Suppose edge vw already exists in G_f, in that case, by incrementing its capacity, we cannot be creating any new negative cost cycles. Thus, f continues to be optimal in G'(fig 3.1 (e)).

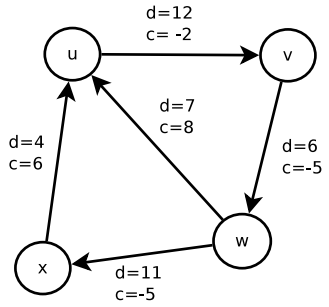
case(ii) : If edge vw does not exist in G_f, then it has now been created in G'_f with a capacity of 1 and some reduced cost $C_p(vw)$. If it so happens that there is no path from w to v in G_f, then f is still optimal in G'. This is because if vw introduces some negative cost cycle in G'_f, there would have been a path from w to v in G_f(fig 3.1 (f)).

case(iii) : Continuing the above case, if there existed a path from w to v in G_f, let P_{wv} be the shortest path from w to v in G_f and let $C_p(P_{wv})$ be its cost. If $C_p(P_{wv}) + C_p(vw) \geq 0$, then this does not form a negative cost cycle. Also, any other cycle involving edge vw would have a cost of at least $C_p(P_{wv})$ from w to v and thus, cannot be a negative cost cycle. Thus, no new negative cost cycles are introduced and f is still optimal(fig 3.1 (g)).

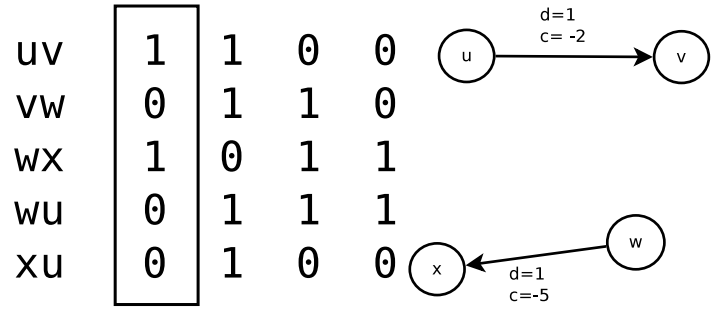
case(iv) : The only case where f is not optimal any more is when the shortest path from w to v, together with edge vw forms a negative cost cycle. To correct this, we augment along this particular negative cost cycle with unit flow which causes edge vw to vanish(fig 3.1 (h)). However, this would introduce residual edges in the other direction in the cycle, which may then close other negative cost cycles elsewhere.

To show that this does not happen, we will readjust the node potentials. Note that changing the node potentials does not affect cycle costs since all the potentials cancel out in the telescoping sum of costs. Let $d(x)$ be the length of the shortest path, with respect to the reduced costs C_p, from node w to node x in G_f. If x is not reachable, set $d(x)$ to a sufficiently large number. Since f is optimal in G, there are no negative-cost cycles in G_f, and thus the shortest path distances $d(x)$ are well-defined. For each node x, define the node potential $p'(x) = p(x) + d(x)$. We claim that for all $ij \in G_f$, the reduced costs

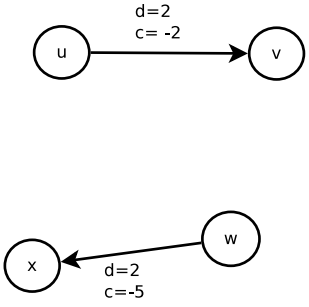
[0]Edited by Frank Li and Wendy Mu.



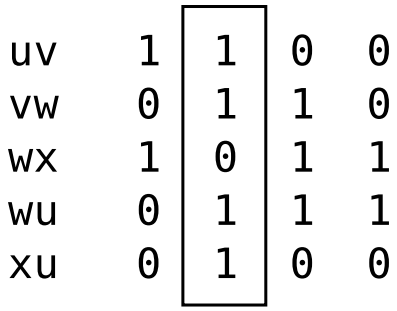
(a) Min cost circulation problem



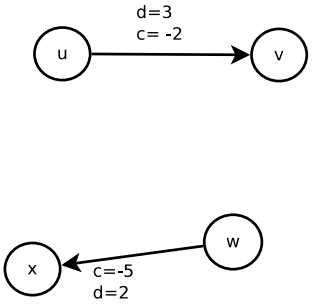
(b) Revealing the first MSB of edge capacities



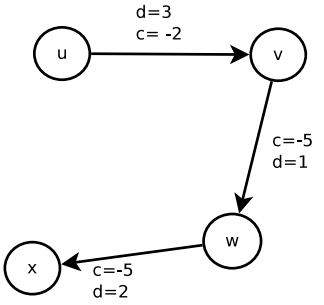
(c) Doubling flows and capacities



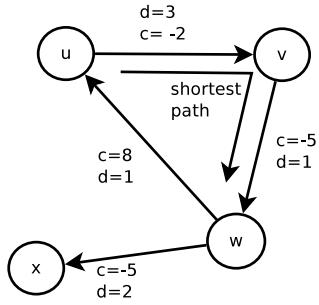
(d) Revealing the second MSB of edge capacities



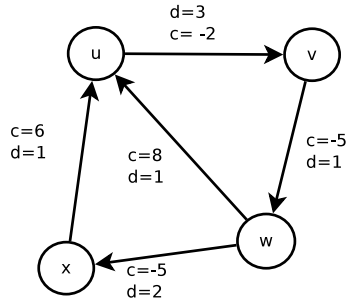
(e) Incrementing capacity of edge uv. Flow remains optimal. This is case (i)



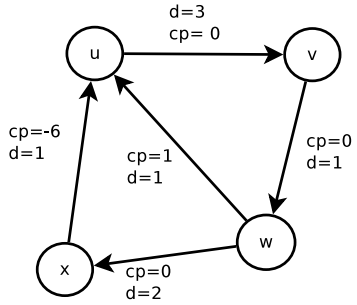
(f) Incrementing capacity of edge vw. This is case (ii) and the flow remains optimal.



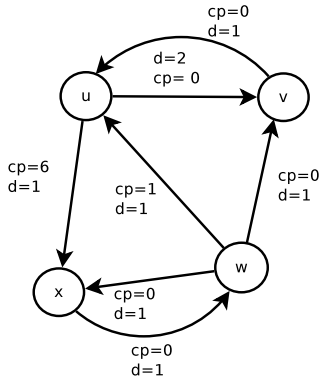
(g) Incrementing capacity of edge wu. No negative cost cycles introduced. This represents case (iii)



(h) Incrementing capacity of edge xu. This is case (iv) i.e. a negative cost cycle is formed.



(i) Reassignment of potentials w.r.t shortest distance from u



(j) Augmentation of unit flow.

$C_{p'}(ij)$ are non-negative. Observe that the shortest path distances $d(x)$ satisfy the optimality condition

$$d(j) \leq C_p(ij) + d(i) \tag{4}$$

where $C_p(ij) = cost(ij) + p(i) - p(j)$ is the reduced cost with respect to p, and the above inequality holds with equality when the edge ij is on the shortest path from node w to node j. It then follows that

$$\begin{aligned} C_{p'}(ij) &= cost(ij) + p'(i) - p'(j) \\ &= cost(ij) + (p(i) + d(i)) - (p(j) + d(j)) \\ &= C_p(ij) + d(i) - d(j) \\ &\geq 0 \end{aligned}$$

Now, using these potentials, $C_{p'}(vw) < 0$ for case (iv). Also, note that any edge on a shortest path between w and any other node will now have a reduced cost of 0. This means that all edges in P_{wv} would have zero reduced cost. More formally, $\forall_{ij \in G_f, ij \in P_{wv}} C_{p'}(ij) = 0$. When we augment 1 unit of flow along the cycle $P_{wv} \cup vw$, the resulting reverse edges in G'_f will have reduced cost $C_{p'} = 0$. Moreover, by augmenting 1 unit along the edge vw, we saturate the edge vw and remove it from G'_f. Note that the augmentation will also create the reverse edge wv in G'_f. However, since $C_{p'}(vw) < 0$, we have $C_{p'}(wv) > 0$. Thus, if we let f' be the circulation at the end of the augmentation, we have $\forall_{ij \in G'_{f'}} C_{p'}(ij) \geq 0$. It follows that f' is an optimal circulation for G'.

Note that cases $(i), (ii)$ and (iii) result in $C_{p'}(vw) \geq 0$ and so, $\forall_{ij \in G'_f} C_{p'}(ij) \geq 0$, which tells us that f itself is optimal. Specifically, when there is no path from w to v, we would have set $d(v)$ to a big number, and hence we can guarantee that $C_{p'}(vw) \geq 0$.

Thus, to compute the new optimal flow, we recompute the potentials, check the sign of $C_{p'}(vw)$ and augment if necessary. The complexity of the above algorithm can be bounded by the time required to perform the shortest path computations. Since the reduced costs of all edges in G_f are non-negative, we can apply Dijkstra's algorithm, which, using Fredman and Tarjan's [2] implementation, can be done in $O(m + n \log n)$ time.

3.2 Capacity Scaling

The technique described in the previous section can be extended to a general min-cost circulation algorithm. Consider the minimum cost circulation problem on the graph $G = (V, E)$ with costs $cost(ij)$ and capacities $cap(ij)$. Let G_k be identical to G, except that the capacity of an edge ij is given by $\lfloor cap(ij)/2^k \rfloor$. Clearly, G_0 corresponds to the original graph G. Let L be the smallest integer such that $2^L > U$, where U is the maximum capacity of an edge in G. We would like to solve the minimum cost circulation problem on G in an incremental fashion through successively solving the minimum cost circulation problems on $G_{L-1}, G_{L-2}, \ldots, G_0$.

We start with the circulation $f = 0$ and the node potential $p(u) = 0$ for all nodes u, which is optimal in the graph G_L. It is clear that in the graph G_{L-1}, the capacity of an edge is either 0 or 1. Thus, we can obtain an optimal circulation for G_{L-1} by running the algorithm in the previous section once per each edge that has capacity 1, i.e. at most m times.

Let f_k be the optimal circulation for the graph G_k. Let xf_k (xG_k) be the circulation (graph) with the flow (capacity) on every edge multiplied by x. Given the optimal circulation f_k, we extend it to f_{k-1} as follows: consider the circulation $2f_k$ in the graph $2G_k$. If f_k is optimal in G_k, then $2f_k$ is optimal in $2G_k$, since by optimality of f_k, there are no negative cycles in the residual graph of G_k corresponding

to f_k. However, the graph G_{k-1} may not be identical to $2G_k$, since from the inequalities

$$2\left\lfloor \frac{cap(ij)}{2^k} \right\rfloor \leq \left\lfloor \frac{cap(ij)}{2^{k-1}} \right\rfloor \leq 2\left\lfloor \frac{cap(ij)}{2^k} \right\rfloor + 1 \tag{5}$$

we see that the capacity of an edge ij in G_{k-1} is either identical to or one more than that of the corresponding edge in $2G_k$. Therefore, to obtain an optimal circulation f_{k-1} for G_{k-1} from the optimal circulation f_k for G_k, we first compute the optimal circulation $2f_k$ for $2G_k$. Then we apply our previous algorithm, and after at most m applications, we will obtain an optimal circulation f_{k-1}.

Since we need to find an optimal circulation for each of the graphs $G_{L-1}, G_{L-2}, \ldots, G_0$, we obtain the following result:

Theorem 3.1. *The above capacity scaling algorithm solves the minimum cost circulation problem in* $O(m\,(m + n\log n)\log U)$ *time.*

To see an example, let us look at figure 3.1 $(b-j)$ which shows the binary expansion of the capacities of the edges of a network. In this graph, G_{L-1}(fig 3.1 (b)) comprises two edges uv and wx and the optimal flow is 0. To now construct the optimal flow for G_{L-2}, we first double the flow and the capacities, and this gives figure (c) of figure 3.1. We then proceed to increment the capacities of edges uv, vw, wu and xu one at a time, as figure 3.1 depicts. At the end, we have an optimal flow for G_{L-2} and all the reduced costs are non negative. We can again double the capacities and flows and proceed till we obtain the optimal flow for G_0.

3.3 Towards a Strongly Polynomial Algorithm

The $\log U$ factor in the running time of the above algorithm makes this algorithm weakly polynomial. We would like to design a strongly polynomial algorithm for min-cost circulation. Roughly speaking, this is an algorithm whose running time depends only on the number of edges and nodes, and does not depend on the number of bits in the capacities or costs.

To obtain such a strongly polynomial algorithm, we first convert the capacitated minimum cost circulation problem to the transshipment problem. In the transshipment problem, the edges in the input graph do not have capacities, and the nodes have demands. The goal is to find a pseudoflow of minimum cost such that, for every node, the difference between the amount of flow leaving the node and the amount of flow entering the node is equal to the demand of the node. In a feasible solution, a node with positive demand will have more outgoing flow than incoming flow, and a node with negative demand will have more incoming flow than outgoing flow.

The minimum cost circulation problem on graph G is converted to a transshipment problem with graph T as follows. For each directed edge uv we introduce a new node $x(uv)$ and replace the edge uv with two uncapacitated directed edges ux and vx, setting $cost(ux) = cost(uv)$ and $cost(vx) = 0$. Furthermore, we add demands of $-cap(uv)$ at x and $cap(uv)$ at v. Figure 2 shows this reduction. If a node in the original graph has multiple incoming edges, the total demand of the node is the sum of the demands due to all the incoming edges. This completes our transformation. The advantages of such a transformation are that there are neither cycles (may be created later, though) nor capacities in the initial graph. We now see how a solution for the transshipment problem yields a solution to the original min-cost circulation problem.

Note that if G had n nodes and m edges, then after transformation we will have $n + m$ nodes and $2m$ edges.

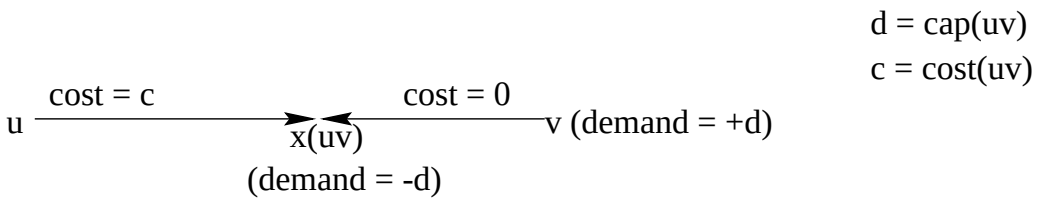


Figure 2: Converting capacities to demands

Theorem A feasible flow f for the transshipment problem is optimal if and only if there are no negative cost cycles in T_f.

Proof. This is very much similar to the proof showing that an optimal min-cost circulation would not have negative cost cycles. Firstly, if T_f had negative cost cycles, we could augment flow along the cycle without disturbing the excesses at any node(since this is a cycle). This would give us a feasible flow f' with lower cost and would thus contradict the optimality of f.

To see the other direction, suppose f is not optimal, but f^* is. Then, $f^* - f$ must be a circulation since the demand constraints are satisfied in both flows and thus the difference should not have any accumulation of flow at any node. Also, since we have infinite capacities, $f^* - f$ is feasible in T_f. Since the cost of f^* is lower than that of f, $f^* - f$ must be a negative cost circulation, which can be decomposed into at least one negative cost cycle. ∎

Once we have obtained an optimal solution to the transshipment problem T, we transform it into a solution to the original problem as follows. For every edge uv in the original problem, we set the flow equal to the flow in edge $u, x(uv)$ in the transformed problem, as shown in figure 3. This flow would be compliant with the capacity constraints of uv since not more than $cap(uv)$ amount of flow is permitted on edge $u, x(uv)$ owing to the demand of $x(uv)$.

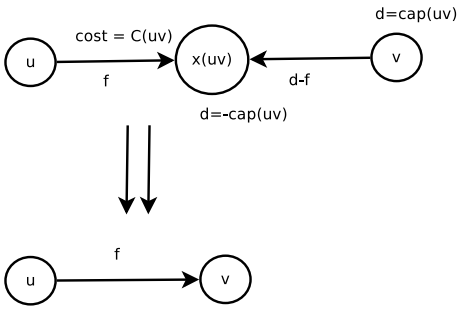


Figure 3: Conversion of a transshipment solution to a min-cost circulation solution

It is easy to see that if there are no negative cost cycles in the residual graph of the transshipment problem, then there would not be any negative cost cycles in the transformed min-cost circulation and thus, we would have solved the min-cost circulation problem. We now look at an algorithm to solve the transshipment problem.

3.3.1 Algorithm SIMPLE-SCALE

We first consider SIMPLE-SCALE, which is not strongly polynomial. This algorithm maintains the following invariant: *There are no negative reduced-cost cycles in the residual graph.* The algorithm also attempts to reduce the largest excess/deficit in the residual graph. Notice that the invariant ensures dual feasibility at all times. Therefore, if the algorithm obtains a primal-feasible solution (i.e. satisfies all the demands, both positive and negative), that solution would be optimal.

The algorithm initially chooses a scale factor Δ to be a power of 2 just larger than the initial maximum demand (in terms of absolute value) in the network, and f as the zero flow. Let $ex_f(v)$ denote the excess flow at node v when the current flow is f. Initially, $ex_f(v)$ is the demand of node v. In general, $ex_f(v)$ is the difference between the demand of v and the difference between the total outgoing flow and the total incoming flow at v.

---

procedure SIMPLE-SCALE

1: $D \leftarrow \max_u |\text{demand}(u)|$
2: $\Delta \leftarrow 2^{\lceil \log D \rceil}$
3: $f \leftarrow 0$
4: $i = 0, \Delta_i = \Delta$
5: **while** $\Delta_i \geq 1$ **do**
6: **while** $\exists u, v, ex_f(u) > \Delta_i/2 \wedge ex_f(v) < -\Delta_i/2$ **do**
7: $f \leftarrow$ Augment f with Δ_i flow from u to v by the shortest path
8: **end while**
9: $\Delta_{i+1} \leftarrow \Delta_i/2$
10: $i++$
11: **end while**

end SIMPLE-SCALE

In phase i, the algorithm repeatedly finds nodes u and v such that $ex_f(u) > \Delta_i/2$, $ex_f(v) < -\Delta_i/2$. Nodes with $|ex_f(v)| > \Delta/2$ are said to be active. The algorithm then uses the shortest augmenting path from u to v for sending a flow of exactly Δ_i units from u to v, and updates f. When there are no more augmentations between "large excess/deficit" nodes are possible, Δ_i is scaled down by a factor of 2. The algorithm terminates when i becomes large enough so that Δ_i becomes 1 for the first time, since at this point the total value of excess is at most n, and we can finish up by at most n unit augmentations over shortest paths.

One concern is that we may not be able to augment by Δ_i units from u to v owing to some bottleneck capacity. To address this, first notice that in the beginning, all capacities are infinite and so, this problem doesn't arise. As we augment flow, reverse edges appear which have some finite capacity. The following lemma addresses this concern(see [13]).

Lemma 3.2. *At every step of the* SIMPLE-SCALE *algorithm, the flow and the residual capacity of every edge in the residual graph is a multiple of the current scale factor* Δ.

Proof: Induct on the number of augmentations and adjustments of Δ. At the beginning of the algorithm, the initial flow is 0 and all capacities are ∞, and the statement is trivially satisfied. Now, assuming the inductive hypothesis, we consider two cases. First: Each augmentation modifies the residual capacity of an edge by either 0 or Δ_i – this preserves the inductive hypothesis. Secondly: Each adjustment of Δ divides it by a factor of 2, which also preserves the inductive hypothesis. ∎

It is important to augment the flow by exactly Δ_i units - augmenting by an amount greater than Δ_i units, even if possible, might destroy the property that residual capacities are multiples of the current

Δ_i, and hence are also multiples of Δ_j for $j > i$. Also, Lemma 3.2 ensures that if there exists a path from u to v in the residual graph, then the capacity along this path is at least Δ_i. Thus, it is possible to augment by exactly Δ_i without trying to identify bottleneck edges along the chosen augmenting path.

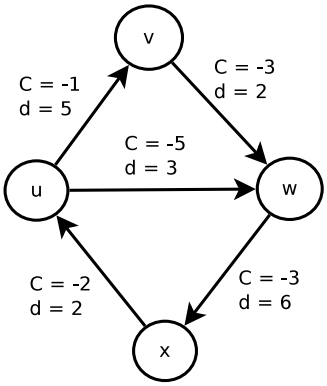
The augmentations done by the algorithm do not create any negative cost cycles and therefore, on termination, we are left with an optimal solution. There are two different ways to see that this is true. Note that before each augmentation we can update the node potentials by the distance (with respect to current reduced costs) from the starting node of the augmenting path. Similarly to what we proved in Section 3.1, this change in potential does not have any effect on our choices of paths and ensures that all newly introduced residual edges have 0 reduced costs. Since we assume (inductively) that there are no negative reduced cost residual edges before augmentation, this assumption holds after augmentation as well. An extra advantage of this approach is that we always deal only with non-negative costs and hence can use Dijkstra's shortest path algorithm.

Complexity analysis: We call a node "active" if the absolute value of excess currently at this node is above $\Delta_i/2$. First observe that an augmentation cannot create a new active node. Moreover, if at iteration i node is active after an augmentation, then absolute value of its excess was above $1.5\Delta_i$ before the augmentation. But this means that during previous iteration, absolute value of its excess was at least $.75\Delta_{i-1} > \Delta_{i-1}/2$. Hence there could be no augmentations between a pair of such nodes at phase i - we would have done this augmentation at the previous phase. This implies that each augmentation makes at least one active node inactive, which in turn leads to $O(n)$ bound on the number of augmentations per phase, one Dijkstra per augmentation.

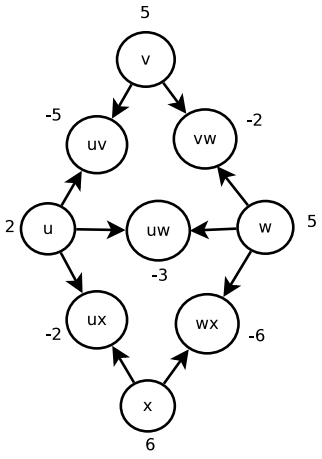
Let n', m' be the number of vertices and edges in the original graph, respectively. There are $n = n' + m'$ vertices and $m = 2m'$ edges in the transformed graph. The running time per phase is bounded by $O(n(m + n \log n))$. Due to the special properties of the transformed graph, we can modify Dijkstra to run in $O(m' + n' \log n')$ time. The trick is that the m' internal nodes can be suppressed while computing shortest paths and once shortest paths are found, they can be reintroduced with their distances computed quickly (see [13] for details).

The initial maximum demand is at most nU, where U is the maximum capacity. Since we choose the initial value of Δ to be the power of 2 just larger than the initial maximum demand, it follows that there are at most $O(\log(nU))$ Δ-scaling phases. Thus, we have the following result:

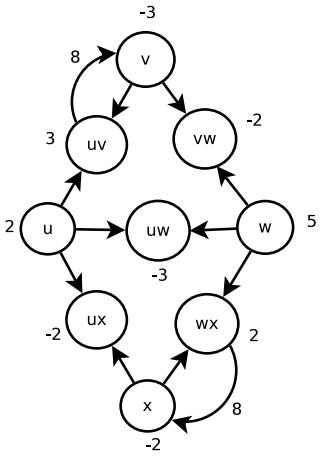
Theorem 3.3. *Using the* Simple-Scale *algorithm, the minimum cost circulation problem for a graph with n nodes and m edges can be solved in $O(m^2 \log n \log(nU))$ time.*



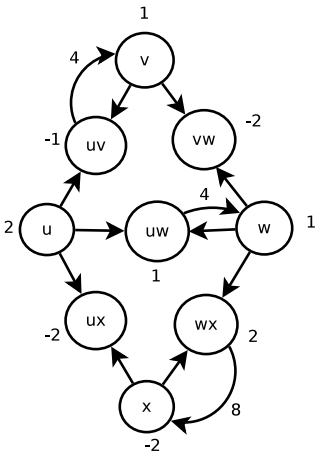
(a) Min-cost circulation problem



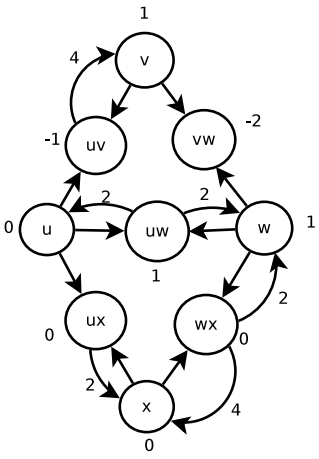
(b) Corresponding Transshipment problem with zero flow



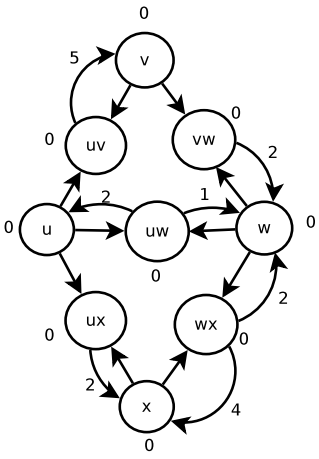
(c) $\Delta = 8$



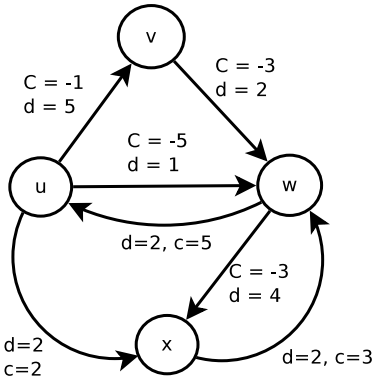
(d) $\Delta = 4$



(e) $\Delta = 2$



(f) $\Delta = 1$



(g) Solution to Min-cost circulation

Figure 3.3.1 shows the working of the algorithm. Figure (a) shows the original min-cost circulation problem and figure (b) shows the equivalent transshipment problem. Edge costs have not been shown in the transshipment figures for clarity. The demands of the nodes are indicated next to them. In figure

(c), we set $\Delta = 8$, since the maximum absolute demand is 6. We then send a flow of $\Delta = 8$ units from v to uv and from x to wx and both these pairs satisfy our augmentation criteria. Note that in figure (e), when we augment flow from u to ux, we send $\Delta = 2$ units through uw(shortest path) as opposed to sending to ux directly. Finally, after the $\Delta = 1$ iteration is done, all nodes have 0 demands and the algorithm terminates. We can translate the final residual network of figure (f) to a residual graph for the min-cost circulation (figure (g)) and this gives us the min-cost circulation.

3.4 Capacity-scaling strongly polynomial algorithm

We will now look at a strongly polynomial implementation of the SIMPLE-SCALE capacity scaling algorithm from the previous section. The original algorithm is due to Orlin. We present a modified algorithm described in the paper by Orlin, Plotkin, and Tardos [14].

3.4.1 Is SIMPLE-SCALE strongly polynomial?

Consider the transshipment problem resulting from the transformation described in the SIMPLE-SCALE algorithm. For the remainder of this section, let $\bar{n}, \bar{m}$ denote the number of nodes and edges in the original graph; let $n = \bar{n} + \bar{m}$ and $m = 2\bar{m}$ denote the number of nodes and edges in the transformed graph.

Let $b(v)$ denote the initial demand, i.e. the initial excess or deficit, at a node $v \in G$. Let $ex_f(v)$ denote the demand at v for the current flow f.

At any stage in the algorithm, for every node v there exists an adjacent edge e such that $f(e) \geq \frac{b(v)-ex_f(v)}{n}$, where f is the current flow. This occurs because $b(v) - ex_f(v)$ is, by construction, the amount of flow going out of v and there are at most n edges incident on v. (Note that the number of edges is in fact smaller – $\bar{n}$ and not n – due to the specific construction used.)

Lemma 3.4. *Given a node v, there exists some iteration j and corresponding Δ_j of* SIMPLE-SCALE *such that $\frac{b(v)-ex_f(v)}{n} \geq 5n\Delta_j$, where f is the flow in iteration j.*

First, note that for any iteration i, $|ex_f(v)| \leq n\Delta_i$ for all nodes v. This is due to the fact that the there is as much positive excess as there is negative one. In other words, even if all excess is concentrated in a single node and the deficits are spread uniformly, each one of these deficits cannot exceed Δ_i at the start of iteration i, providing a bound on the total amount of the concentrated excess.

Suppose v first becomes active in iteration i with corresponding Δ_i. Then we know $b(v) \geq \frac{1}{2}\Delta_i$. Now consider the iteration j in which $\Delta_j = \frac{\Delta_i}{16n^2}$. Then

$$\begin{aligned} \frac{b(v) - ex_f(v)}{n} &\geq \frac{\frac{1}{2}\Delta_i - n\Delta_j}{n} \\ &= 8n\Delta_j - \Delta_j \\ &\geq 5n\Delta_j \end{aligned}$$

So in iteration j, v has an adjacent edge e with flow $f(e) \geq 5n\Delta_j$. Since $\frac{\Delta_i}{\Delta_j} = 16n^2$, $j - i = O(\log n)$; that is, it takes $O(\log n)$ iterations after v first becomes active for some adjacent edge e to satisfy this inequality[1].

Now note that during a single phase q there are at most n augmentations, and hence at most $n\Delta_q$ flow is added to any edge. Since we scale down Δ by a factor of two each iteration, the sum of all flow added to edge e after iteration j is bounded above by

$$n\Delta_j + n\frac{\Delta_j}{2} + n\frac{\Delta_j}{4} + \ldots \leq 2n\Delta_j$$

Then since e has flow $\geq 5n\Delta_j$, both e and its reverse residual edge have capacities $\geq 5n\Delta_j$ (in fact, one of these edges has inifinite capacity from the original transformation). Therefore, neither edge will

[1] It is possible to easily tighten the constants, but this will not improve the overall asymptotic running time.

get saturated after iteration j, so far as we are concerned, they both have infinite capacity from this point forth. SIMPLE-SCALE can use this fact to contract this pair of edges, reducing the number of nodes by one.

When we contract this edge, we must take care to rebalance the potentials so that we only contract on 0 reduced-cost edges. Otherwise, we would not be accounting for the cost of pushing flow within a contracted node. Also, if contraction would create multiple edges to the same node, we treat all such edges as a single edge and use the lowest-cost edge until it is saturated, at which point we switch to the next-lowest cost edge.

This seems to imply that each node can be part of at most $O(\log n)$ augmentations. Why, then, do we not have a strongly polynomial algorithm? The answer lies in what happens after a contraction. Suppose nodes u and v get contracted. The excess at the new node is $ex_f(u) + ex_f(v)$, and the initial demand at this node was $b(u)+b(v)$. But $ex_f(u)+ex_f(v)$ may be much larger than $b(u)+b(v)$ (consider the case where $b(u)$ is very positive and $b(v)$ is very negative), and it might take a large number of iterations before $ex_f(u)+ex_f(v)$ falls below $b(u)+b(v)$. Therefore, the above analysis does not hold for a contracted node.

3.4.2 Making SIMPLE-SCALE strongly polynomial

This problem can be fixed by using more involved augmentations which eliminate the possibility of very small $b(v)$ values in contracted nodes. Once again, Δ is scaled down by a factor of two each iteration, and all the augmentations are done in units of Δ.

Define $S_f(\alpha) = \{v : ex_f(v) > \alpha\}$ and $T_f(\alpha) = \{v : ex_f(v) < -\alpha\}$. For each Δ_i, the algorithm does the augmentations in two steps:

Step 1: While there is a $u \in S_f(\frac{n-1}{n}\Delta_i)$ and $v \in T_f(\frac{1}{n}\Delta_i)$, augment Δ_i units of flow from u to v.

Step 2: While there is a $u \in S_f(\frac{1}{n}\Delta_i)$ and $v \in T_f(\frac{n-1}{n}\Delta_i)$, augment Δ_i units of flow from u to v.

As before, the algorithm contracts an edge whenever the flow on that edge exceeds $5n\Delta_i$, where Δ_i is the current value of Δ. Using this approach we can assure that $b(u)+b(v)$ is not small when two nodes u and v are contracted.

We say node v is active in the current Δ-scaling phase if $v \in S_f(\frac{n-1}{n}\Delta_i) \cup T_f(\frac{n-1}{n}\Delta_i)$. The node v is said to be activated by the current Δ-scaling phase if v is not active at the end of the previous Δ-scaling phase and it becomes active either at the beginning of this phase when Δ is scaled down, or when excess is collected after an edge gets contracted.

Lemma 3.5. *There are at most n augmentations per phase.*

Proof: By the end of the phase there should be no active nodes left. For contradiction, assume that there is a positive active excess left at the end of an phase i. By definition, its excess value exceeds $\frac{n-1}{n}\Delta_i$. Thus the total absolute value of the deficits exceeds $\frac{n-1}{n}\Delta_i$, which in turn implies that there exists a deficit with absolute value above Δ_i/n. This deficit, together with the above excess are a valid pair for yet another augmentation in phase i, which means we stopped that phase too early. Active deficit can be dealt with similarly.

This implies that at the beginning of iteration i, all active excesses have absolute values between $2\frac{n-1}{n}\Delta_i$ and $\frac{n-1}{n}\Delta_i$. Hence a single augmentation deactivates such a vertex. ∎

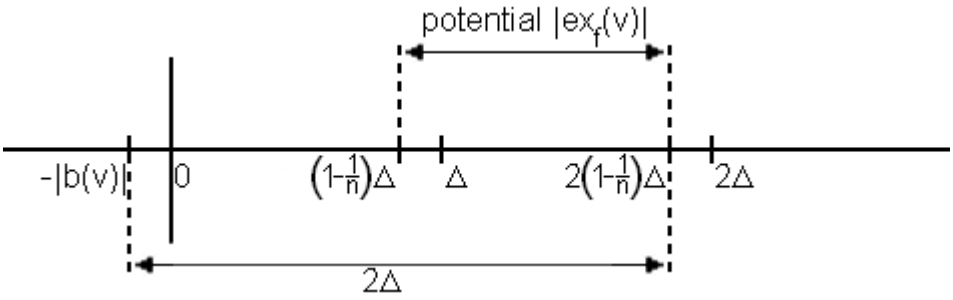


Figure 4: Our augmentation process ensures that the excess of a contracted node v is in the shown range, and that the initial demand of v was far enough from 0 that we can claim strong polynomiality.

Given the above observations, the following lemma implies a strongly polynomial running time. This lemma holds for the original nodes in the transformation as well as the nodes that are created by edge contractions.

Lemma 3.6. *A node v can be activated at most $O(\log n)$ times.*

Proof: If v was a node in the transformation before SIMPLE-SCALE was applied, i.e. v is not a contracted node, then the lemma is easy to show using an analysis very similar to that in Section 3.4.1.

Now suppose v is a new node created from an edge contraction. Then v might get activated once initially due to contraction. However, when it is activated for the second time, this must occur because of Δ-scaling. Therefore,

$$\frac{n-1}{n}\Delta \leq |ex_f(v)| < 2\frac{n-1}{n}\Delta$$

We are concerned about the case when $|b(v)|$ is small compared to $|ex_f(v)|$ However, since $b(v) - ex_f(v)$ must be an integral multiple of 2Δ (as all previous augmentations were multiples of 2Δ), we have (see Figure 4)

$$|b(v)| \geq \left|2\left(1-\frac{1}{n}\right)\Delta - 2\Delta\right| = \frac{2}{n}\Delta$$

Then $|ex_f(v)| \leq n\Delta$ and $|b(v)| \geq \frac{2}{n}\Delta$, and $\frac{|ex_f(v)|}{|b(v)|} = O(n^2)$. Therefore, after $O(\log(n^2)) = O(\log n)$ activations, $|b(v)| \geq |ex_f(v)|$. At this point, the original analysis applies, and after an additional $O(\log n)$ activations, $\frac{b(v)-ex_f(v)}{n} \geq 5n\Delta$, and v would disappear due to contraction. In total, v can become activated $O(\log n)$ times before it disappears.

Using the above lemma together with the fact that each augmentation can be implemented with one call to Dijkstra's shortest path algorith, implies the following result.

Theorem 3.7. *The above algorithm solves the min-cost circulation problem in $O(n^2 \log^2 n) = O(\bar{m}^2 \log^2 \bar{n})$ time.*

The above analysis implicitly assumes that there is no time spent on Δ-scaling phases during which no augmentations take place. This assumption is valid since finding nodes which fall between Δ and 2Δ involves looking at the current excesses of each node. A slight modification of the algorithm allows us to track these values as they are examined and effectively "jump over" several factors of Δ if there do not exist any nodes in that range.

4 Cost scaling approaches for min-cost circulation problem

Note that an optimal circulation does not change if all costs are doubled, because no new negative-cost cycles are introduced. So this leads us to consider a cost-scaling approach to solving min-cost circulation. In particular, we consider what happens to the optimal flow when the cost on an edge is increased or decreased by 1.

Suppose the cost of an edge uv in the residual graph is decreased by 1. Then this edge could be part of a negative-cost cycle. This would happen only if the reduced cost along this edge previously had reduced cost 0, so that now it is -1.

To resolve this, we can compute shortest paths from v. If there is no path of 0 reduced cost edges to u, then there is no path from v to u, and adding uv will not introduce a cycle. Otherwise, we compute max-flow from v to u along only 0-reduced-cost edges capped by the capacity of the uv edge. Augmenting by the max flow along these paths and uv creates a cut separating u and v. At this point, uv is no longer part of a negative-cost cycle, since there is no 0-reduced-cost path from v to u.

Finally, we adjust the potentials so that each edge has nonnegative reduced cost. Note that only uv has negative reduced cost. Any edge along which we pushed flow in the previous step had 0 reduced cost, and thus its reverse residual edge also has 0 reduced cost, and by optimality of the given previous flow, all other edges also have nonnegative reduced costs. We find the set S of all nodes reachable from v along only 0-reduced-cost edges, and decrease the potential of every node in S by 1. Note that for every edge with both endpoints in S or both endpoints not in S, the reduced cost of the edge stays the same. For every edge from a node outside S to a node inside S, the reduced cost will increase by 1. For every edge from a node inside S to a node outside S, the reduced cost will decrease by 1, but it must have had reduced cost > 0, and thus the resulting reduced cost will also be nonnegative. Therefore, after adjusting the potentials in this way, we obtain a min-cost circulation with nonnegative reduced cost edges in the residual graph. See Figure 5.

This leads to a cost-scaling algorithm that starts with all flows being 0 and all edges having 0 cost. In each iteration, the cost of all edges are doubled, and then the next bit of the cost of each edge is revealed. At all times, no negative-cost cycles exist, so the flow is always optimal. This algorithm is simple but slow.

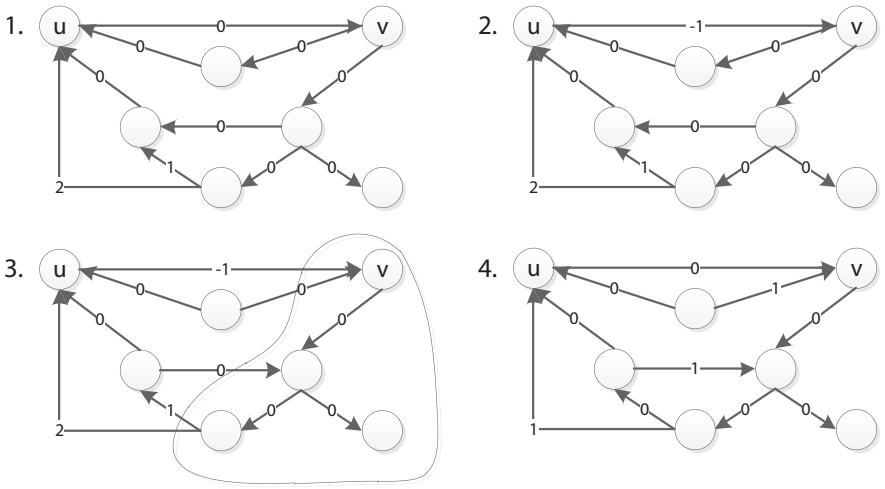


Figure 5: Example of cost-scaling. Step 1 is the original residual graph with reduced costs. In Step 2, the reduced cost of uv decreases by 1. In Step 3, a max flow has been run from v to u, generating a cut that defines set S, outlined. In Step 4, all node potentials in S are decreased by 1, eliminating all negative cost edges.

5 Cost-Scaling Strongly Polynomial Algorithm

In this section we describe how to use cost-scaling to get a polynomial time min-cost-flow algorithm. Then we show that this algorithm, with minor modifications, is in fact strongly-polynomial.

For now, let's assume that the costs are integers. Recall that a circulation is optimum (min-cost) if and only if there are no negative-cost cycles in the residual graph. In other words, there exists p, such that $c_p(e) \geq 0$ for all residual edges. This characterization is of the true/false type, i.e. it can be used to show that a circulation is optimum or not, but cannot be directly used to measure how close a circulation is to the optimum. We will use a modification of this characterization as follows.

Definition 5.1. *A flow f is said to be* **ϵ-optimal** *if and only if there exists a node potential p such that for all edges $e \in G_f$, $c_p(e) \geq -\epsilon$.*

Note that an optimum flow is ϵ-optimal for $\epsilon = 0$.

Lemma 5.2. *The cost of an ϵ-optimal circulation is at most $m\epsilon U$ more than the optimum.*

Proof: Recall that the switching from using original costs to using reduced costs does not change the cost of the circulation. Assume that f is $\epsilon - optimal$. This means that all residual edges have cost above $-\epsilon$. The only way to decrease the cost is to somehow increase the flow along negative reduced-cost residual edges. In the worst case all edges have maximum possible residual capacity U and all have reduced cost $-\epsilon$, and we somehow succeeded in saturating all of them. This will reduce the cost by at most $m\epsilon U$. ∎

Lemma 5.3. *If f is ϵ-optimal for $\epsilon < 1/n$ then f is optimum.*

Proof. Each edge e in a residual cycle has $c_p(e) > -1/n$. Any residual cycle Γ must therefore have a cost $c(\Gamma) > (-1/n) * |\Gamma| > -1$ since there can be at most n edges on a cycle. But all costs are integral; therefore, the cost of any residual cycle must be non-negative, and thus f is optimum. ∎

It is easy to see that a circulation f is ϵ-optimal for $\epsilon = |\max_{e \in G_f} c(e)|$. This implies that every f is optimal for some ϵ. Thus, there is a minimum value of ϵ such that f is ϵ-optimal. A natural question to ask is whether we can compute this minimum value of ϵ. As we discussed above, this value can be then used to assess how far is f from the optimum.

First assume that we *guessed* some ϵ. From definition of ϵ-optimality of f, there exists p such that $c_p(e) \geq -\epsilon$ for all residual edges e. Now add ϵ to the cost of every edge. The new reduced costs of residual edges are all non-negative, which implies that after the cost modification we have no negative-cost residual cycles *if the guess of ϵ was valid*. [Note that this statement is correct, even though we do not know what p is, only that it exists.]

Now add a special node s and connect it with directed 0-cost edges to all nodes in the graph, and compute distances from s to all nodes using residual edges. If the guess of ϵ was valid, the shortest path computation will produce valid distances. Moreover, these distances can be used as values for the potential p in the definition of ϵ-optimality to prove that indeed f is ϵ-optimal. This is due to the fact that shortest paths properties imply that $c_p(uv) = c(uv) + p(u) - p(v) = c(uv) + dist(s,u) - dist(s,v) \geq 0$.

If the guess was wrong, that is, it wasn't large enough to compensate all negative-cost cycles, then the shortest path computation will return with "invalid, negative cost cycle exists".

There are two immediate possible ways to find a valid ϵ. We could use a binary-search approach, but, instead, a better approach is to relate minimum ϵ to another value, the *minimum-mean cost cycle*. More precisely, define mean-cost of a cycle to be equal to the cost of the cycle divided by the number of edges in the cycle, and denote minimum such mean by

$$\mu(G_f, c) = \min_{\Gamma} \frac{\sum_{e \in \Gamma} cost(e)}{|\Gamma|}$$

Here the parameters G_f and c refer to the fact that the graph in question is residual graph with respect to the current circulation and costs c. Karp's algorithm, which is structurally similar to Bellman-Ford, can be used to compute μ in $O(mn)$ time.

Lemma 5.4. *If f is not an optimum flow, then $\epsilon(f) = -\mu(G_f, c)$, where $\epsilon(f)$ is the smallest ϵ such that f is ϵ-optimal.*

Proof. $\geq$ **direction:**
Suppose f is ϵ-optimal. Then, there is a node potential p such that on the minimum mean cost cycle Γ

$$c(\Gamma) = \sum_{e \in \Gamma} c_p(e) \geq -\epsilon * |\Gamma|$$

$$\frac{c(\Gamma)}{|\Gamma|} \geq -\epsilon$$

Thus, by definition of μ, it follows that $\mu \geq -\epsilon$.

$\leq$ **direction:**
Now, we want to show that $\mu \leq -\epsilon$. It suffices to prove that f is $-\mu$-optimal, because this means that $-\mu$ is a valid ϵ for f, so the minimum ϵ, $\epsilon(f)$, can be no larger than that.

Denote $c' = c - \mu$. We consider the cost of the minimum mean-cost cycle Γ w.r.t. c':

$$c'(\Gamma) = \sum_{e \in \Gamma} (c(e) - \mu) = c(\Gamma) - \mu|\Gamma| \geq 0$$

Thus, there are no negative-cost cycles in G_f with respect to c'. To see this, we consider some other cycle Γ': it has a higher mean cost than Γ, because Γ is defined as the the minimum mean-cost cycle. Therefore, one can compute potentials p for which f is $-\mu$-optimal. ∎

Observe that if we compute reduced costs that prove that f is $-\mu$-optimal, then along the minimum cost mean cycle, each edge must have a reduced cost of exactly μ.

The above discussion immediately implies the following algorithm: Repeatedly find the minimum mean cost cycle and augment along this cycle so that at least one edge gets saturated. After at most m augmentation we get rid of all possible minimum-mean-cost residual cycles. This implies that the value of $\epsilon = -\mu$ should decrease. The algorithm terminates when $-\mu$ falls below $1/n$.

The problem with the above approach is that although it guarantees that at each phase (at most m cycle augmentations per phase) the value of ϵ is decreased, we are not guaranteed a large decrease. Instead, we will use the following simple modification of the above algorithm:

- Compute μ
- Compute reduced costs
- Repeatedly find cycle that consists of only negative reduced-cost residual edges, augment along this cycle
- when no such cycles found, go back to the first step.

Every augmentation along a cycle saturates at least one negative-reduced-cost residual edge. It also might introduce new residual edges, but all of them have positive cost. Thus, after at most m augmentations, we are left with no cycles that consist solely of negative reduced-cost edge. Thus, at this point, each cycle in the residual graph must have at least one edge with a non-negative reduced cost, and therefore the cost of a length K cycle has to be at least $-(K-1)\epsilon$. This implies that the new minimum mean cost cycle has mean cost bounded by

$$\mu' \geq -(1-1/k)\epsilon \geq -(1-1/n)\epsilon$$

Therefore, at each new iteration, ϵ must decrease by a factor of at least $1-1/n$.

Recall that $|C_{\max}|$ is an upper bound on the initial value of ϵ and we need to reduce ϵ to below $1/n$. This implies $O(n\log(nC_{\max}))$ phases, where at each phase we do at most m augmentations, each costing $O(m)$ time. The time to compute minimum-mean cost cycle at the beginning of each phase is smaller than the running time of the rest of the phase, leading to total $O(m^2 n\log(nc_{max}))$ bound on the running time of the algorithm.

Next we will show that a better analysis [4] of essentially the same algorithm gives a strongly polynomial guarantee on the running time.

The following lemma shows that if the *optimum* reduced cost of some edge is very high compared to the current ϵ, then the current ϵ-optimal circulation and an optimal circulation have the same flow on this edge.

Lemma 5.5. *Let p^* be the optimal labels for the flow problem and f^* the corresponding flow. Suppose an arc e has $|c_{p^*}(e)| > n\epsilon$. Then for any ϵ-optimal flow f, $f(e) = f^*(e)$.*

Proof. It is enough to prove this lemma for the case $c_{p^*}(e) > n\epsilon$ (for the other case consider the reverse arc). Observe that in this case $f^*(e) = 0$. Otherwise, the reverse arc would be in G_{f^*} and will have negative reduced cost with respect to p^*, contradicting the optimality of f^*.

Assume f is ϵ-optimal, but $f(e) \neq f^*(e)$. As we have shown above, this means $f(e) > f^*(e)$.

Consider $G_> = \{e' \in E \mid f(e') > f^*(e')\}$. $G_>$ consists of edges where f is larger than f^*. By construction, the edge e is in $G_>$. Moreover, all edges e' in $G_>$ are in the residual graph of f^* and hence have $c_{p^*}(e') \geq 0$.

By considering the difference between f and f^* (the circulation $f - f^*$ in G_{f^*}) and using a flow-decomposition argument, it is easy to see that $G_>$ must contain a cycle, say Γ, that passes through e. [Intuitively, the extra flow on e has to come from somewhere.]

The cost of Γ is at least

$$c(\Gamma) = \sum_{e' \in \Gamma} c_{p^*}(e') \geq c_{p^*}(e) > n\epsilon$$

Consider the cycle $\overline{\Gamma}$ obtained by reversing the edges on Γ. Since the cost of Γ is above $n\epsilon$, the cost of the reversed cycle is below $-n\epsilon$ (in other words, it is "very negative"). But the reversed cycle is in G_f. [Can you explain why?] This is a contradiction, since the ϵ-optimality of f implies that cost of any residual cycle in G_f is bounded by $-n\epsilon$. ∎

An edge e which has $|c_{p^*}(e)| > n\epsilon$ is said to be ϵ-fixed. Observe that the above lemma implies that all ϵ-optimal flows must agree on the flow on this edge. In particular, the flow on this edge is the same for all values of $\epsilon' < \epsilon$.

We want to prove that as we decrease ϵ, we get new edges with such a large reduced cost frequently, so that all the edges become fixed quickly.

Lemma 5.6. *Consider an $n\epsilon$-optimal circulation f that is not $n\epsilon'$-optimal for any $\epsilon' < \epsilon$. Then there exists an edge $e \in G_f$ that is ϵ'-fixed for all $\epsilon' < \epsilon$ but not $n\epsilon$-fixed.*

Proof. Consider the minimum mean cost cycle Γ in G_f. Since f is $n\epsilon$-optimal, and we can augment along this cycle to get another $n\epsilon$-optimal circulation with different flows on these edges, the edges in this cycle are not $n\epsilon$-fixed.

Let μ be the cost of the minimum mean cost cycle Γ. $|\mu| = n\epsilon$ by definition of ϵ ($|\mu| \leq n\epsilon$ because f is $n\epsilon$-optimal, and $|\mu| \geq n\epsilon$ since f is not $n\epsilon'$-optimal for any $\epsilon' < \epsilon$). Thus, there exists an edge in this cycle such that $|C_{p^*}(e)| \geq n\epsilon$. By the previous lemma, e is ϵ'-fixed for all $\epsilon' < \epsilon$. ∎

The last lemma says that when ϵ falls by a factor of n, the flow on at least one more edge gets fixed. Since in every phase we are guaranteed to reduce ϵ by $(1 - 1/n)$, an edge gets "fixed" after $O(n \log n)$ phases. Since there are only m edges, there can be only $O(mn \log n)$ phases. For each phase, we have at most m augmentations, and each augmentation can be done in $O(m)$ time.

Theorem 5.7. *The cost scaling algorithm described in this section works in $O(m^3 n \log n)$ time.*

6 Cones and Fundamental Theorem of Linear Inequalities

6.1 Definitions

We begin with some conventions and definitions. Throughout these sections, we will assume we are working in the vector space R^n. Vectors in this space will be denoted by lowercase Roman letters a, b, c, etc. Scalars will be denoted by lowercase greek letters $\alpha, \beta, \gamma, \ldots$, and matrices by capital Roman letters A, B, C, etc. (capitals will also be used to denote sets). Juxtaposition will denote scalar multiplication, dot product, or matrix multiplication depending on the types of the operands involved. For sets X and Y, let $X + Y$ denote $\{x + y \mid x \in X, y \in Y\}$ and similarly let rX denote $\{rx \mid x \in X\}$. We now define some geometric objects which will appear frequently:

Definition 6.1. *A set $C \subset R^n$ is a* convex cone, *or just a cone, if $\lambda C + \mu C \subseteq C$ for any $\lambda, \mu \geq 0$. The cone* generated *by $\{x_1, \ldots, x_m\}$ is the smallest cone containing the x_i, namely $\{\sum \lambda_i x_i \mid \lambda_i \geq 0\}$, and is called a* finitely generated cone, *denoted by* cone$(\{x_i\})$.

Definition 6.2. *A* polyhedron *is the intersection of finitely many affine half-spaces. It thus has the form $\{x \mid Ax \leq b\}$. A* polytope *is a bounded polyhedron. An alternative, equivalent definition (equivalence will follow from a lemma to be proved later) is that a polytope is the convex hull of a finite set of points.*

6.2 The Fundamental Theorem of Linear Inequalities

We first state the fundamental theorem, and then discuss its meaning.

Theorem 6.3. *Let $a_1, \ldots, a_m \in R^n$. Then $\forall b \in R^n$, either*

1. *$b = \sum \lambda_i a_i$ for some $\lambda_1, \ldots, \lambda_m \geq 0$, or*
2. *$\exists c$ such that the hyperplane $\{x \mid cx = 0\}$ contains at least $t - 1$ linearly independent a_i with*
   - *$cb < 0$,*
   - *$ca_i \geq 0$ for all i,*
   - *$t = \text{rank}\{a_1, \ldots, a_m, b\}$*

What this theorem is saying is that if we consider the cone generated by the a_i, then either b is in the cone, or there exists a plane which separates b from the cone. Note that the two conditions in the theorem are exclusive, since when situation 1 holds, having $cb < 0$ necessarily implies that for some i, $ca_i < 0$, violating condition 2.

Proof. The definition of t is actually a compact way to describe two cases. If b is linearly independent of the a_i, then a hyperplane satisfying the theorem is just any hyperplane which contains all the a_i but not b (clearly such a hyperplane exists). On the other hand, if b is a linear combination of the a_i, then we can restrict our attention to the t-dimensional subspace spanned by the a_i, and find a hyperplane in this space which is spanned by $t - 1$ of the a_i. So to simplify matters, let's also assume that the a_i span R^t. From the $\{a_i\}$, choose some arbitrary basis $D = \{a_{i_j} \mid j = 1, \ldots, t\}$ of R^t. We will iteratively modify this basis until it provides a proof of the theorem. To do so, write $b = \sum \lambda_{i_j} a_{i_j}$. If all the $\lambda_{i_j} \geq 0$, then case 1 holds and we are done. If not, choose the smallest index i_k such that $\lambda_{i_k} < 0$, and remove a_{i_k}

[1] Edited by Charles Chen, Peter Lofgren and Shayan Ehsani.

from D. The remaining vectors define a hyperplane. Let c be a normal to this hyperplane, with the direction of c chosen so that $cb < 0$. Now check whether for each i, $ca_i \geq 0$. If so, case 2 holds and the theorem is proven. If not, take the smallest index s such that $ca_s < 0$ and add a_s to the basis. Note that since $\lambda_{i_k} < 0$ and $0 > cb = \sum \lambda_{i_j} ca_{i_j} = \lambda_{i_k} ca_{i_k}$ by our choice of c, $ca_{i_k} > 0$, so $a_s \neq a_{i_k}$ and thus, we have a different basis than the one we had at the beginning of the iteration. This ends one iteration.

To prove the theorem, it remains to show that this iterative procedure eventually terminates in one of the ways mentioned above. Since the number of possible bases is finite (though exponential in t), it suffices to show that no basis can arise more than once. Let D_i denote the basis before iteration i, and suppose $D_k = D_l$ for some $k < l$. Let r be the highest index for which a_r was removed during some iteration between k and $l-1$, inclusive, say during iteration p. Since $D_k = D_l$, a_r was necessarily put back at some iteration preceding l, say at iteration q. Let us concentrate on D_p and D_q. Let $D_p = \{a_{i_j}\}$, $b = \sum \lambda_{i_j} a_{i_j}$, and let c_q be the normal to the hyperplane found in iteration q. Consider

$$c_q b = \sum \lambda_{i_j} c_q a_{i_j}.$$

This quantity is less than 0, because c_q is defined so that $c_q b < 0$. However, we will show a contradiction by proving that the right hand side of this equation is positive. We do so by showing that each term in the right hand sum is nonnegative, and that one is positive.

First consider terms with an index $i_j > r$. If $a_{i_j} \in D_p$, then since no vector of index exceeding r was removed between iterations k and l, it must be the case that $a_{i_j} \in D_q$ as well. But now we need merely recall that c_q was defined so that $c_q a_{i_j} = 0$.

Now consider the term with index r, which must be in the sum because of the way D_p is defined. Since a_r was brought back in iteration q, necessarily $c_q a_r < 0$. Also, in iteration p vector a_r was discarded, so it must be that $\lambda_r < 0$. Thus the term with index r in the above equation is positive.

Finally, consider $i_j < r$. $\lambda_{i_j} \geq 0$ because otherwise this index would have been selected and discarded in iteration p. On the other hand, $c_q a_{i_j} \geq 0$ because otherwise index i_j would have been used in iteration q. Thus $\lambda_{i_j} c_q a_{i_j} \geq 0$. The contradiction has thus been shown. ∎

Those who have seen the simplex algorithm may note the similarity of what we are doing here to the simplex method. In particular, consider applying the simplex method to find a feasible solution to

$$\left[\begin{pmatrix} \\ a_1 \\ \\ \end{pmatrix} \begin{pmatrix} \\ a_2 \\ \\ \end{pmatrix} \cdots \right] \Lambda = b, \text{ subject to } \Lambda \geq 0$$

where $\Lambda = (\lambda_1, \ldots, \lambda_m)$. We either find a solution (a non-negative combination of the a_i) or find a proof that no such solution exists. Like simplex, the above procedure has exponential time complexity, since it may have to try a number of bases which is exponential in size of problem description (or equivalently, in dimension n).

Notice that the rules for adding and removing basis vectors are crucial for the property that each basis appears at most once. There exist many such *anticycling rules*, other than the ones presented here, based on the smallest indices. However, for any known anticycling rule, specific input examples have been found on which this procedure runs in exponentially many steps.

6.3 Applications of the Fundamental Theorem

Consider the following alternative definition of cones: Let a *polyhedral cone* be a set of the form $\{x \mid Ax \leq 0\}$. It turns out that we can show the following, using the fundamental theorem just proved:

Theorem 6.4. *A cone is polyhedral if and only if it is finitely generated.*

Proof. We first prove that any finitely generated cone is polyhedral. Consider a cone generated by $x_1, \ldots, x_m \in R^n$. Consider the set $\mathcal{H}$ of half-spaces through the origin which correspond to hyperplanes defined by some $n-1$ linearly independent x_i and contain every x_i. Note there are only finitely many such half-spaces. Each such hyperspace is convex which means it contains all convex combinations of the x_i and therefore contains C. Thus $C \subset \bigcap \mathcal{H}$. On the other hand, consider some $b \notin C$. By the fundamental theorem b is on the wrong side of at least one of the half-planes in $\mathcal{H}$, so $b \notin \bigcap \mathcal{H}$. It therefore follows that $C = \cap \mathcal{H}$ and is therefore a polyhedral cone.

Now consider the other direction. Suppose $C = \{x \mid a_1 x \leq 0, a_2 x \leq 0, \ldots\}$. Consider the cone generated by the a_i. By the previous paragraph, it is a polyhedral cone, i.e. it has the form $C' = \{x \mid b_1 x \leq 0, b_2 x \leq 0, \ldots\}$. Let C'' be the cone generated by the b_j. We claim $C = C''$. To prove $C'' \subset C$, it suffices to show that $\forall j$, $b_j \in C$. But since $a_i \in C'$ by definition, it is necessarily the case that $a_i b_j \leq 0$; therefore $b_j \in C$ follows. To prove $C \subset C''$, suppose $x \in C$ but $x \notin C''$. Then the fundamental theorem tells us there is a hyperplane separating x from the b_j, i.e. a vector c such that $cx > 0$ but $cb_j \leq 0$ for each j. From this deduce that $c \in C'$. Since C' is generated by the a_i, we have $c = \sum \lambda_i a_i$ where the λ_i are positive. Then $cx = \sum \lambda_i a_i x$. Since $x \in C$, we have $a_i x \leq 0$ for each i. Thus $cx \leq 0$, contradicting the fact that $cx > 0$. ∎

Note that a polyhedral cone is a more restricted entity than a general cone, because a general cone can have sides which are not planes (*e.g.* the cone in R^3 defined by $x^2 + y^2 < z$). Recall that we define a polyhedron to be a subset of R^n that can be written as $\{x \mid Ax \leq b\} = \{x \mid a_1 x \leq b_1, a_2 x \leq b_2, \ldots, a_m x \leq b_m\}$, where a_i is the i'th row of matrix A, and we define a polytope to be a bounded polyhedron. In the figure below you can see a polytope in the two dimensions space and its corresponding a_i's.

Lemma 6.5. *Any polyhedron $P = \{x \mid Ax \leq b\}$ is of the form $Q + C$ for some polytope Q and cone C.*

Proof. The idea is to lift up the dimension of the polyhedron by 1, so consider the vectors $\{\binom{x}{\lambda} \mid x \in R^n, \lambda \in R, \lambda \geq 0, Ax \leq \lambda b\}$. This is a polyhedral cone in R^{n+1}, since the constraint $Ax \leq \lambda b$ can be written as $\begin{pmatrix} A & -b \end{pmatrix} \binom{x}{\lambda} \leq 0$. Thus by the previous theorem it is generated by some set $A' = \{\binom{x_1}{\lambda_1}, \binom{x_2}{\lambda_2}, \ldots\}$, where we can normalize these vectors so that each λ_i is either 1 or 0. Let $X_0 = \{x_i \mid \lambda_i = 0\}$ and $X_1 = \{x_i \mid \lambda_i = 1\}$. We claim that this cone can be taken as the sum of the cone C generated by X_0 and the polytope Q generated by X_1. To prove this, consider $p \in P$. We have

$$\begin{aligned} p \in P &\iff Ap \leq b \\ &\iff \binom{p}{1} = \sum_{x \in X_1} \alpha_x \binom{x}{1} + \sum_{x \in X_0} \mu_x \binom{x}{0} \end{aligned}$$

(At this point, moving in either direction in the implication, we can deduce that $\sum \alpha_x = 1$ and $\mu_x \geq 0$.)

$$\begin{aligned} &\iff p = q + c \quad \text{where} \quad q = \sum_{x \in X_1} \alpha_x x, c = \sum_{x \in X_0} \mu_x x \\ &\iff p = q + c \text{ with } q \in Q \text{ and } c \in C \end{aligned}$$

∎

In other words, the above theorem states that if you have a general polyhedron, then any point in this polyhedron can be represented by a sum of two points, one in a cone and one in a polytope.

7 Farkas Lemma and Linear Programming Duality

7.1 Farkas' Lemma

In this section we will use a simple variant of the *Farkas' lemma* which characterizes a necessary and sufficient condition for the existence of a feasible non-negative solution to a set of linear inequalities with the form $Ax = b$. From this, we will construct an alternate form of Farkas' lemma which gives a necessary and sufficient condition for the existence of any feasible solution to a set of linear inequalities with the form $Ax \leq b$.

Lemma 7.1 (Farkas' Lemma)**.** *Given a matrix A and vector b, there exists some $x \geq 0$ satisfying $Ax = b$ if and only if for any vector y, $yA \geq 0$ implies $yb \geq 0$.*

It follows that given any linear programming problem, there is an easy proof of the feasibility (exhibit a feasible point) *or* non-feasibility (exhibit a witness y with $yA \geq 0$ but $yb < 0$) of the solution.

Proof. If there exists a solution x such that $Ax = b$, then consider a row vector y with $yA \geq 0$. Since $yb = y(Ax) = (yA)x$, and since by hypothesis $yA \geq 0$ and $x \geq 0$, it follows that $yb \geq 0$. On the other hand, suppose there is no solution. Consider A as a row of column vectors $a_1, \ldots, a_m$ and consider the cone generated by the a_i. Since there is no solution, it follows that b is not in the cone generated by the a_i (since this is what $Ax = b$ with $x \geq 0$ means). Using the Fundamental Theorem of Linear Inequalities proved earlier, we know there exists some y such that $ya_i \geq 0$ for every i (so that $yA \geq 0$), but such that $yb < 0$. ∎

Lemma 7.2 (Farkas' Lemma, alternate form)**.** *Given a matrix A and a vector b, there exists some vector x satisfying $Ax \leq b$ if and only if for any vector $y \geq 0$, $yA = 0$ implies $yb \geq 0$.*

Proof. Examine the condition $\exists x : Ax \leq b$. Although x is not necessarily non-negative, we can rewrite any x such that $x = x_1 - x_2$ with $x_1, x_2 \geq 0$. Therefore there exists a solution to $[A]\,[x] \leq b$ if and only if there exists a solution to

$$\begin{bmatrix} A & -A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq b, \text{ where } x_1, x_2 \geq 0$$

Since the right side of the equation dominates, we can create an equality by adding a non-negative vector x_3 to the left side. These variables are called *slack variables.*

$$\underbrace{\begin{bmatrix} A & -A & I \end{bmatrix}}_{A'} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = b, \text{ where } x_1, x_2, x_3 \geq 0$$

The existence of x such that $Ax \leq b$ is equivalent to the existence of non-negative $x' \geq 0$ such that $A'x' = b$. By Lemma 7.1,

$$\begin{aligned} (\exists x' \geq 0 : A'x' = b) &\Leftrightarrow (\forall y : yA' \geq 0 \Rightarrow yb \geq 0) \\ &\Leftrightarrow (\forall y : yA \geq 0, -yA \geq 0, y \geq 0 \Rightarrow yb \geq 0) \\ &\Leftrightarrow (\forall y : yA = 0, y \geq 0 \Rightarrow yb \geq 0) \end{aligned}$$

∎

7.2 Standard form of LP Duality Theorem

To solve a linear program $max\{\, cx : Ax \leq b \,\}$, we create a second linear program inherently related to the first. We will call the original program the *primal* program, and we will create what is called the *dual* program. For the above linear program, the dual is $min\{\, yb : yA = c, y \geq 0 \,\}$. The importance of the dual is brought out by the following theorem

Theorem 7.3 (Duality Theorem)**.** *Given matrix A and vectors b and c,* $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$ *so long as both sets are not empty.*

Proof. Proving the duality theorem is equivalent to proving the following two conditions hold:

(1) For all feasible x and y, $cx \leq yb$.

(2) There exists some feasible x^* and y^*, such that $cx^* \geq y^*b$.

Condition (1) follows directly from the definition of the primal and dual constraints. Since $c = yA$ then $cx = yAx$ for any feasible x and y. Also, since $Ax \leq b$ and $y \geq 0$, then $yAx \leq yb$. Combining these two, $cx \leq yb$.

Proving condition (2) is the more difficult feat. We will use Farkas' lemma to do so. We want to show the existence of an x and y such that all of the following conditions hold: $Ax \leq b$, $y \geq 0$, $yA = c$, and $cx \geq yb$. These equations can be combined into the following linear system:

$$\underbrace{\begin{bmatrix} A & 0 \\ -c & b^t \\ 0 & A^t \\ 0 & -A^t \\ 0 & -I \end{bmatrix}}_{Q} \begin{pmatrix} x \\ y^t \end{pmatrix} \leq \underbrace{\begin{pmatrix} b \\ 0 \\ c^t \\ -c^t \\ 0 \end{pmatrix}}_{\beta}$$

We must show that there exists a solution to this system. To apply Lemma 7.2 we must show that for all $\gamma \geq 0 : \gamma Q = 0 \Rightarrow \gamma\beta \geq 0$. For $\gamma = \begin{pmatrix} u & \lambda & v & w & q \end{pmatrix}$, the condition $\gamma Q = 0$ is,

$$\begin{pmatrix} u & \lambda & v & w & q \end{pmatrix} \begin{bmatrix} A & 0 \\ -c & b^t \\ 0 & A^t \\ 0 & -A^t \\ 0 & -I \end{bmatrix} = 0$$

Using Farkas' lemma, to show the existence of a solution to this problem, we must show that the condition ($uA - \lambda c = 0$ and $\lambda b^t + (v - w)A^t \geq 0$) implies ($ub + (v - w)c^t \geq 0$). To do so, we consider two cases.

- Case 1: $\lambda > 0$. Using the two assumptions, we can write:

$$ub = \lambda^{-1}(\lambda b^t)u^t \geq \lambda^{-1}(w - v)A^t u^t = \lambda^{-1}(w - v)\lambda c^t.$$

  Thus $ub + (v - w)c^t \geq 0$.

- Case 2: $\lambda = 0$. Assume that $\gamma \geq 0, \gamma Q = 0$ does not imply $\gamma\beta \geq 0$ for all γ. Then there exists such a γ so that $uA = 0$, $(v-w)A^t \geq 0$, and $ub + (v-w)c^t < 0$. However, if we consider this same system without the constraint that $cx \geq yb$, together with this same γ without the λ component, we can apply Farkas' Lemma to show that even this reduced system is infeasible. Observe that this system is just the combination of the primal and dual systems, and so hence its infeasibility implies that either primal or dual is infeasible, contradicting the assumptions.

For either case, we proved that for all $\gamma \geq 0 : \gamma Q = 0 \Rightarrow \gamma\beta \geq 0$, and so by Farkas' lemma, we have shown the existence of such solutions x^* and y^*. ∎

7.2.1 Alternate Forms of LP Duality

There are similar duality theorems for various other forms of linear programs. These allow many forms of primal programs to be translated into related dual programs.

Theorem 7.4 (Alternate Duality Theorems)**.** *Given matrix A and vectors b and c, the following equalities hold, whenever the involved minimums and maximums exist.*

- $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$
- $\max\{cx \mid x \geq 0, Ax \leq b\} = \min\{yb \mid y \geq 0, yA \geq c\}$
- $\max\{cx \mid x \geq 0, Ax = b\} = \min\{yb \mid yA \geq c\}$

Proof. The first form is exactly that which was proven in the original theorem. Given the first equation, to derive the second equality, we must introduce the constraint $x \geq 0$. Apply the first equation with a modified A':

$$\underbrace{\begin{bmatrix} A \\ -I \end{bmatrix}}_{A'}(x) \leq \underbrace{\begin{pmatrix} b \\ 0 \end{pmatrix}}_{b'}$$

Notice that $\{x \geq 0, Ax \leq b\} = \{A'x \leq b'\}$, and so if the first has a feasible solution so does the second. Now we can apply the first duality statement to get that:

$$\max\{cx \mid A'x \leq b'\} = \min\{y'b' \mid y' \geq 0, y'A = c\},$$
$$y' = \begin{pmatrix} y \\ y_0 \end{pmatrix}$$

This program is equivalent to $\min\{yb \mid y \geq 0, yA \geq 0\}$, and so we have the second form of duality.

The third version is exactly symmetric to the first in terms of x and y. We could also derive it from the second considering an appropriate matrix A'. ∎

The above theorem deals with the case where both the primal and the dual problems are feasible. What can we say about the case where all we are told is that the primal is feasible and that the optimum value is bounded ? Now we will show that in this case the dual is feasible and there exists a dual feasible solution that proves that the primal optimum (max) is bounded.

Theorem 7.5. *Let $Ax \leq b$ have at least one solution. Suppose $cx \leq \delta$ holds for all vectors x lying in $Ax \leq b$. Then there exists $\delta' \leq \delta$ such that $cx \leq \delta'$ is a non-negative combination of inequalities in $Ax \leq B$. That is there exists a vector y which contains the coefficients for this combination, such that $yA = c$, $yb = \delta'$, and $y \geq 0$.*

Proof. ¿From the duality theorem, if both primal and dual are feasible, we know that $\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0, yA = c\}$, and this theorem is satisfied by any such optimal dual solution.

To derive a contradiction, assume that there is no feasible solution to the dual problem. Then by applying Farkas' lemma (Theorem 7.1), there must exist some vector z, such that both $Az \leq 0$, and $cz > 0$. Consider the effect of adding z to a feasible primal solution. (We have assumed existence of such primal solution.) Since $Az \leq 0$, adding z will not violate any of the constraints, and so it will still be a feasible solution to the primal. Since $cz > 0$, the cost value of this new primal solution will be strictly greater after adding z. And since we can continually add in more copies of z, then the primal problem cannot have any δ which bounds the value of the optimum primal solution. Therefore if the primal problem is non-empty and bounded, then the dual problem must have some feasible solution.

Since both problems have feasible solutions, then by the duality theorem, there is some optimal dual solution y^* such that $y^*A = c$, $y^* \geq 0$, and $y^*b = \delta'$ where δ' is the maximal cost value of the primal. This y^* satisfies this theorem. ∎

Given a primal program and its dual, the above reasoning and the duality theorem ensures us that there are only four possible scenarios.

1. The primal and dual programs are both non-empty and bounded, and thus the optimal values are equal.
2. The primal is non-empty but unbounded, and thus the dual is empty, in which case the optimal value for both problems is $+\infty$.
3. The dual problem is non-empty but unbounded, and thus the primal is empty, in which case the optimal value for both problems is $-\infty$.
4. Both the primal and dual problems are empty, and thus the optimal values for both are undefined.

7.3 Complementary Slackness

Earlier, we introduced the idea of a slack variables to turn inequalities to equalities. For instance, if $Ax_0 \leq b$, we added a non-negative slack variable x_1, so that $Ax_0 + x_1 = b$. The following theorem relates the value of the slack variables $(b - Ax_0)$ of a solution to the corresponding dual variables.

Theorem 7.6 (Complementary Slackness)**.** *Given the primal linear program $\max\{cx \mid Ax \leq b\}$ and the dual $\min\{yb \mid y \geq 0, yA = c\}$, and given feasible solution: x^* and y^*, the following are equivalent:*

(1) x^* and y^* are optimum.

(2) $cx^* = y^*b$.

(3) $\forall i : (y_i^* > 0 \rightarrow a_i x^* = b_i)$.

Proof. The equivalence of statements (1) and (2) is exactly the duality theorem. So assume statement (2), $cx^* = y^*b$. Since $c = y^*A$, this equality can be written, $y^*Ax^* = y^*b$, and so $y^*(Ax^* - b) = 0$. Since both $y^*, (Ax^* - b) \geq 0$, then each component in the dot product is non-negative. For the sum to be equal to 0, each component must have one (or both) of y_i^* and $(a_i x^* - b_i)$ equal to 0. Therefore, for any i, if $y^* > 0$, then $(a_i x^* = b_i)$. Therefore (2) implies (3).

To see that (3) implies (2), assume that $\forall i : (y_i^* > 0 \rightarrow a_i x^* = b_i)$. Clearly, if $y_i^* = 0$, then $y_i^* a_i x^* = y_i^* b_i$. Also if $y_i^* > 0$, then by the assumption $a_i x^* = b_i$, and thus $y_i^* a_i x^* = y_i^* b_i$. So $\forall i : y_i^* a_i x^* = y_i^* b_i$. Since this is true for all i, $y^*Ax^* = y^*b$, and since $c = y^*A$, then $cx^* = y^*b$. Therefore (3) implies (2), and all the statements are equivalent. ∎

Carefully consider the complementary slackness theorem. Given an optimal primal and dual solution, that theorem states that if the optimum value of the dual variable y_i is strictly greater than zero, then the corresponding constraint must be tightly satisfied by any optimum primal solution. However the converse is not necessarily true. That is, if a constraint is tight, the corresponding dual variable might still equal zero. The following generalized theorem shows that if the constraint is tight, even though the dual variable might still equal zero, there must exist *some* optimal dual solution whose corresponding variable will be positive.

Theorem 7.7. *Given the primal linear program* $\max\{cx \mid Ax \leq b\}$ *and the dual* $\min\{yb \mid y \geq 0, yA = c\}$*, if both limits are defined, then exactly one (but not both) of the following statements is true:*

(a) The primal has an optimal solution x^* with $a_i x^* < b_i$.

(b) The dual has an optimal solution y^* with $y_i^* > 0$.

Proof. Homework. ∎

7.4 Intuition behind Duality and Complementary Slackness

To better understand the intuition in the previous theorem, let's view the constraint problem from a physicist's point of view. Each of the hyperplanes that are constraints, are represented as walls. Our solution is a ball floating in space. Assuming the program has some feasible solution, then the ball is always placed inside the set of walls. The objective function is a directional force pushing the ball. The goal is to take all feasible positions, x, and to maximize the objective function over the set.

We know that in the physical model, the objective function will be maximized when it is against a face, or in a corner. At this point the object will stop moving, because the objective force will be exactly balanced by the combined forces of the walls that the object is up against.

In Figure 6, the a_i's represent the directions of the constraints, and the vector c represents the objective function. At the optimal point, the force c will exactly be balanced by the restraining forces of the walls. Since the walls only provide a force in the direction opposite of the a_i's, then $-c + \sum_i y_i(a_i) = 0$, where $y_i \geq 0$. In other words, c can be expressed as a non-negative combination of the rows of A. That is $c = y_1 a_1 + ... + y_m a_m$ where $y_i \geq 0$. These coefficients correspond to the dual variable y. Furthermore, the intuition of slackness can be seen. The distance between a position x and a wall corresponds to the slackness at that constraint. This value is equal to $b - a_i x$. In the physical model, a specific wall can only provide a balancing force if the position x is up against the wall, that is if the system is tight at that constraint. Therefore, if $y_i > 0$, then that wall is providing some non-zero balancing force, and therefore the position x must be against that wall. In terms of the complementary slackness conditions,

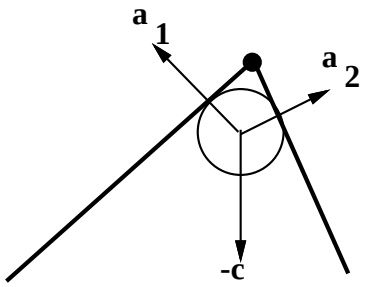


Figure 6: The Physicist's View

$\forall i, (y_i > 0 \Rightarrow a_i x = b_i)$. However it is possible that the ball could be up against a wall that is not contributing any force. If the objective function runs parallel to the constraint, the ball could be resting in a corner against that wall, even though the wall is providing no support. In terms of the slackness conditions, it is possible to have both ($y_i = 0$ and $a_i x = b_i$). However by slightly rotating everything, this degeneracy can always be removed.

8 Gonzaga's Interior Point Algorithm

In a linear program, if an optimum exists, it must be at a vertex of the polytope defined by the constraint hyperplanes. A natural approach to solving a program is to visit the vertices in some systematic order—this is what the simplex algorithm does, for example. A problem is that there are more than polynomially many vertices, so any algorithm that might visit many of them may be slow in some cases.

This problem motivates *interior point methods* that stay within the interior of the convex body defined by the linear program, without visiting every vertex. These methods move toward the optimum while maintaining their distance from the boundary. This is analogous to a ship sailing near a shoreline looking for a particular city: A ship need not traverse the depths of each fjord, but may examine them from afar. Interior point algorithms achieve this goal by defining a function that includes a *barrier* term that keeps the algorithm "away from the edges."

8.1 Definitions & Notation

We are given a standard linear program. The bars in $\bar{c}$ and $\bar{A}$ distinguish "original" values from the "scaled" values that will arise in the algorithm.

$$\begin{aligned} \min\ & \bar{c}x & & \\ \text{s.t. } & \bar{A}x & = & b \\ & x & \geq & 0 \end{aligned}$$

We make these simplifying assumptions:

- $\bar{A}$ has full rank,
- $(\bar{A}\bar{A}^t)^{-1}$ exists,
- the optimum value of the objective function is zero: $\min \bar{c}x = 0$,
- the feasible set is bounded, and
- an initial feasible point x^0 is available.

Any linear program may be transformed to have these properties, but we do not prove this here.

Our strategy will be to define a potential function that simultaneously measures (1) how close we are to the optimum point, and (2) how close we are to a boundary of the polytope. Then we will use gradient descent to move toward the optimum step by step, without getting too close to the boundaries.

Definition 8.1. $\bar{f}(x) = q \log(\bar{c}x) - \sum_{i=1}^{n} \log x_i$

We aim to decrease the value of the potential function. The $\log(\bar{c}x)$ term decreases as we get closer to the optimum point; recall that by assumption, $\min \bar{c}x = 0$. $-\sum \log x_i$ is the *barrier term* that keeps the algorithm away from the boundaries; it increases as any x_i gets closer to 0. q is a scale factor that makes the two terms proportional.

The gradient is

$$\nabla \bar{f}(x) = \frac{q}{\bar{c}x}\bar{c} - (\frac{1}{x_1}, \ldots, \frac{1}{x_n}).$$

[1] Edited by David Fifield, Jonah Brown-Cohen, Hrysoula Papadakis.

A straightforward application of gradient descent would take $x^{k+1} = x^k - \delta\nabla\bar{f}(x^k)$ for some small δ. However this may result in an x^{k+1} that is outside the boundaries (some x_i^{k+1} is negative), or one that is not feasible ($\bar{A}x^{k+1} \neq b$). To solve these problems we make two further refinements to the algorithm.

To keep each x within the boundaries, we choose a small enough δ at each step. How small δ must be depends on how close x is to the boundaries, and also on the direction of the gradient (we may take bigger steps if we are not heading straight for a boundary). We define an ellipsoid that stays within the boundaries and find the optimal point within it. This is the purpose of the D scaling matrix in the algorithm.

To keep each x feasible, we take a projection of each gradient step. Let $h = -\delta\nabla\bar{f}(x)$. That x is feasible means that $\bar{A}x = b$. So it must also be the case that after taking a step, $\bar{A}(x + h) = b$. This means that $\bar{A}h = \vec{0}$; in other words, that s is in the null space of $\bar{A}$. It suffices to project each step onto the null space of $\bar{A}$; this is what the matrix P does in the algorithm.

8.2 The Algorithm

$k \leftarrow 0$
Repeat:

Scaling:

$D \leftarrow \text{diag}(\frac{1}{x_1^k}, \ldots, \frac{1}{x_n^k})$
$A \leftarrow \bar{A}D^{-1}$
$c \leftarrow \bar{c}D^{-1}$
$y^k \leftarrow Dx^k$

Projection:

$h \leftarrow P(-\nabla f(\vec{1}))$, where $P = (I - A^t(AA^t)^{-1}A)$

Update Current Point:

$y^{k+1} \leftarrow y^k + \lambda h$, where $\lambda = \frac{0.3}{\|h\|}$

Scale Back:

$x^{k+1} \leftarrow D^{-1}y^{k+1}$
$k \leftarrow k + 1$

Until: $\bar{c}x^k < \epsilon$

Note that the function f (as opposed to $\bar{f}$) is defined over the unscaled and unbarred A and c. The vector $\vec{1}$ contains 1 in every position.

8.2.1 Details of Scaling

Let us get some intuition about scaling. It could be the case that if we follow direct gradient descent we will hit the boundary of the polytope very quickly (and so our step size would have to be small). On the other hand if we don't follow gradient descent we may be able to take a very large step, but the potential will not be dropping quite as quickly. The scaling attempts to find a middle ground between these two extreme approaches. We will be taking a step that's in the general direction of the gradient, but one that allows us to take a large step, and decrease our potential function by a constant additive factor. As we will see below, the first order approximation to out progress is product of (negative) gradient and the step. Scaling essentially find a point that maximizes this product.

By the way we have defined the matrix D, at the end of the scaling step,

$$y^k = Dx^k = \vec{1}$$

Further,

$$Ay = \bar{A}D^{-1}Dx = \bar{A}x$$
$$cy = \bar{c}D^{-1}Dx = \bar{c}x$$

Thus, if we now consider the problem of minimizing cy subject to $Ay = b$, we are solving the same linear problem as the original one. The value of the optimum is same as for the original problem, i.e. zero. Further note that $f(y)$ and $\bar{f}(x)$ differ only by an additive factor:

$$\begin{aligned} f(y) &= q\log(cy) - \sum_{i=1}^{n} \log y_i \\ &= q\log(\bar{c}x) - \sum_{i=1}^{n} \log(D_i x_i) \\ &= q\log(\bar{c}x) - \sum_{i=1}^{n} \log D_i - \sum_{i=1}^{n} \log x_i \\ &= \bar{f}(x) - \sum_{i=1}^{n} \log D_i \end{aligned}$$

Therefore it is sufficient to prove a reduction in $f(y)$ at each iteration, which directly translates into the same reduction in $\bar{f}(x)$.

8.2.2 Details of Projection

Let z be any vector. z can be decomposed into two vectors $z = z_1 + z_2$ such that $z_1 \in \text{Null}(A)$, and $z_2 \in \text{Range}(A^t)$. Since z_2 is in the range of A^t, there exists some vector w such that $z_2 = A^t w$. We now solve for z_1 given z and A:

$$\begin{aligned} z &= z_1 + z_2 \\ Az &= Az_1 + Az_2 \\ &= 0 + AA^t w \\ w &= (AA^t)^{-1}(Az) \\ z_1 &= z - A^t w = (I - A^t(AA^t)^{-1}A)z \end{aligned}$$

This is the derivation of the matrix P used in the algorithm. The projection of the gradient makes sure that the new solution y^{k+1} will be a feasible solution.

Another simple observation that will be useful later: let x be some vector in null space of A and let $z = z_1 + z_2$ as above. Then $x^t z_2 = x^t A^t w = 0$, i.e. $x^t z = x^t z_1$.

8.2.3 Current Update

Remember that $y^k = \vec{1}$. So we could have written the update as $y^{k+1} = \vec{1} + \lambda\vec{h}$. The value of λ is set such that $y^{k+1} \geq 0$, and thus remain feasible.

8.3 Analysis

8.3.1 Feasibility

Lemma 8.2. *For all values k, $Ax^k = b$. In other words, the solution remains feasible at all times.*

Proof. Follows from our choice of projection matrix P and the step multiplier λ. In particular, suppose $Ax^{k-1} = b$. Note that $x^k = x^{k-1} + D^{-1}\lambda\vec{h}$ and by choice of projection, $\vec{h}$ is in the kernel of A. When multiplying x^k on the left by A, the second term vanishes and we are left with $Ax^{k-1} = b$. ∎

8.3.2 Progress

We proceed to show that in each iteration of the algorithm the potential function decreases by a constant additive factor (we will show that it decreases by at least 0.1). We begin by proving a technical lemma:

Lemma 8.3. $f(\vec{1} + \lambda\vec{h}) \leq f(\vec{1}) + \lambda(\nabla f(\vec{1}))^t\vec{h} + 2\lambda^2\|\vec{h}\|^2$

Intuitively, this lemma is simply bounding $f(\vec{1} + \lambda\vec{h})$ by its first-order Taylor expansion (the first two terms) plus the degree-two error term.

Proof. We begin with a simple calculus fact: $\log(1+\delta) \geq \delta - 2\delta^2$. We now divide our potential function into two parts to proceed:

$$f(y) = \underbrace{q\log(cy)}_{f_1} - \underbrace{\sum_{i=1}^{n}\log y_i}_{f_2}$$

We will bound both f_1 and f_2 individually. Using the fact above, we can get the following bounds on f_2:

$$\begin{aligned} f_2(1+\lambda\vec{h}) &= \sum_{i=1}^{n}\log(1+\lambda h_i) \\ &\geq \sum_{i=1}^{n}\lambda h_i - 2\sum_{i=1}^{n}\lambda^2 h_i^2 \\ &= \underbrace{f_2(\vec{1})}_{=0} + \lambda\underbrace{(\nabla f_2(\vec{1}))^t}_{=\vec{1}}\vec{h} - 2\lambda^2\|\vec{h}\|^2 \end{aligned}$$

Since f_1 is concave, we can use the first-order Taylor series expansion to obtain

$$f_1(\vec{1} + \lambda\vec{h}) \leq f_1(\vec{1}) + \lambda(\nabla f_1(\vec{1}))^t\vec{h}$$

Putting the two together gives us the desired result. ∎

Now we want to show that we make some progress in every step. We begin by proving that $\|\vec{h}\|$ is bounded below by a constant.

Lemma 8.4. $\|\vec{h}\| \geq 1$.

Proof. Let us denote by y^* the true optimum, so that $cy^* = 0$. Observe that since both y^* and $\vec{1}$ are feasible solutions, $(y^* - \vec{1}) \in \text{Null } (A)$.

$$\begin{aligned} h(y^* - \vec{1}) &= -P(\nabla f(\vec{1}))(y^* - \vec{1}) \text{ By definition of } h \\ &= -\nabla f(\vec{1})(y^* - \vec{1}) \\ &= \left(-\frac{qc}{c\vec{1}} + \vec{1}\right)(y^* - \vec{1}) \\ &= -\frac{q}{c\vec{1}} \underbrace{cy^*}_{=0} + \vec{1}y^* + q - n \\ &= \vec{1}y^* + q - n \end{aligned}$$

The second step follows from the fact that as discussed above, every vector can be decomposed into a sum of two orthogonal components: one in the null space of A and one in the range of A^t. Consider such a decomposition of the gradient vector. When multiplied by $(y^* - 1)$ which is in the null space of A, the second component of the gradient vanishes. Therefore projecting the gradient onto the null space and then multiplying by $(y^* - 1)$ has the same effect as simply multiplying it by $(y^* - 1)$.

Now set $q = n + \sqrt{n}$. Then:

$$\begin{aligned} h(y^* - \vec{1}) &= \sum_{i=1}^{n} y_i^* + \sqrt{n} \\ &\geq \|y^*\| + \sqrt{n} \end{aligned}$$

The last inequality follows since y_i^*s are nonnegative. It then follows that:

$$\|\vec{h}\|\|y^* - \vec{1}\| \geq \|y^*\| + \sqrt{n}$$

Thus,

$$\|\vec{h}\| \geq \frac{\|y^*\| + \sqrt{n}}{\|y^* - \vec{1}\|} \geq \frac{\|y^*\| + \sqrt{n}}{\|y^*\| + \|1\|} = 1$$

∎

The above two lemmas give us the building blocks to the following key theorem. It shows that at every iteration the objective function will decrease by at least 0.1. Remember that in the beginning of the iteration we have normalized the y vector to be $\vec{1}$.

Theorem 8.5. $f(y^{k+1}) - f(\vec{1}) \leq -0.1$

Proof.

$$f(y^{k+1}) = f(\vec{1} + \lambda\vec{h}) \leq f(\vec{1}) + \lambda \underbrace{(\nabla f(\vec{1}))^t \vec{h}}_{=-\|h\|^2} + 2\lambda^2\|\vec{h}\|^2$$

We will want to pick a value for λ that balances the first and second terms. For example, if we pick $\lambda = \frac{0.3}{\|\vec{h}\|}$ we get:

$$f(y^{k+1}) - f(\vec{1}) \leq -0.3\|\vec{h}\| + 0.18 \leq -0.1$$

∎

8.4 Runtime Analysis

Definition 8.6. *Let $L = \Theta(\log(\text{product of all input numbers}))$. In other words, L is some constant factor times the number of bits needed to represent the input.*

We state without proof that given an LP in the form described above, we can find an initial solution x^0 such that $f(x^0) = O(qL)$. We will now give some intuition of how long we need to run the algorithm.

Observe that since all of the values in the input are integers, any vertex point that is not optimal cannot be arbitrarily close to 0 in weight. In particular, since a vertex is by definition an intersection of n hyperplanes, for any non-optimal vertex v_1, cv_1 is greater than $\frac{1}{det}$ where *det* is the determinant of some n by n matrix. By definition, the log of the largest determinant *det* can be bounded by $O(L)$. Therefore, suppose we find a solution with potential less than $-O(qL)$, the above discussion implies that any vertex with a better or equal cost must be optimal.

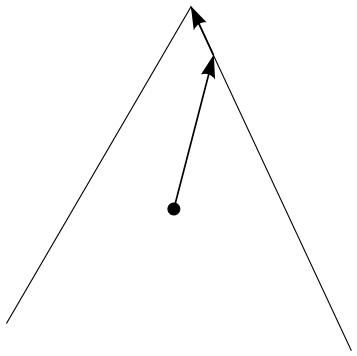


Figure 7: Rounding to Nearby Vertex

Further, given an interior point we can always find a nearby vertex with better or equal cost. For any point that is not a vertex there are always two opposite directions along which you can move without leaving the polytope. Along one of these directions the objective function will be nondecreasing. So simply move in this direction until a constraint becomes tight. At this point, we are in a face of the polytope. We can iterate this process, and at each step we arrive in a lower dimensional face. So after at most n steps, we will have reached a vertex. Since at all times during this process the objective function does not decrease, this vertex will have better or equal cost.

Thus if we have found a solution x^k that has such a low potential, we can simply round it to a vertex as described, and we can do this without increasing the value of the LP solution. The only thing we now consider is the number of iterations to reach such an x^k.

Theorem 8.7. *The algorithm will terminate after at most $O(nL)$ iterations. Further, each iteration can be implemented in $O(n^3)$ in a straightforward manner.*

Proof. We have claimed that in the beginning $f(x^0) = O(nL)$. Further, we can terminate and round to a vertex when $f(x^k) = -O(nL)$. Since we set $q = O(n)$ and decrease the potential function by at least 0.1 at every iteration, we know that the number of iterations is limited by $O(nL)$. Further, each iteration takes several matrix multiplication steps, and can be implemented in $O(n^3)$. ∎

Finally, we note that each iteration's runtime can be sped up a little bit, such that it takes $O(n^2)$. This would make the overall running time of the algorithm $O(n^3L)$. Also, observe that the Gonzaga

method only uses the primal formulation. One can get another $\sqrt{n}$ factor of improvement by considering the dual concurrently with the primal.

9 The Ellipsoid Method

Review

The ellipsoid method, and the modified simplex method presented in this section, solves the problem of *strict linear inequalities* (SLI), which asks to find an x satisfying $Ax < b$, where A is an $m \times n$ matrix. We shall see that SLI is polynomially time equivalent to LP. A polynomial time solution to SLI therefore also solves LP in polynomial time. This improves on the theoretical running time of the classical simplex method, which has a worst case exponential running time. It is important to note here that solving the LP $\{max\ cx | Ax \leq b\}$ is equivalent to just solving the linear constraints $\{Ax \leq b, -cx \leq -\alpha\}$ and then performing a binary search over α to find the optimal solution.

9.0.1 Historical Background

In 1979 the Soviet mathematician L.G. Khachian [8] developed the so-called **ellipsoid method**, originally devised for nonlinear nondifferentiable optimization [11, 20, 22]. Since the ellipsoid method was the first polynomial-time algorithm for linear programming, it received a great deal of attention not only in the scientific literature, but also in the nonscientific press. The ellipsoid method even appeared in the New York Times: *"An Approach to Difficult Problems,"* (Nov. 27, 1979) - see [15].

Based on work of mathematicians such as Levin [11], Shor [20], and Yudin et. al. [22] in the USSR, Yamnitsky and Levin, inspired by Khachian's ellipsoid method, published an algorithm [21], in which simplices are used instead of ellipsoids. This algorithm is somewhat slower than the ellipsoid method, but it is simpler and seems to have the same theoretical applicability.

9.0.2 SLI $\equiv$ LP

We follow Papadimitriou et. al. [15] pp. 172, when reviewing why SLI is polynomially equivalent to LP. Recall that LP is the problem: max $c'x$, subject to $Ax \leq b$ and $x \geq 0$. The proof is in two stages. First one verifies that LP is polynomially equivalent to *linear inequalities* (LI), which asks to find *just one* feasible solution to $Ax \leq b$. Second, one establishes that there is an $\epsilon > 0$, such that LI: $a_i x_i \leq b_i, 1 \leq i \leq m$ has a solution, *iff* the system of strict linear inequalities (SLI): $a_i x_i < b_i + \epsilon, 1 \leq i \leq m$ has a solution. Combining the two facts yields the desired result.

Verifying the first equivalence is mainly a matter of solving LP by iterating LI and binary search to obtain an optimal solution (See [15], Theorem 8.2 for detail.).

We cover the equivalence between SLI and LI in more detail. Let ϵ satisfy

$$0 < \epsilon < (nB)^{-(n+1)}$$

where B is bounded by the maximal size integer in A and b. Then we show

$$a_i x \leq b_i, 1 \leq i \leq m \quad \text{is feasible} \quad \text{iff} \quad a_i x < b_i + \epsilon, 1 \leq i \leq m \quad \text{is feasible}$$

Obviously, since ϵ is non-negative, LI $\Rightarrow$ SLI. For the other direction, let x_0 be a solution of SLI, and collect in the set I, those coordinates, where the reverse implication might be violated:

$$I = \{i \mid b_i \leq a_i x_0 < b_i + \epsilon\}$$

ie., I is the set of problematic indices, where SLI is feasible, but LI might not be feasible. We will now establish that if LI is feasible, then there exists a point x_0 such that the a_i, for $i \in I$ span A [2]. To show this, start with some x_0 and assume that there exists an a_j, $j \notin I$, independent of all $a_i, i \in I$. Take this a_j, where we know that $a_j x_0 < b_j$, and let z satisfy

$$a_j z = 1 \text{ and } a_i z = 0 \forall i \in I$$

Such a z always exists since a_j is independent of all $a_i, i \in I$, so we can choose z as the perpendicular vector to the plane spanned by $a_i, i \in I$ and normalize it to get the above relation correctly.

Now update x_0 to $x_0 + \lambda z$. Now increase λ starting from 0, until some new k enters the set I. Notice that k may be different than j, but now either a_j can be expressed in terms of the updated I (= old $I \cup \{k\}$), or one can repeat the process noting that we run out of rows.

It is therefore possible to express every a_j, $j \notin I$ via a linear combination of the vectors formed by indices in I.

Let $I' \subseteq I$ be a basis that spans all rows of A. By Cramer's rule, to solve a system of equalities $Ax = b$, we need to set $x_i = D_i/D$, where D_i is determinant of A and D_i is determinant of a matrix obtained by replacing ith column of A by vector b. To apply this to our case, let A' by the (full rank) matrix formed by columns $a_i, i \in I'$. To solve $A'\beta_j = a_j$ we can set the i-th coordinate of β_j to be $\beta_{ij} = \frac{D_{ij}}{|D|}$, where D is determinant of A' and D_{ij} is determinant of matrix obtained from A' by replacing column i by vector a_j.

Therefore we have:

$$a_j = \sum_{i \in I'} \beta_{ij} a_i, \quad \text{where } \beta_{ij} = \frac{D_{ij}}{|D|} \quad \text{is found by Cramer's Theorem}$$

Now let $\hat{x}$ satisfy

$$a_i \hat{x} = b_i, \quad i \in I'$$

Such an $\hat{x}$ exists as all a_i,$i \in I'$ are independent vectors and hence the matrix formed will be full rank. Observe that, by Cramer's rule, all coordinates of $\hat{x}$ are rational with common denominator D.

Then for every $j \notin I'$ one can establish:

$$\begin{aligned}
|D|(a_j \hat{x} - b_j) &= \sum_{i \in I'} D_{ij} \underbrace{a_i \hat{x}}_{b_i} - |D| b_j \\
&= \sum_{i \in I'} D_{ij} b_i - |D| b_j \\
&= -\sum_{i \in I'} D_{ij} \underbrace{(a_i x_0 - b_i)}_{\leq 0} + |D| \underbrace{(a_j x_0 - b_j)}_{< \epsilon} \\
&< \epsilon |D| \\
&\leq 1
\end{aligned}$$

One before last inequality follows from the fact that j might be in I but not in I'. The last inequality follows by the choice of ϵ.
Since the denominators in the rational vector $\hat{x}$ all divide D, $|D|(a_j \hat{x} - b_j)$ is an integer, which, by the derivation above, is non-positive. Hence:

$$a_j \hat{x} \leq b_j \quad \text{for every } j$$

[2] We will assume that rows of A span the whole space, otherwise we can restrict our attention to the lower dimensional space spanned by these rows.

which shows that $\hat{x}$ is a solution to LI.

9.1 The Yamnitsky-Levin (Modified Ellipsoid) Algorithm for SLI

The algorithm of Yamnitsky and Levin [1982] solves SLI. Our presentation of this algorithm follows Chvátal's book [1].

The algorithm maintains a simplex that contains the feasible polytope. At each iteration the algorithm either finds a feasible solution and terminates or replaces the current simplex with one of a smaller volume. If the volume of the simplex becomes too small, the algorithm concludes that the problems is infeasible.

Step 1. $S \leftarrow$ enclosing simplex.
Step 2. If *center(S)* is feasible then return else consider the corresponding half-simplex $S_{\frac{1}{2}}$ find a smaller simplex $\tilde{S}$ containing $S_{\frac{1}{2}}$ $S \leftarrow \tilde{S}$
Step 3. If *vol(S)* is *"small"* then return (*"infeasible"*) else repeat **Step 2.**

9.2 Definitions

The terminology used in the algorithm above is explained by the following definitions.

Definition 9.1. *A set $S \subset \Re^n$ is a* convex hull *if S is the intersection of all convex sets containing S.*

Definition 9.2. *Points $v_0, v_1, ..., v_n$ are said to be in* general position *if*

- *the vectors $v_1 - v_0$,..., $v_n - v_0$ are linearly independent,*
- *dim $V(v_0, v_1, ..., v_n) = n$,*
- *no hyperplane passes through all of them.*

Definition 9.3. *An* n-simplex *is the convex hull of $n+1$ points in general position.*

Definition 9.4. *A set of vectors is a (convex)* polytope *if it is the convex hull of finitely many vectors.*

Definition 9.5. *Suppose S is an n-simplex in $\Re^n$, then its center is defined by*

$$center(S) = \frac{\sum_{i=0}^{n} v_i}{n+1}$$

and its volume by

$$vol(S) = \frac{\det(A)}{n!} \quad \text{where } A = \begin{pmatrix} v_1 - v_0 \\ v_2 - v_0 \\ \vdots \\ v_n - v_0 \end{pmatrix}.$$

Definition 9.6. *A* half-simplex $S_{\frac{1}{2}}$ *is the intersection of a simplex* S *with half-space whose bounding hyperplane passes through the center of* S.

Note that $S_{\frac{1}{2}}$ is not necessarily a simplex according to definition 9.3.

Exercise: How can we find the initial simplex of volume $n^{n^{O(1)}}B^{n^{O(1)}}$ (here B is the largest integer in the input)?

Exercise: Show that if an enclosing simplex has volume $n^{-n^{O(1)}}B^{-n^{O(1)}}$, then the feasible polytope is empty.

9.3 Key Lemma

We establish polynomial time termination of the algorithm using the lemma presented in this paragraph.

Lemma 9.7. *Let* $S_{\frac{1}{2}}$ *be a half-simplex of* S*, then there exists a simplex* $\tilde{S} \supseteq S_{\frac{1}{2}}$ *such that* $r = \frac{vol(\tilde{S})}{vol(S)} < e^{-\frac{1}{2n^2}+\frac{1}{6n^3}}$.

Proof. Let S be the convex hull of points $v_0, v_1, ...v_n$ and let $S_{\frac{1}{2}}$ be the intersection of S with half-space $a'x \leq b$. Define the affine transformation $e(x) = b - a'x$ for all points x. Let k be the subscript that maximizes $e(v_k)$. Since $S_{\frac{1}{2}} \neq \emptyset$, we know $e(v_k) > 0$. Define points $\tilde{v_0}, \tilde{v_1}, ..., \tilde{v_n}$ by

$$\tilde{v_i} = v_k + \frac{v_i - v_k}{d_i}, \quad \text{with } d_i = 1 - \frac{e(v_i)}{n^2 e(v_k)}.$$

We show that the convex hull $\tilde{S}$ of $\tilde{v_0}, \tilde{v_1}, ..., \tilde{v_n}$ has the desired properties.

That $\tilde{S}$ is an n-simplex is seen by inspection of definition 9.3.

To establish that $\tilde{S} \supseteq S_{\frac{1}{2}}$, pick $x \in S_{\frac{1}{2}}$. Since $x \in S$, $x = \sum_{i=0}^{n} t_i v_i$ for some t_i such that $\sum_{i=0}^{n} t_i = 1$ and $0 \leq t_i \leq 1$. Define

$$\tilde{t_i} = \begin{cases} d_i t_i, & i \neq k, \\ d_k t_k + \frac{e(x)}{n^2 e(v_k)}, & i = k. \end{cases}$$

Since $e(x) = \sum_{i=0}^{n} t_i e(v_i)$,

$$\begin{aligned} \sum_{i=0}^{n} \tilde{t_i} &= \sum_{i=0}^{n} d_i t_i + \frac{e(x)}{n^2 e(v_k)} \\ &= \sum_{i=0}^{n} t_i - \frac{\sum_{i=0}^{n} t_i e(v_i)}{n^2 e(v_k)} + \frac{e(x)}{n^2 e(v_k)} \\ &= \sum_{i=0}^{n} t_i \\ &= 1. \end{aligned}$$

Since $e(v_i) \leq e(v_k)$, we have $0 \leq \tilde{t_i} \leq 1$, hence the $\tilde{t_i}$ are within the proper bounds, they also establish

that $x \in \tilde{S}$, i.e., $\sum_{i=0}^{n} \tilde{t_i}\tilde{v_i} = x$, since

$$\begin{aligned}
\sum_{i=0}^{n} \tilde{t_i}\tilde{v_i} &= \sum_{i=0}^{n} d_i t_i (v_k + \frac{v_i - v_k}{d_i}) + \frac{e(x)}{n^2 e(v_k)} v_k \\
&= \underbrace{\sum_{i=0}^{n} v_i t_i}_{x} + \underbrace{v_k \sum_{i=0}^{n} t_i d_i + \frac{e(x)}{n^2 e(v_k)} v_k}_{v_k} - v_k \underbrace{\sum_{i=0}^{n} t_i}_{1} \\
&= x.
\end{aligned}$$

Thus $x \in \tilde{S}$.

Next we show that the ratio r between the volume of $\tilde{S}$ and the volume S is less than $e^{-\frac{1}{2n^2}+\frac{1}{6n^3}}$. Note that $\tilde{v_k}= v_k$, that each $\tilde{v_i}$ with $i \neq k$ lies on the line passing through v_k and v_i, and that the distance from v_k to $\tilde{v_i}$ equals the distance from v_k to v_i divided by d_i. Therefore, $r = \prod_{i \neq k} \frac{1}{d_i}$. Since $e(v_i) \leq e(v_k)$, we have $d_i \geq 1 - \frac{1}{n^2}$. For $c = center(S)$, $e(c) = 0$. We have

$$\begin{aligned}
\sum_{i \neq k} d_i &= \sum_{i \neq k} (1 - \frac{e(v_i)}{n^2 e(v_k)}) \\
&= n - \frac{1}{n^2 e(v_k)} \sum_{i \neq k} e(v_i) \\
&= n - \underbrace{\frac{(n+1)e(c)}{n^2 e(v_k)}}_{0} + \underbrace{\frac{e(v_k)}{n^2 e(v_k)}}_{\frac{1}{n^2}} \\
&= n + \frac{1}{n^2}.
\end{aligned}$$

We want to minimize a product of variables subject to (a) a positive lower bound on each variable, and (b) the sum of the variable equal to a prescribed number. It is easy to show that at most one variable exceeds its lower bound in every optimal solution. Therefore

$$\begin{aligned}
\prod_{i \neq k} d_i &\geq (1 - \frac{1}{n^2})^{n-1}(1 + \frac{1}{n}) \\
&= (1 - \frac{1}{n^2})^n \frac{1 + \frac{1}{n}}{1 - \frac{1}{n^2}} \\
&= (1 - \frac{1}{n^2})^n (1 - \frac{1}{n})^{-1}
\end{aligned}$$

Taking logarithm, we get

$$\begin{aligned}
\lg\left((1 - \frac{1}{n^2})^n (1 - \frac{1}{n})^{-1}\right) &= n \lg(1 - \frac{1}{n^2}) - \lg(1 - \frac{1}{n}) \\
&= -n \sum_{i=1}^{\infty} \frac{1}{i \cdot n^{2i}} + \sum_{i=1}^{\infty} \frac{1}{i \cdot n^i} \\
&> \frac{1}{2n^2} - \frac{1}{6n^3}.
\end{aligned}$$

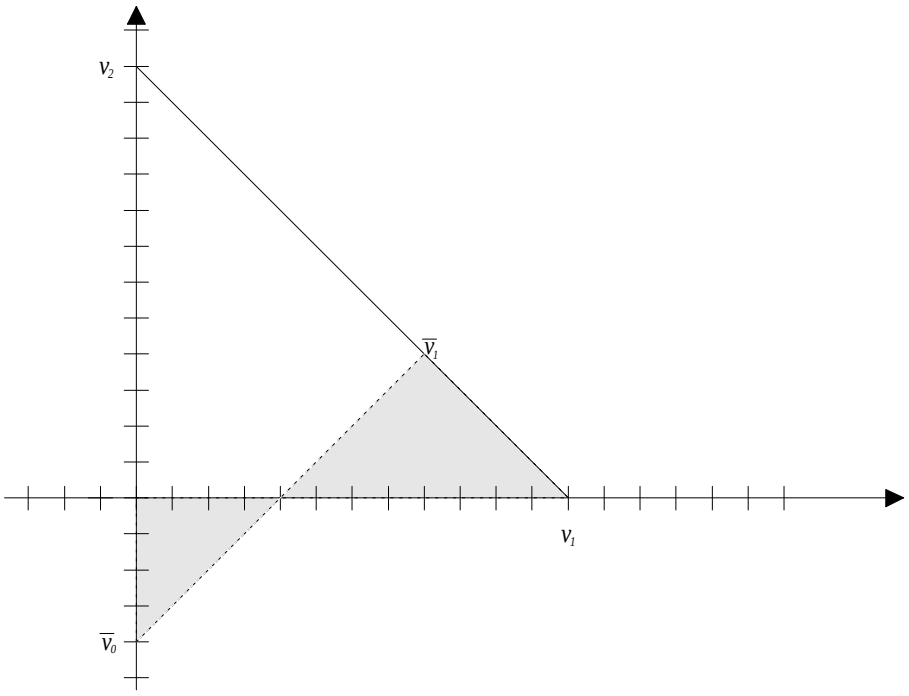
Therefore $\prod_{i \neq k} d_i > e^{\frac{1}{2n^2} - \frac{1}{6n^3}}$, or

$$r = \prod_{i \neq k} \frac{1}{d_i} < e^{-\frac{1}{2n^2} + \frac{1}{6n^3}}.$$

∎

9.4 Example

The following example illustrates how the algorithm works.



The algorithm is given the simplex S, with corners $v_0 = (0,0), \quad v_1 = (6,0)$, and $v_2 = (0,6)$. An oracle or some other device tells that the feasible solution within the simplex must satisfy $x \leq 2$. As we required, the hyperplane $x = 2$ intersects the center (2,2).

The hyperplane gives rise to the affine transformation $e(x,y) = 2 - x$, which has a maximum in $v_k = (0,6)$, i.e., $k = 2$. Plugging in to the definition of $\tilde{v}$'s give the values:
$\tilde{v}_0 = (0,6) + \frac{1}{1 - \frac{2}{2^{2 \cdot 2}}}[(0,0) - (0,6)] = (0,-2)$,
$\tilde{v}_1 = (0,6) + \frac{1}{1 - \frac{-4}{2^{2 \cdot 2}}}[(6,0) - (0,6)] = (4,2)$,
$\tilde{v}_2 = (0,6)$.

The simplex $\tilde{S}$ determined by $\tilde{v}_0, \tilde{v}_1$ and $\tilde{v}_2$ clearly contains $S_{\frac{1}{2}}$, which is delimited by $(0,0), (0,6), (2,4), (2,0)$. We also have that $\frac{vol(\tilde{S})}{vol(S)} = \frac{8}{9} < e^{-\frac{1}{2 \cdot 2^2} + \frac{1}{6 \cdot 2^3}}$ as expected.

9.5 Examples of using separation oracle

The Levin-Yamnitsky and Ellipsoid algorithms have an important property: all that is required to be able to run the algorithm (and solve the LP) is to be able to take a point x and, if it is not feasible, produce a hyperplane separating the feasible region from x. Such a subroutine is commonly called a "Separation Oracle". The implication is that even if the LP is given implicitly and has an exponential number of inequalities, we can still solve it in polynomial time as long as we can provide a polynomial time separation oracle. Following are several simple examples.

Minimum Cost Arborescence. Given a directed graph with positive cost edges $c(e)$ and a special node $r \in E$, find a min-cost tree rooted at r which reaches each of the nodes in the graph.

There exists a combinatorial algorithm to solve the minimum cost arborescence problem, which relies on solving the following linear program with a primal-dual method:

$$\begin{aligned} \min & \sum_{e \in E} c(e)x(e) \\ & \sum_{e \in \delta^-(S)} x(e) \geq 1 \qquad \forall S \subseteq V - \{r\} \\ & x(e) \geq 0 \end{aligned}$$

where $\delta^-(S)$ is the collection of edges entering the set of nodes S.

Instead, we could use the Ellipsoid algorithm to directly solve this linear program in polynomial time. Even though the number of constraints is exponential (since the number of sets S is exponential), we can easily implement a separation oracle. Indeed, consider some candidate point x. If $x(e) < 0$ for some edge $e \in E$, then return that inequality. Otherwise, consider giving each edge e a capacity of $x(e)$. If we have $\sum_{e \in \delta^-(S)} x(e) < 1$ for some $S \subseteq V - \{r\}$, then $(V - S, S)$ gives an r-t cut of value less than 1 for any $t \in S$. In particular, the minimum r-t cut must have value less than 1. Hence, for every $t \in V - \{r\}$, we consider the minimum r-t-cut problem in which the capacity on edge e is given by $x(e)$. This can be solved by one maximum flow computation per vertex, for a total of $n - 1$ applications of the maximum flow algorithm. If for some $t \in V - \{r\}$, the minimum r-t cut has value less than 1, then we have found a violated inequality. Otherwise, x is feasible. Since the above separation oracle can be implemented in polynomial time, the linear program can be solved in polynomial time. In general, the Ellipsoid algorithm guarantees that we find an optimum vertex solution, but makes no guarantee that this vertex will have integer coordinates. Nevertheless, for this particular problem, Edmonds proved that the special structure of the above LP guarantees that the vertices correspond to arborescenses.

Maximum Independent Set. Given a graph $G = (V, E)$, find a subset $S \subseteq V$ of maximum size such that there are no edges between vertices of S.

The integer program for this problem is:

$$\begin{aligned} \max & \sum_i x_i \\ & x_i + x_j \leq 1 \quad \forall (ij) \in E \\ & x_i \in \{0, 1\} \end{aligned}$$

[2]Edited by Jeffrey Chen and Joshua Wang

Relaxing this IP gives a very bad approximation to the integer problem. In particular, for a complete graph on n vertices, taking $x_i = 1/2$ for each i gives a feasible solution to the relaxed problem with value $n/2$, while the maximum independent set can only contain a single vertex.

We can approach this problem by using "cutting planes". The key idea is that the optimum to the linear program may be at a vertex that is far away from any integer solutions. By adding constraints that leave the set of integer solutions unchanged but remove fractional solutions, the optimum solution to the LP will be closer to an integer point.

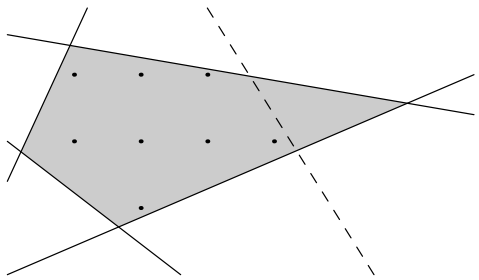


Figure 8: A linear program described by the solid lines, with feasible region shaded and feasible integer points displayed. If the optimum solution is the right vertex, then the addition of a new constraint, the dashed line, moves the optimum much closer to an integer solution.

Specifically, for the maximum independent set problem the following property holds: for each odd cycle C we can add at most $\frac{|C|-1}{2}$ of its vertices to the independent set. This is equivalent to defining a new variable $y_{ij} = 1 - x_i - x_j$ for each edge ij and, for each odd cycle C, adding a constraint $\sum_{ij \in C} y_{ij} \geq 1$. In particular, this property prevents the $x_i = 1/2$ point for complete graphs that we had trouble with earlier.

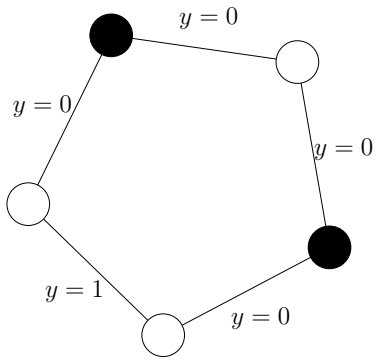


Figure 9: In this odd cycle, we can select at most two vertices, so at least one edge will have $y = 1$.

The resulting LP has an exponential number of constraints (since there might be an exponential number of odd cycles in G), but again it is easy to implement a separation oracle in polynomial time. In particular, all we have to show is that we can compute minimum-cost odd cycle, where edge costs are given by y_{ij}. To this end, we construct an undirected bipartite graph $G' = (V' = V_1 \cup V_2, E')$ such that there exists a vertex $i_1 \in V_1$ and a vertex $i_2 \in V_2$ corresponding to each vertex $i \in V$. For an edge $ij \in E$, we add two edges $i_1 j_2$ and $j_1 i_2$ to E'. Odd cycles in G passing through i correspond to paths from i_1 to i_2 in G'. We can find the shortest path from i_1 to i_2 for each $i \in V$ in polynomial time (and consequently the minimum cost cycle), yielding the desired polynomial time separation oracle.

Note that the solution to the modified LP with cutting planes still does not yield an optimum independent set. However, since this is an NP-hard problem (with approximation hard as well), this was to be expected.

10 Multicommodity flow

The multicommodity flow problem involves simultaneously shipping several different commodities from their respective sources to their sinks in a single network, such that the total amount of flow going through each edge does not exceed its capacity. Each commodity has an associated demand, which is the amount of this commodity that we wish to ship.

In the first section, we show how to formulate this problem as a Linear Program and present its dual, which has a natural interpretation in terms of a length function. We will also consider some variants of the problem such as the Max-Sum flow problem and the concurrent flow problem.

As a linear program, the multicommodity flow problem can be solved in polynomial time using interior point based linear programming algorithms. Algorithms having running times such as $O(\sqrt{km}\,(m \lg n + \lg D \cdot U) \cdot k^2 mn)$ are known. (n, m, k, D and U denote the the number of vertices, edges, commodities, largest demand and capacities respectively.) Such a running time is not satisfactory.

We present an algorithm which provides approximate solutions within any desired accuracy and only requiring time proportional to $kmn\epsilon^{-2}$ with some polylog terms, where ϵ is the sought accuracy.

In the second section, we present the main ideas of this algorithm: we define a potential function which roughly corresponds to the maximum load on any edge of the network. This potential is then minimized using a variant of a gradient descent, where the current flow is modified slightly at each step towards the point corresponding to a superposition of k single commodity minimum cost flow.

The third section is devoted to the details of the algorithm.

10.1 Linear Programming Formulation

10.1.1 Primal Multicommodity Flow

An instance of the *multicommodity flow problem* consists of a directed graph $G = (V, E)$, with nonnegative capacity $Cap(vw)$ for every edge $(vw) \in E$, and k commodities, numbered 1through k. While we deal with the directed case, it is straightforward to extend it to the undirected case. Each specification for commodity i consists of a source-sink pair $s_i, t_i \in V$ and a nonnegative demand d_i. We will denote the number of nodes by n, and the number of edges by m. For notational convenience we assume that $m \geq n$, and that the graph G is connected and has no parallel edges. We assume that the capacities and the demands are integral, and denote the largest capacity by U and the largest demand by D.

A multicommodity flow f consists of a function $f_i(vw)$ on the edges of G for every commodity i, which represents the *flow* of commodity i on edge vw.

A multicommodity flow f is *feasible* if it satisfies the following system of inequalities:

$$\sum_{(s_i v)\in E} f_i(s_i v) - \sum_{(v s_i)\in E} f_i(v s_i) = d_i \qquad \text{for all sources } s_i \tag{6}$$

$$\sum_{(t_i v)\in E} f_i(t_i v) - \sum_{(v t_i)\in E} f_i(v t_i) = -d_i \qquad \text{for all sinks } t_i \tag{7}$$

$$\sum_{(vw)\in E} f_i(vw) - \sum_{(wv)\in E} f_i(wv) = 0 \qquad \text{for all commodities } i, v \in V - \{s_i, t_i\} \tag{8}$$

$$\sum_i f_i(vw) \leq Cap(vw) \qquad \text{for all edges } (vw) \tag{9}$$

$$f_i(vw) \geq 0 \qquad \text{for all commodities } i, \text{ edges } (vw) \tag{10}$$

Equations (6) and (7) guarantee the demands are satisfied, while conservation constraints (8) ensure flow does not accumulate at non-source, non-sink nodes. The different commodities interact only through the capacity constraints (9).

Shahrokhi and Matula [18] study an optimization version of the above problem, which they call the *concurrent flow problem*. Here, the goal is to minimize λ, a capacity scaling factor, such that there exists a feasible flow in the network with capacities scaled by λ:

$$\sum_i f_i(vw) \leq \lambda Cap(vw) \qquad \text{for all edges } (vw) \tag{9'}$$

The original problem is feasible if $\lambda \leq 1$, which corresponds to the original capacity constraints, for the concurrent flow problem. An equivalent formulation is the Max Sum problem, whose objective is to maximize z, a demand scaling factor, such that

$$\sum_{(s_i v)\in E} f_i(s_i v) - \sum_{(v s_i)\in E} f_i(v s_i) = z d_i \qquad \text{for all sources } s_i \tag{6'}$$

$$\sum_{(t_i v)\in E} f_i(t_i v) - \sum_{(v t_i)\in E} f_i(v t_i) = -z d_i \qquad \text{for all sinks } t_i \tag{7'}$$

and the original capacity equation (9) holds. The exercise showing the two formulations are equivalent using $\lambda^* = 1/z^*$ is left to the reader.

10.1.2 Notation

A solution f to the multicommodity flow problem is an array of k individual flows. We call such an array a multiflow, and we denote by f_i the flow of f for commodity i, and by f the edgewise sum of the k flows of f.

Given a cost function ℓ on the edges of G and a multiplier λ, we define $f_i^*(\ell, \lambda)$ to be the minimum cost flow with respect to ℓ that meets the demand for commodity i while respecting the capacity constraints λu. We denote by $f^*(\ell, \lambda)$ the array of the k minimum cost flows, and by $f^*(\ell, \lambda)$ their edgewise. Parameters ℓ and λ are dropped when there is no risk of confusion.

The cost of a flow f_i with respect to a cost function ℓ is denoted by the inner product $\ell \cdot f_i$. Similarly we write

$$\ell \cdot Cap = \sum_{e \in E} \ell(e) Cap(e)$$

which will be interpreted later as the total "volume" of the graph.

10.1.3 Dual Concurrent Multicommodity Flow

To construct the dual, we associate *price* variables $p_i(v)$ with the conservation equations (6), (7), (8) and *length function* variables $\ell(vw)$ with the capacity constraints (9′).

$$\text{maximize} \quad \sum_i d_i(p_i(t_i) - p_i(s_i)) \qquad \text{subject to}$$

$$p_i(v) - p_i(w) + \ell(vw) \geq 0 \qquad \text{for all commodities } i \text{ and edges } (vw) \qquad \ell(vw) \quad \geq 0 \qquad \text{for all edges } (vw)$$

$$\ell \cdot Cap = \sum_{e \in E} \ell(e) Cap(e) = 1 \qquad \text{i.e. } vol_\ell \text{ is } 1$$

Define $dist_\ell(i)$ to be the distance from s_i to t_i under a (non-negative) distance function ℓ. Given any such ℓ, we can construct p_i such that the above equations are satisfied, maximizing the objective function. Note that we can subtract $p_i(s_i)$ from each $p_i(u)$ without changing the above equations, so we can choose $p_i(s_i) = 0$. Thus, $p_i(u)$ represents the distance from s_i to u under ℓ.

Lemma 10.1. *For feasible λ and ℓ, $\sum_i d_i dist_\ell(i) \leq \lambda$.*

Proof. The claim follows from duality, but in order to develop intuition about the algorithm discussed in the next section, it is instructive to prove this claim directly.

Since ℓ is feasible in the dual,

$$\lambda = \sum_{e \in E} \lambda Cap(e) \ell(e)$$

Since cumulative flow on edge e is bounded by $\lambda Cap(e)$,

$$\begin{aligned} \sum_{e \in E} \lambda Cap(e) \ell(e) &\geq \sum_{e \in E} \sum_i f_i(e) \ell(e) \\ &= \sum_i \sum_{e \in E} f_i(e) \ell(e) \end{aligned}$$

Finally, we can get a lower bound on the cost of flow i by ignoring capacity constraints. Without these, we can push all flow along the shortest path between s_i and t_i. Hence, we know that:

$$\sum_i \sum_{e \in E} f_i(e) \ell(e) \geq \sum_i d_i dist_\ell(i)$$

Combining these yields the desired inequality. ∎

In fact, strong duality implies the following.

Theorem 10.2. *If λ^* denotes the optimal primal value and ℓ^* the optimal dual, then* $\sum_i d_i dist_{\ell^*}(i) = \lambda^*$.

Observe that the optimal flow f^* only uses the shortest paths with respect to the optimal length function ℓ^*. Suppose flow $f_i(vw) > 0$. Complementary slackness implies that $\ell(vw) = p_i(w) - p_i(v)$. Hence, the length of any $s_i \rightsquigarrow t_i$ path with positive flow is equal to $p_i(t_i)$, which is equal to the distance to t_i from s_i.

However, we use a different form of inequality for our goal. Define $C_i(\ell, \lambda)$ to be the cost of the flow of commodity i, $\sum_e f_i(e)\ell(e)$ and $C_i^*(\ell, \lambda)$ to be the cost of the optimum min-cost flow f_i^* with respect to ℓ, and the capacities scaled by λ.

Theorem 10.3. *For a multicommodity flow f satisfying capacities λCap and a length function ℓ,*

$$\sum_e \lambda \ell(e) Cap(e) \geq \sum_i \sum_e \ell(e) f_i(e) = \sum_i C_i(\ell, \lambda) \geq \sum_i C_i^*(\ell, \lambda).$$

Furthermore, if ℓ and λ are optimal, there exists a multicommodity flow f (satisfying capacities λCap) for which all of the above terms are equal.

Proof. We can view $\ell(e)$ as the length of an edge and $Cap(e)$ as its cross section. Then, $\sum_e \lambda \ell(e) Cap(e)$ is the volume of the system, and $\sum_i \sum_e \ell(e) f_i(e)$ is the volume of the flow we push. The first inequality just says that the volume of the system is at least the volume of the flow we push. Reinterpreting $\ell(e)$ as cost, the second inequality just says that the cost of our flow is at least the cost of the minimum cost flow f^*.

Formally, the first inequality follows from the feasibility of f. The second equality follows from the definition and the third from optimality of f^*.

Clearly, $\lambda^* \geq \sum_i C_i^*(\ell, \lambda^*) \geq \sum_i C_i^*(\ell, \lambda) \geq \sum_i d_i dist_{\ell^*}(i)$. Combining this with Theorem 10.2, we see that all the terms in the inequalities above must be equal when $\lambda = \lambda^*$ and $\ell = \ell^*$. ∎

10.2 Max-sum flow variant

In the max-sum flow variant, our objective is to maximize the total amount of all commodities shipped. This problem can create "unfair" dependencies since we can drastically change the shipment of different commodities without changing the objective function.

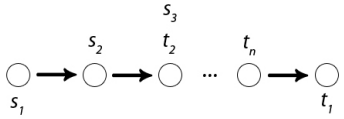


Figure 10: Multicommodity flow example

In Figure 10, each edge has capacity 1. To maximize total flow, we can ship 1 unit for each one of the commodities 2 and above, while not shipping anything of commodity 1.

In the concurrent flow case, suppose we have demands $d_i = 1$ for each commodity i. Here, the optimal solution is $\lambda \approx 2$. Note that these two variants produce very different solutions, so there is also no obvious way to translate solutions between variants.

10.3 Framework for the Multicommodity Flow Algorithm

It appears that multicommodity flow problems are significantly more difficult to solve than single-commodity maximum flow problems. There are several factors that contribute to this. Among them is the fact that the minimum-cut, maximum-flow theorem does not hold for multicommodity flows.

Also, the notion of the "residual graph" that appears to be very useful in the context of single-commodity flows does not seem to extend in a natural way to the multicommodity flow case. Integrality properties do not extend to the solution as in single-commodity flow problems.

Since multicommodity flow problems are linear programs (LPs), we can use interior-point polynomial time LP algorithms to solve them. Unfortunately, this leads to poor asymptotic performance, mostly due to the large number of variables (one per each commodity-edge pair). In many cases it is sufficient to produce *approximate* solutions. An *ϵ-optimal flow f* satisfies capacity constraints λCap and demands d_i such that λ is within a $(1+\epsilon)$ factor of the optimal λ^*. We say λ is ϵ-optimal.

Leighton, Makedon, Plotkin, Stein, Tardos, and Tragoudas [9] presented the first efficient approximation algorithm for multicommodity flow. Their paper shows that an ϵ-optimal flow can be computed in $\tilde{O}(\epsilon^{-2}kmn)$ time by a randomized algorithm. These results were extended to the more general case of packing and covering by Plotkin, Shmoys, and Tardos in [16]. We follow the latter paper.

10.3.1 Approximate Complementary Slackness

The two inequalities of Theorem 10.3 become equalities for an optimal solution. We now show that an ϵ-optimal solution is one for which these inequalities are near equalities (with an appropriate length function ℓ).

Definition 10.4. *Let $\epsilon > 0$ be an error parameter, f a flow satisfying capacities λu, and ℓ a length function. We say that f, λ and ℓ satisfy the* **ϵ-complementary slackness conditions** *if the following hold:*

$$(1-\epsilon)\lambda \sum_e \ell(e)Cap(e) \quad \leq \quad \sum_i C_i(\ell,\lambda) \tag{11}$$

$$(1-\epsilon) \sum_i C_i(\ell,\lambda) \quad \leq \quad \sum_i C_i^*(\ell,\lambda) \tag{12}$$

In order to simplify later proofs, we replace the complementary slackness conditions by the following conditions:

$$(1-\epsilon)\lambda \sum_e \ell(e)Cap(e) \leq \sum_i C_i(\ell,\lambda) \tag{R1}$$

$$\sum_i C_i(\ell,\lambda) - \sum_i C_i^* \leq \epsilon \left(\sum_i C_i(\ell,\lambda) + \lambda \sum_e \ell(e)Cap(e) \right) \tag{R2}$$

The reader can verify that (R1), (R2) are equivalent to (11), (12) up to a constant factor in the error parameter. This follows by adding up the two inequalities.

For some flow f, let λ be the minimum feasible increase in capacity. If for some ℓ, (R1) and (R2) are closed for $\epsilon = 0$, then $\lambda = \lambda^*$. This is approximately true even if (R1) and (R2) are approximately closed.

Theorem 10.5. *If f, ℓ and λ satisfy (R1) and (R2) when $\epsilon < \frac{1}{7}$, then $\lambda \leq (1 + 5\epsilon)\lambda^*$.*

Proof.

$$\begin{aligned}
\lambda \sum_e \ell(e) Cap(e) &\leq \frac{1}{(1-\epsilon)} \sum_i C_i(\ell, \lambda) && \text{from (R1)} \\
&\leq \frac{1}{(1-\epsilon)^2} \sum_i C_i^*(\ell, \lambda) + \frac{\epsilon}{(1-\epsilon)^2} \lambda \sum_e \ell(e) Cap(e) && \text{from (R2)} \\
\lambda &\leq \frac{1}{(1-\epsilon)^2 - \epsilon} \frac{\sum_i C_i^*(\ell, \lambda)}{\sum_e \ell(e) Cap(e)} \\
&\leq \frac{1}{(1-\epsilon)^2 - \epsilon} \lambda^* && \text{since } \epsilon < \tfrac{1}{7} \\
&\leq (1 + 5\epsilon)\lambda^*
\end{aligned}$$

∎

10.3.2 Main Idea Behind the Algorithm

Define $\lambda(e) = \dfrac{f(e)}{Cap(e)}$ where $f(e)$ is the total flow through edge e. We seek a flow that satisfies demand and minimizes the scale factor by which we multiply capacities to make the flow feasible. That is, we want to minimize the scale factor over all flows that meet the demand.

$$\min_f \lambda_f = \min_f \max_{e \in E} \lambda(e)$$

One simple way to view this is by looking at the λ_f parameter as measuring *congestion* of an edge so that we are looking for a flow that minimizes congestion.

The min-max function is not smooth and does not give us a way to compare different flows well. A flow may require a large capacity over one single edge or may require uniform λ over all edges. Since we are more interested in the latter, we will define a potential funtion which will closely resemble the min-max value. In particular, define $\Phi(f) = \sum_e \exp[\alpha \lambda(e)]$.

A natural way to minimize the potential define above is to start from an initial point and "move" in the direction of steepest descent, i.e. opposite to the gradient of the potential. In other words, from a given multiflow f,

$$\hat{f} = f + \sigma(-\nabla_f \Phi), \qquad \left(\nabla_f \Phi(e) = \sum_e \frac{\partial \Phi(f)}{\partial f(e)} \right)$$

where the step size σ, is chosen suitably. A small σ would be more desirable because the gradient will also be changing over this path. However smaller σ would also mean a large number of steps and a high running time.

However, there is a major drawback to this method. Given a flow f_i to start with, the resulting point $\hat{f}_i$ will not be a flow in general. An immediate idea would be to take a projection of this gradient onto the feasible space of flows. But again, projection is a costly operation. Also, we may be very close to the boundary and the projection will only allow us to take a small step.

Hence we should move towards that point of the feasible space which decreases λ the most and in some form also captures the negative gradient direction. We therefore define the cost function

$$\ell(e) = \frac{1}{Cap(e)} \exp\left[\alpha \frac{f(e)}{Cap(e)}\right] = \frac{1}{\alpha} \frac{\partial \Phi(f)}{\partial f(e)} \tag{13}$$

We compute k min-cost flows which satify the capacites $\lambda_f Cap(e)$. We update our flow by a suitable constant σ (to be defined later). We will choose the parameter α to satisfy (R1) at all times and stop the algorithm as soon as (R2) is satisfied, which will guarantee ϵ-optimality. Notice that condition (R2) says that $\Delta f \cdot \nabla \Phi$ is small, i.e. loosely speaking, there is no way to reduce $\Phi(f)$ if f is to remain a flow. There are a number of issues to be addressed:

1. Correctness of our approach
2. How to get started, i.e. how to compute an initial multiflow that meets the demands, from which to start the descent
3. Are we sure that we reduce Φ in each iteration? If so, does it decrease by a non-trivial amount?
4. If $\Phi(f)$ decreases, is it true that λ_f also decreases?

The issues described above will be dealt with in detail in the next section.

10.3.3 Maintaining R1

For our algorithm to work, (R1) must always be able to be satisfied. We can do this by choosing the parameter α that appears in the length function to be large enough that (R1) is satisfied. The intuition is that when α is large the congested edges contribute so much to the volume that all the edges congested by at most $\frac{\lambda}{1+\epsilon/2}$ contribute only $\frac{\epsilon}{2}$ fraction. Therefore Φ is mostly the volume of the congested edges, and for these the flow and capacity are roughly the same!

Lemma 10.6. *If multiflow f satisfies demands d_i with capacity scaled by λ, and $\alpha \geq (2+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(2m\epsilon^{-1})$, then f and the length function ℓ defined by (13) satisfy condition (R1).*

Proof. Assume we choose $\alpha \geq (2+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(2m\epsilon^{-1})$.

Consider an edge e_1 such that, $f(e_1)/Cap(e_1) < \lambda/(1+\frac{\epsilon}{2})$. Denote the set of all such edges by E_1. Thus,

$$Cap(e_1)\ell(e_1) = \exp[\alpha f(e_1)/Cap(e_1)] < \exp[\alpha\lambda/(1+\frac{\epsilon}{2})].$$

Since λ is chosen as the minimum possible value such that the current flow f satisfies capacities λCap, there exists some edge e_2 with $f(e_2) = \lambda Cap(e_2)$. Thus, $Cap(e_2)\ell(e_2) = \exp[\alpha f(e_2)/Cap(e_2)] = \exp[\alpha\lambda]$.

Therefore,

$$\sum_e \ell(e)Cap(e) \geq Cap(e_2)\ell(e_2) = \exp[\alpha\lambda].$$

Combining this inequality with the previous inequality,

$$\begin{aligned}\frac{Cap(e_1)\ell(e_1)}{\sum_e \ell(e)Cap(e)} &\leq \frac{\exp[\alpha\lambda/(1+\frac{\epsilon}{2})]}{\exp[\alpha\lambda]} \\ &= \exp[-\alpha\lambda\epsilon/(2+\epsilon)] \\ &\leq \frac{\epsilon}{2m}\end{aligned}$$

Hence the total contribution to $\lambda\sum_e \ell(e)Cap(e)$ from all edges in E_1 is at most $\frac{\epsilon}{2}$.

Let E_2 be the set of all edges for which $f(e) \geq \lambda Cap(e)/(1+\frac{\epsilon}{2})$, so all the edges not in E_1. Clearly,

$$\begin{aligned}(1-\frac{\epsilon}{2})\lambda\sum_e \ell(e)Cap(e) &\leq \lambda\sum_{e\in E_2}\ell(e)Cap(e) \\ &\leq \lambda(1+\frac{\epsilon}{2})\sum_{e\in E_2}\ell(e)f(e) \leq \lambda(1+\frac{\epsilon}{2})\sum_e \ell(e)f(e)\end{aligned}$$

Since $(1-\frac{\epsilon}{2})/(1+\frac{\epsilon}{2}) \geq 1-\epsilon$, (R1) is satisfied and the lemma follows. ∎

10.4 Details of the Algorithm

We are now ready to present the algorithm rigorously, choose appropriate parameters, and prove bounds on its running time.

We assume that we have a routine MIN-COST-FLOW(s,t,d,ℓ,u) which computes the single commodity flow that minimizes $\sum_e \ell(e)f(e)$ while meeting the demand d from vertex s to vertex t and respecting the capacity constraints u. We denote T_{mcf} its running time. See the end of this section for a discussion on algorithms for min-cost-flow.

10.4.1 Getting Started

To start the algorithm, we have to compute an initial feasible flow. As we see below, we can construct one that is at most k times worse than the optimal flow. This bound will be used to know how many iterations have to be performed.

To find a starting flow f^0, we first find for each commodity i, the maximum flow g_i from s_i to t_i that is compatible with the capacity constraint Cap. Let $\lambda_i = \frac{d_i}{g_i}$. Then define, for each i,

$$f_i = \lambda_i g_i$$

which carries the required demands d_i and respects the capacity $\lambda_i Cap$. Observe that $\lambda_i \leq \lambda^*$, since if it were not the case the optimal flow divided by λ^* would define a flow for commodity i of value greater than g_i and g_i is maximal. The superposition f of all f_i's is a valid initial feasible flow, and, by construction,

$$\lambda^* \leq \sum_i \lambda_i \leq k\lambda^*$$

10.4.2 The Elementary Step

The following procedure takes a flow and returns a flow with significantly reduced potential using a number of minimum cost flow computations.

Reduce $\Phi(f, \alpha, \epsilon)$

$\lambda \leftarrow max_e \left(\frac{f(e)}{Cap(e)}\right)$.
$\sigma \leftarrow \frac{\epsilon}{4\alpha k \lambda}$.
$\ell(e) = \frac{1}{Cap(e)} \exp\left[\alpha \frac{f(e)}{Cap(e)}\right], \quad \forall e.$
$f_i^* \leftarrow$ MIN-COST-FLOW$(s_i, t_i, d_i, \ell, \lambda Cap), \quad i \in \{1, \ldots, k\}.$
$\hat{f}_i \leftarrow f_i + \sigma(f_i^* - f_i), \quad i \in \{1, \ldots, k\}.$
if $\ell \cdot (f - f^*) \leq \epsilon(\ell \cdot f + \lambda u \cdot \ell)$ **return** f
else return $\hat{f}$.

Intuitively, this algorithm takes advantage of the fact that edges that have been saturated have the largest values of ℓ. Edges not saturated have much smaller values of ℓ because $\ell = \frac{1}{cap} \cdot e^{f \cdot cap^{-1} \alpha} \approx \frac{1}{cap}$, which is small. Thus f_i^* tries to avoid saturated edges.

Our computation of $\hat{f}$ moves the solution towards unsaturated (cheaper) flows. Saturated edges will be reduced to $(1 - \sigma) f_i$ and f_i^* avoids edges with large ℓ, so the flow will not be increased by more than the σf_i we are reducing it by. Thus we expect flow will be reduced except where absolutely necessary.

The basic idea is that we are normalizing, pushing down flow on saturated edges and bringing up flow on the unsaturated ones. If the algorithm makes progress, it should obviously generate an optimal flow as the termination condition is (R2) being satisfied and (R1) is always satisfied. But how do we show that progress is made?

The following lemma shows that if (R2) is not satisfied, Φ reduces by a significant amount. The general approach is to consider the first order approximation and show it provides enough progress, using the second order approximation to formally prove that the progress is sufficient. Intuitively, the first order term "helps" progress while the second order term has the opposite effect, but the latter is relatively small so the overall progress is good.

Lemma 10.7. *Assume that* $\alpha \geq \frac{2+\epsilon}{\lambda \epsilon} \ln \frac{2m}{\epsilon}$. *If* $\epsilon \leq \frac{1}{7}$ *and the flow* f *is such that* λ *is* not ϵ*-optimal, then*

$$\frac{\Phi - \hat{\Phi}}{\Phi} \geq \frac{\epsilon^2}{4k} \tag{14}$$

where Φ *is the potential of the initial flow* f *and* $\hat{\Phi}$ *is the potential of the returned flow* $\hat{f}$.

Proof. Since we choose α satisfying (R1), and since λ is not ϵ-optimal, Theorem 10.5 implies that (R2) is not satisfied.

Note that $\Phi = \sum_e Cap(e)\ell(e)$. Let us examine each term of $\Phi - \hat{\Phi}$ separately (we write $\hat{\ell}$ for the cost function corresponding to $\hat{f}$):

$$Cap(e)\ell(e) - Cap(e)\hat{\ell}(e) \quad = \quad Cap(e)\ell(e)\left(1 - \exp\left[\alpha\sigma \frac{f^*(e) - f(e)}{Cap(e)}\right]\right)$$

This is an exponential function, and we would like to bound it. Before showing a correct bound, we will try using the standard first-order exponential bound $e^x \approx 1 + x$. Applying the bound, we get the following as an approximation for the exponential function if σ is small:

$$\begin{aligned} 1 - \exp\left(\alpha\sigma + \frac{f^*(e) - f(e)}{Cap(e)}\right) &\approx 1 - \left(1 + \alpha\sigma\frac{f^*(e) - f(uv)}{Cap(e)}\right) \\ &= -\alpha\sigma\frac{f^*(e) - f(uv)}{Cap(e)} \\ &= \alpha\sigma\frac{f(e) - f^*(uv)}{Cap(e)} \end{aligned}$$

This makes for a reasonably large value of $Cap(e)\ell(e) \cdot (1 - \exp[\cdots])$, and which we can simplify as follows:

$$Cap(e)\ell(e)\left(\alpha\sigma\frac{f(e) - f^*(uv)}{Cap(e)}\right) = \ell(e) \cdot \alpha\sigma(f(e) - f^*(uv))$$

If we examine the rightmost term closely, we find it reminds us of $\sum C_i - \sum C_i^*$, which appears in (R1) and (R2). Intuitively, we can sum over all edges, and thus:

$$\Delta\Phi \approx \alpha\sigma(C_i - C_i^*)$$

We know (R1) holds, so therefore (R2) must be false—if (R2) were true, we would be done, and we know our solution is not yet optimal. Thus we can apply the reverse of (R2):

$$\Delta\Phi \approx \alpha\sigma(C_i - C_i^*) \geq \alpha\sigma \cdot \epsilon\left(C_i + \lambda\sum_e Cap(e)\ell(e)\right)$$

We can observe that C_i is small, and $\sum Cap(\cdot)\ell(\cdot)$ is equal to the current Φ, so we can write this as:

$$\Delta\Phi \approx\geq \alpha\sigma\epsilon\Phi\lambda = \alpha\frac{\epsilon}{4\alpha k\lambda}\epsilon\Phi\lambda = \frac{\epsilon^2}{4k}\Phi$$

In other words, the exponent disappears, and the difference in Φ is related to Φ. Due to R2, we have our desired measurable decrease in Φ. In fact, since the decrease is known, we find that to reduce Φ to $\approx \frac{1}{\epsilon}\Phi$, we need only $\frac{k}{\epsilon^2}$ iterations of our algorithm. This means that to reduce Φ by a factor of two, we need only a constant number of iterations.

Now that we have shown an intuitive justification for approximating the exponential, we can show that the same bound applies using a real, accurate, proof.

Lemma 10.8. *For $|x| \leq \frac{\epsilon}{4} \leq \frac{1}{4}$, the following holds:*

$$e^x \leq 1 + x + \frac{\epsilon}{2}|x|$$

Proof. Bound the exponential by a second order Taylor series. ∎

We have choosen σ small enough for the above claim to hold. Note that the flow $f_i^*(e)$ of the i^{th} MIN-COST evaluation through edge e is at most $\lambda Cap(e)$. Thus the value $f^*(e)$ can be at most $k\lambda Cap(e)$.

Thus $\sigma\alpha(\frac{f^*(e)-f(e)}{Cap(e)}) \leq \frac{\epsilon}{4}$. Using the Taylor expansion above,

$$\begin{aligned} Cap(e)\ell(e) - Cap(e)\hat{\ell}(e) &\geq \ell(e)\alpha\sigma\left(f(e) - f^*(e) - \frac{\epsilon}{2}|f(e) - f^*(e)|\right) \\ &\geq \ell(e)\alpha\sigma\left(f(e) - f^*(e) - \frac{\epsilon}{2}(f(e) + f^*(e))\right) \text{ since } f^*, f \geq 0. \end{aligned}$$

We now sum over all edges. For simplicity, denote $C_i = C_i(\ell, \lambda)$ and $C_i^* = C_i^*(\ell, \lambda)$. Since $\sum_i C_i = \sum_e \ell(e)f(e)$,

$$\begin{aligned} \Phi - \hat{\Phi} &\geq \alpha\sigma(\textstyle\sum_i C_i - \sum_i C_i^* - \frac{\epsilon}{2}(\sum_i C_i + \sum_i C_i^*)) \\ &\geq \alpha\sigma(\textstyle\sum_i C_i - \sum_i C_i^* - \epsilon\sum_i C_i) && (\text{as } \textstyle\sum_i C_i^* \leq \sum_i C_i) \\ &\geq \alpha\sigma\lambda_f\epsilon\textstyle\sum_e Cap(e)\ell(e) && (\text{by } (\neg\text{R2})) \\ &= \alpha\sigma\lambda_f\epsilon\Phi \end{aligned}$$

The claimed bound follows from substituting σ by its value. ∎

10.4.3 From Reducing Φ to Reducing λ

From the above discussion, we require $O(k\epsilon^{-2})$ iterations to to decrease Φ by a constant factor. However decreasing Φ does not mean decreasing λ by the same factor.

Given an initial flow f with $\lambda = \lambda_0$, the following procedure guarantees that the flow f' it returns verifies $\lambda_{f'} \leq (1-\beta)\lambda_0$, or it returns an ϵ-optimal flow. The correctness of the algorithm is straightforward once we prove that it terminates. Notice that we choose α so that (R1) remains valid during all iterations.

Reduce$\lambda(f, \beta)$
$\alpha_0 \leftarrow \frac{2+\epsilon}{\lambda_0\epsilon} \ln \frac{2m}{\epsilon}$.
$\alpha \leftarrow \frac{\alpha_0}{(1-\beta)}$.
while $\lambda \geq (1-\beta)\lambda_0$ and there is progress
  $f \leftarrow$ Reduce$\Phi(f, \alpha, \epsilon)$.
return f.

Consider Φ for values of λ in the interval $(\frac{\lambda_0}{2}, \lambda_0)$.

$$\exp\left[\frac{\alpha\lambda_f}{2}\right] \leq \Phi(f) \leq m\exp\left[\alpha\lambda_f\right]$$

The first inequality follows from the fact there is at least one edge which is congested more than $\frac{\lambda_0}{2}$. The second one follows from the fact that at most m edges are in the graph and maximum congestion is λ_0.

Note that $\alpha = \alpha_0/(1-\beta)$ is retained constant during the interval $[(1-\beta)\lambda_0, \lambda]$ and is not varied every time according to the current value of λ. This is done in order to relate the decrease in Φ and λ. Using α depending on $\lambda_0/2$, for any $\lambda \geq \lambda_0$ does not hurt the previous analysis, since the required bound gets satisfied.

Thus if Φ decreases by a constant factor $\tilde{O}(\lambda_0\alpha)$ times, λ has to decrease by half. Thus in $\tilde{O}(k\epsilon^{-3})$ iterations, we are done. However, each iteration is k mincost flows, requiring a total time of $\tilde{O}(k^2mn\epsilon^{-3})$.

10.4.4 Reducing Running Time

From the above building block we can devise two different ways to reach the optimal λ^*. The first is to start with the initial solution, which is at most k times worse than the optimal, then halve $\lambda \log k$ times. We obtain the following time bound:

Lemma 10.9. *λ^* can be found in $\tilde{O}(k^2mn\epsilon^{-3})$ time by successive halving of λ.*

In the above running time, one of the ϵ^{-1} can be removed by using a trick of "scaling" on ϵ. This can be described as follows: Fix an ϵ_0, and compute an ϵ_0 approximate solution using the above strategy, which takes, $\tilde{O}(k^2mn\epsilon_0^{-3})$. Note that ϵ_0 is small but constant, say for example $1/7$. Now we have a solution which is ϵ_0 approximate to start with. Next, obtain a solution which is $\epsilon_0/2$ approximate from the ϵ_0 approximate solution. In general, at any stage start from an ϵ' solution to obtain an $\epsilon'/2$ solution. For $\lambda \in [(1+\epsilon'/2)\lambda^*, (1+\epsilon')\lambda^*]$,

$$\exp\left[\alpha(1+\frac{\epsilon'}{2})\lambda^*\right] \leq \Phi \leq \exp\left[\alpha(1+\epsilon')\lambda^*\right]$$

Hence, to obtain the $\epsilon'/2$ solution from the ϵ' solution requires a factor of $\exp \alpha\lambda^*\epsilon'$ reduction in Φ, which requires $O(\alpha\lambda^*\epsilon')$, or $\tilde{O}(k\epsilon'^2)$, steps. The above procedure is repeated $\log(\epsilon'/\epsilon)$ times. Thus the net running time is bounded above by

$$\begin{aligned}
& \tilde{O}\left(k^2mn\epsilon_0^{-3} + kmn\sum_{j=0}^{\log(\epsilon'/\epsilon)}\frac{k}{(\epsilon 2^j)^2}\right) \\
\leq\ & \tilde{O}\left(k^2mn\epsilon_0^{-3} + kmn\frac{k}{\epsilon^2}\sum_{j=0}^{\infty}2^{-2j}\right) \\
=\ & \tilde{O}\left(k^2mn\epsilon_0^{-3} + kmn\frac{k}{\epsilon^2}\right) \\
=\ & \tilde{O}\left(\frac{k^2mn}{\epsilon^2}\right)
\end{aligned}$$

This leads to the following theorem:

Theorem 10.10. *An ϵ-optimal flow can be computed in $\tilde{O}(k^2mn\epsilon^{-2})$ time.*

10.4.5 Using Randomization

By resorting to randomization, we can knock off a factor of k, and get a better running time, though in expectation instead of in the worst case. The idea is to use a larger step σ while performing only one MCF. The expected decrease in Φ is the same as before.

Theorem 10.11. *An ϵ-optimal flow can be computed in $\tilde{O}(kmn\epsilon^{-2})$ expected time.*

Proof. We had updated the current flow at each iteration by computing k minimum-cost flows, one per commodity. This actually forced us to take steps of σ which was small. runtimes to the previous

technique. We will now choose one commodity uniformly at random and compute the min-cost flow for that community. The change per edge if we move commodity i will be

$$Cap(e)\ell(e) - Cap(e)\hat{\ell}(e) = Cap(e)\ell(e)\left(1 - \exp\left[\sigma\alpha\frac{f_i^*(e) - f_i(e)}{Cap(e)}\right]\right)$$

Since f' is a single commodity flow, we chose $\sigma = \frac{\epsilon}{4\alpha\lambda_f}$. Using the same claim as in Lemma 10.7, we get:

$$Cap(e)\ell(e) - Cap(e)\hat{\ell}(e) \geq \ell(e)\alpha\sigma\left(f_i(e) - f_i^*(e) - \frac{\epsilon}{2}(f_i(e) + f_i^*(e))\right)$$

Note also that $C_i = \sum_e \ell(e)f_i(e)$. We can now calculate the expected decrease in potential as:

$$\begin{array}{rcll} E[\Phi - \hat{\Phi}] & \geq & E[\alpha\sigma(C_i - C_i^* - \frac{\epsilon}{2}(C_i + C_i^*))] & \\ & = & \frac{\alpha\sigma}{k}(\sum_i C_i - \sum_i C_i^* - \frac{\epsilon}{2}(\sum_i C_i + \sum_i C_i^*))] & \\ & \geq & \frac{\alpha\sigma}{k}(\sum_i C_i - \sum_i C_i^* - \epsilon\sum_i C_i) & (\text{as } \sum_i C_i^* \leq \sum_i C_i) \\ & \geq & \frac{\alpha\sigma}{k}\lambda_f\epsilon\sum_e Cap(e)\ell(e) & (\text{by } (\neg\text{R2})) \\ & = & \frac{\alpha\sigma\lambda_f\epsilon\Phi}{k} & \\ & = & \frac{\epsilon^2\Phi}{4k} & \end{array}$$

Notice that this proves the same lemma as we had proved before. Therefore after $\tilde{O}(k\epsilon^{-3})$ we are done. Notice that the cost of an iteration now is a single min-cost flow, which is $\tilde{O}(mn)$. This gives us an algorithm of running time $\tilde{O}(kmn\epsilon^{-3})$.

We can now use the same technique as in the previous section, i.e. to consider a two phase approach and scaling. We would now require $\tilde{O}(k\epsilon^{-2})$ iterations to reduce λ. Which completes the proof. ∎

However the algorithm can be derandomized by choosing commodities in a round robin fashion.

10.4.6 Precision Concerns in Minimum-Cost Flow

One point remains in determining the run time of the algorithm: Which minimum-cost flow routine should we use? The length function, which is exponential, would seem to make standard cost scaling algorithms inappropriate. But it turns out we can use an approximation to the length function and still obtain essentially the same analysis. The main idea is to observe that instead of computing the minimum cost flow at each iteration, it is sufficient to compute a flow with cost is within $\frac{\epsilon}{2k}\lambda\Phi + \frac{\epsilon}{2}C_i$ additive error of the optimal.

11 The Fractional Packing Problem

In this lecture we will generalize the techniques previously developed in the context of approximation algorithms for multicommodity flow and apply them to a larger class of linear programs. In the last section we gave a complete derivation of these techniques; in this and subsequent sections, we will use the generalization without proofs, as a "black box". Additional details can be found in [17].

11.1 Introduction

We consider the following class of problems:

Definition 11.1 (Fractional packing problem)**.** *Given a polytope P and a matrix $A \geqslant 0$, find $x \in P, x \geqslant 0$, such that $Ax \leqslant b$.*

A variety of problems can be expressed as fractional packing problems. An example of a simple packing problem is the knapsack problem. Another example is a decision version of maximum flow. Given a graph G with m edges, source s, and sink t, we can formulate the problem of whether the maximum flow has value at least F as follows: Let x_i be the flow on path p_i. The capacity constrains are

$$\forall e, \sum_{e \in p_i} x_i \leqslant cap(e)$$

We can express this in the form $Ax \leqslant b$, where b is the matrix of the capacity on each edge and A is the matrix whose rows are the edges e and whose columns are the paths p. An element A_{ij} is defined as 1 if $e_i \in p_j$, else 0. The polytope P expresses the demand

$$\sum_i x_i = F$$

Note that here we have a particular kind of polytope: a simplex.

An alternative is to express the constraints in terms of $Ax \leqslant \lambda b$, with an objective of minimizing λ. This is the form we will use in the following sections.

11.2 The Algorithm

We now describe an algorithm to find approximate solutions for the fractional packing problem. Although fractional packing can be solved exactly using linear programming, we expect this method to be much faster than solving using linear programming techniques.

We assume that we have a fast subroutine, i.e. an *oracle*, to solve the following optimization problem:

Definition 11.2 (Problem solved by the oracle)**.** *Given a polytope P, a matrix A and an m-dimensional vector $y \geqslant 0$, find $\tilde{x} \in P$ such that:*

$$c\tilde{x} = \min(cx : x \in P), \text{ where } c = y^t A$$

For a given error parameter $\epsilon > 0$, we want to find $x \in P$ such that $Ax \leq (1+\epsilon)b$. In other words, we want $\lambda \leqslant (1+\epsilon)$. This will mean that x is an *ϵ-approximate solution* to the PACKING problem.

As in the multicommodity flow algorithm [10], the core of this algorithm is a procedure which produces a new feasible solution (x, λ) using a dual solution y defined as a function of x, where

$$y_i = \frac{1}{b_i} e^{\alpha a_i x / b_i}$$

for appropriate choice of α. This y_i measures how badly x satisfies inequality i. At each iteration of the packing algorithm, the oracle tries to reduce the values of x that participate in "bad" inequalities by setting

$$x \leftarrow (1 - \sigma)x + \sigma\tilde{x}$$

An entire iteration of the algorithm runs as follows:

1. $y_i = \frac{1}{b_i} e^{\alpha a_i x / b_i}$
2. $c = y^t A$
3. $\tilde{x} \leftarrow$ output of oracle
4. $b = (1 - \sigma)x + \sigma\tilde{x}$

11.3 Running Time

The running time of the relaxed decision procedure depends on the width of P relative to $Ax \leqslant b$. Intuitively, this is the largest possible "overflow" of a constraint.

Definition 11.3 (Width of P)**.** *The width of P relative to $Ax \leqslant b$ is defined by*

$$\rho = \max_i \max_{x \in P} \frac{a_i x}{b_i}$$

In general, ρ might be large, even superpolynomial in the size of the problem. See [17] for techniques to reduce ρ.

Theorem 11.4 ([17])**.** *The algorithm uses*

$$O(\epsilon^{-2} \rho \log(m\epsilon^{-1}))$$

calls to the subroutine (11.2) for P and A, plus the time to compute Ax for the current iterate x between consecutive calls [3]*.*

Note that the running time does not depend explicitly on n, the dimension of P. This makes it possible to apply the algorithm to problems defined with an exponential number of variables, assuming we have a polynomial-time subroutine to compute a point $x \in P$ of cost $y^t Ax$ given any positive y, and that we can compute Ax for the current iterate x in polynomial time. A slightly different generalization of the multicommodity flow algorithm was discovered by Grigoriadis and Khachiyan [6].

[3] See [17] for proof in details.

11.4 Application to multicommodity flow

In the context of multicommodity flow, $Ax \leq \lambda b$ represents the capacity constraints; the convex set P represents flows that satisfy all the demands and that, for each commodity, satisfy the capacity constraints. Observe that the needed optimization subroutine corresponds to optimizing a given cost function over the convex set P, which is equivalent to solving k min-cost flows. Since no point in P can overflow the capacity by more than a factor of k, the value of ρ for the multicommodity flow problem stated in this way is bounded by k.

It is important to observe that different (equivalent) formulations of the same problem might lead to different values of ρ. For example, an alternative formulation of multicommodity flow as packing is to define P as the set of flows that satisfy the demands, i.e. remove the capacity constraints for individual commodities from P. The optimization procedure (11.2) needed here is equivalent to k shortest paths instead of k min-cost flows that was needed before. Unfortunately, the value of ρ in this formulation can be as large as maximum demand over minimum capacity, which can be exponential in the size of the problem. Roughly speaking, there is a tradeoff between the complexity of P (and hence the complexity of the optimization procedure) and ρ, which, as we will see, determines the number of iterations.

12 Scheduling unrelated parallel machines

Suppose that there are n jobs and m machines, and each job must be scheduled on exactly one of the machines. For simplicity of notation, assume that $n \geq m$. Job j takes p_{ij} time units when processed by machine i. The *length* of a schedule is the maximum processing time of any machine to run all the jobs assigned to that machine; the objective is to minimize the schedule length. This problem, often denoted $R||C_{\max}$, is NP-complete, and in fact, Lenstra, Shmoys and Tardos [7] have shown that there does not exist an ϵ-approximation algorithm with $\epsilon < 1/2$ unless $P = NP$.

In the weighted variant of unrelated machine scheduling, a decision of assigning job j to machine i carries cost c_{ij}. The goal is to find a schedule with minimum length and with total cost below some given budget B.

"Unrelated" means that the time it takes machine i to execute job j is an arbitrary nonnegative number. This is in contrast to a model where, if the machines are identical, the p_{ij}'s are independent of j. In this "related" model, it may be the case that job j_1 takes half as long to execute as j_2, no matter which machine is used.

For real world machines, the unrelated model is a reasonable model. Consider a set of computers, and suppose they are all identical. Then data would be processed at the same speed across all machines. However, some data may prefer machines closer to the one on which it resides, thus giving an unrelated processing time. In the case that the machines are not identical, different jobs may require different capabilities with which some machines are better equipped than others, such as disk, memory, cache, and so on.

We will be trying to optimize the time when the last job is finished. There are other reasonable things to optimize, such as how long a job waited before it was executed. But we will not consider those problems here. Our objective is called the *makespan* objective.

Our approach to solving the scheduling problem for the unrelated model will be to use the packing problem as a starting point and find a way to massage it into an appropriate integer solution. We will first construct an LP for the problem, see that the rounding it gives us is unacceptable, and then construct a better LP to work with.

12.1 Solving the relaxation

Unrelated machines scheduling can be stated as an IP [4]. The basic idea is to state an appropriate relaxation of this IP, solve it, and round the solution without significantly increasing the length and cost. The unweighted case was considered by Lenstra, Shmoys, and Tardos [7]. They have proposed an approach where a vertex of an appropriate relaxation of length T can be rounded to a solution with length bounded by $2T$.

Shmoys and Tardos [19] came up with a simpler approach that can solve the weighted case and does not require us to obtain a vertex of the relaxed LP. This section describes their algorithm.

Suppose that there exists a schedule of length T with cost less than B. Then the following linear program has a feasible solution:

[4] A greedy heuristic for solving IP may not do well on this problem because the processing time of each job may not be identical. Hence we can not reserve spare machines in advance as not all machines can do all jobs and the p_{ij}'s are unrelated.

$$\sum_{j=1}^{N} p_{ij}x_{ij} \quad \leq \quad T, \;\; i = 1, \ldots, M; \tag{15}$$

$$\sum_{i=1}^{M} x_{ij} \quad = \quad 1, \; j = 1, \ldots, N; \tag{16}$$

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij}x_{ij} \quad \leq B \quad ; \tag{17}$$

$$x_{ij} \quad = \quad 0 \;\text{ if } p_{ij} > T, \; i = 1, \ldots, M, \; j = 1, \ldots, N, \tag{18}$$

$$x_{ij} \quad = \quad 0 \;\text{ if } c_{ij} > B, \; i = 1, \ldots, M, \; j = 1, \ldots, N, \tag{19}$$

$$x_{ij} \quad \geq \quad 0 \;\text{ if } p_{ij} \leq T, \; i = 1, \ldots, M, \; j = 1, \ldots, N. \tag{20}$$

Recall that our goal is to solve the IP. Since we can't enforce integer solutions we need some way of preventing "bad" fractional solutions. One type of an undesirable fractional solution is when we assign a fraction of a job to a machine on which that job *alone* would violate our time bound T or budget B. As a concrete example, suppose the fractional solution splits a job j_0 into many tiny pieces, allocating each piece to a machine where p_{ij_0} is huge, say $1000T$. The fractional solution may have a reasonable processing time. But rounding would assign the job to one of the undesirable machines, which already exceeds the allowed time.

To avoid this situation, we add constraints (18) and (19). These constraints can be seen as cutting planes which help us avoid such undesirable fractional solutions. To have a visual understanding of these cutting planes, suppose we have a convex set of feasible points, containing both integer points and lots of fractional points. The cutting planes cut away portions of the convex set that we know are useless, i.e. sections that only contain fractional points. Although this will change the fractional solution space, it will not change the integer solution space, which means that the LP is still valid for our final goal. The rounding procedure can now take advantage of the fact that the fractional solution already satisfies the additional constraints.

At this point we could solve the problem with an LP solver, where the number of variables is the number of jobs multiplied by the number of machines. We could use a black box to determine the value of T by binary search. As a sanity check, we can bound the range of T by mT_{OPT} where m is the number of machines. This bound arises from the fact that we could simply greedily assign each job to the best machine for it.

This approach works, but runs slower than using a fractional packing algorithm. To apply the approximate packing algorithm, we let P be defined by the constraints (16–20). The rest of the inequalities correspond to the $Ax \leq b$ constraints. It is easy to see that $\rho \leq N$: for any $x \in P$, $x_{ij} > 0$ implies that $p_{ij} \leq T$, and so $\sum_{j=1}^{N} p_{ij}x_{ij} \leq NT$ for each machine $i = 1, \ldots, M$.

Recall that the approximate packing algorithm relies on assigning a dual variable to each inequality in the $Ax \leq b$ part of the definition of the packing problem. In our case, the vector of dual variables is $y = (y_1, y_2, \ldots, y_M, z)$, where y_i corresponds to machine i and z corresponds to the budget constraint. The coefficient of x_{ij} in the aggregated objective function $y^t Ax$ is $y_i p_{ij} + z c_{ij}$. Since $P = P^1 \times \cdots \times P^N$, where each P^j is a simplex, we can minimize this objective function by separately optimizing over each

P^j. For a given P^j, this is done by computing the minimum modified processing time $y_i p_{ij} + c_{ij} z$, where the minimization is restricted to those machines for which $p_{ij} \leq T$ and $c_{ij} \leq B$.

Each iteration takes $O(MN)$ time and $\rho \leq N$. Hence, for any fixed $\epsilon > 0$ we can find a solution $\bar{x}$ of length at most $(1+\epsilon)T$ and cost at most $(1+\epsilon)B$ in $O(MN^2 \log M)$ time, if one of length T and cost B exists.

Recall that $P = P^1 \times \cdots \times P^N$, and we can also take advantage of this structure using randomization. Observe that $\rho^j \leq 1$, $j = 1, \ldots, N$, and we can optimize over P^j in $O(M)$ time; we can also compute the updated values Ax in $O(M)$ time. Applying the randomized variant of the approximate packing algorithm, we get a randomized algorithm that takes $O(N \log N)$ iterations, each of which takes $O(M)$ time. Furthermore, the solution $\bar{x}$ is expected to have $O(N \log N)$ positive components, since at most one is added at each iteration. To convert this relaxed decision procedure into an approximation algorithm, we use a bisection search to find the best length T. Since the schedule in which each job is assigned to the machine on which it runs fastest is within a factor of M of the optimum, $O(\log M)$ iterations of this search suffice.

12.2 Rounding Fractional Solutions

After computing a solution to the relaxed scheduling, we will use the algorithm of Shmoys and Tardos [19] to round this fractional solution to an integer solution, i.e. a solution in which each job is completely assigned to a single machine. Our main tool will be a special bipartite graph that corresponds to the fractional solution. In this graph, each job is represented by one node, and each machine is represented by several nodes. The machines are assigned one node for each full "job" assigned to it in the fractional solution. Specifically, for machine i, let $k_i = \lceil \sum_j x_{ij} \rceil$, and create k_i nodes for machine i. Note that k_i shows the number of jobs assigned to the machine i.

The edges are added to this graph in the following way. Take a specific machine i and consider all jobs j such that $x_{ij} > 0$, i.e. all jobs that use this machine in the fractional solution. Order these jobs according to decreasing p_{ij} values, i.e. we will handle the largest jobs first. Machine node order is important. We are going to "fill" each machine node, and so we have to know where to add arcs and when a node "fills up."

Deal with the first job j on this list by adding an arc from job node j to the first node associated with machine i. We view this as *assigning* job j to this node. For the remaining jobs, we need to know the sum of the x_{ij} for jobs already assigned to nodes representing machine i. Call this S_i. This sum will have an integer part, ($[S_i]$) and a fractional part, $frac(S_i) = S_i - [S_i]$. If adding x_{ij} for the current job leaves the integer part unchanged, then we connect the arc to the current machine i node. If, however, this addition "rolls" the integer part over, then we will fill the curent node with $(1 - frac(S_i)) - x_{ij}$ by adding an arc with this weight from j to the current node. Then we will put the remaining workload of x_{ij} by adding an arc from j to the *next* node of machine i. Note that we will have enough nodes of machine i to assign all the jobs, because of our definition of k_i. [5]

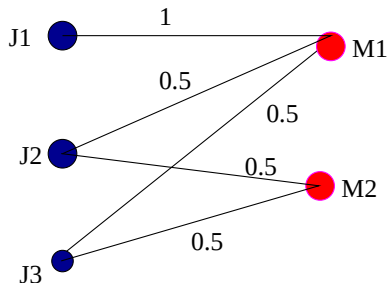


Figure 11: Example: Fractional matching solution of jobs (left nodes) to machines (right nodes).

The purpose of splitting each machine into one subnode per full job it was assigned, instead of having simply a bipartite graph with one node on the left per job and one node on the right per machine, is to avoid the problem of a rounding that may assign many jobs to large machines. By splitting one machine up, we are essentially separating small and large p_{ij}'s and rounding them separately.

We now have a bipartite graph with fractional weights on the edges where the sum of the weights on the arcs incident to any job and any separate machine node is equal to 1. This is a *fractional matching* that can be converted to an integral matching of equal or smaller cost that matches all the job nodes. In order to see why this is true, consider the following (inefficient) algorithm.

Pick any fractional edge $j_1 i_1$. The reason that we cannot increase its weight to 1 or decrease it to 0 is that this will cause the weight incident to i_1 to differ from 1. Note that there must be at least one

[5] In fact we *fill* the last machine node completely by creating a *slack* job if necessary. The sum of all fractions assigned to slack jobs can easily be seen to be integer by considering the integer number of jobs in the original problem.

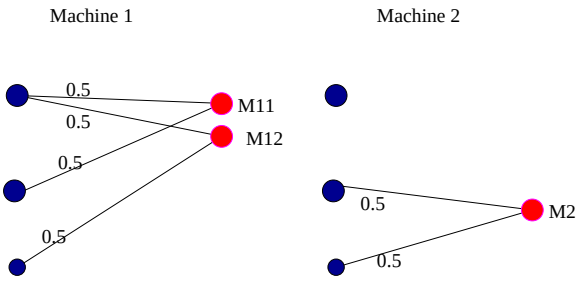


Figure 12: Dividing Machines into separate nodes: Machine 1 has total weight of 2 jobs, hence is represented by 2 nodes. Here we assume that $p_{21} \leq p_{11} \leq p_{31}$.

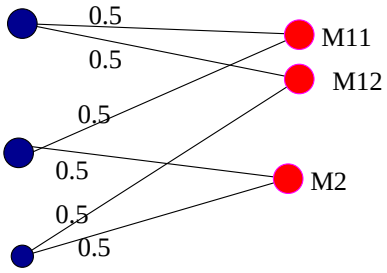


Figure 13: Separate machine graphs are combined.

additional fractional edge $j_2 i_1$ incident to i_1. Trying to increase or reduce the weight on $j_2 i_1$ means that we need to reduce or increase the weight on some other edge $j_2 i_2$, etc. Continuing in this fashion, we have to reach j_1 again at some point, since every new node implies another fractional edge. This creates an *augmenting cycle*.

The augmenting cycle has an edge of maximum flow and another of minimum flow. Each of these is a fraction between 0 and 1. If the cost of the augmenting cycle is positive in one direction, it is negative in the other. Thus we can increase or decrease the weights on the cycle in such a way as to not increase the cost. We increase or decrease the cycle flow in the cost reduction direction until maximum flow becomes 1 or the minimum flow becomes 0. Since the above procedure changes only fractional weights, and changes at least one edge from fractional to integer each iteration, it will terminate after number of augmentations equal to the number of edges.

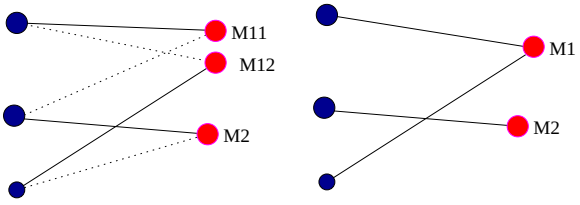


Figure 14: Graphs is augmented to give integer flows.

Another way to convert this fractional matching to an integral matching with the lower or same cost is to use the fact that vertices of the matching polytope are integral, so every fractional matching can be written as a convex combination of polynomial number of integral matchings. Thus one of them has less or equal cost to this fractional solution. In other words, just find an integral solution to the min-cost bipartite matching problem and use it. Existence of such integral solution is guaranteed by the

integrality property of the vertices of the polytope.

Properties of the solution Now we show that the properties of the constructed bipartite graph imply that any integral matching corresponds to a schedule of length at most $2T$ (while keeping the cost at most B as shown above). Look at the time required by machine i. There are k_i nodes corresponding to machine i in our graph and for each of these, there will be at most one job scheduled on machine i in the integer solution. In the fractional solution there are several incident edges, one of which will become the single edge in the integer solution.

Consider the total time required on this machine in the fractional solution, $T_{\texttt{fract}}$. Obviously $T_{\texttt{fract}} \leq T$, and if T_l is the time taken by jobs on node l for $l \leq k_i$ then $T_l = \sum x_{ij}^l p_{ij}$. (where x_{ij}^l is weight of edge from job j to the lth node for machine i).

Now $T \geq T_{\texttt{fract}} = T_1 + T_2 + \cdots + T_{k_i}$. Consider the values of p_{ij} for the jobs assigned to the first node associated with machine i in our construction of the bipartite graph and define $p_1^{\texttt{max}}$ and $p_1^{\texttt{min}}$ as the greatest and least of these. (Similarly for $p_l^{\texttt{max}}$ and $p_l^{\texttt{min}}$.)

Let T_{round} define the workload on machine i in the rounded solution. We see that $T_{\texttt{round}} \leq p_1^{\texttt{max}} + p_2^{\texttt{max}} + \cdots + p_{k_i}^{\texttt{max}}$. By the way we assigned the x_{ij}, we know that $T_l \geq p_l^{\texttt{min}} \geq p_{l+1}^{\texttt{max}}$ for $l < k_i$. Thus, we have

$$T_{\texttt{round}} \leq p_1^{\texttt{max}} + T_1 + T_2 + \cdots + T_{k_i - 1}$$

Since $p_1^{\texttt{max}} \leq T$ (by the cutting plane constraints), and $T_1 + T_2 + \cdots + T_{k_i - 1} \leq T_{\texttt{fract}} \leq T$, we get the desired result $T_{\texttt{round}} \leq 2T$.

13 Approximation Algorithms for Multicut Problems

13.1 Introduction

There are two natural variants of the multicommodity flow problem:

- MAX SUM FLOW: Maximize the total demand satisfied in the system.
- CONCURRENT FLOW: Minimize λ such that all demands are satisfied and capacity constraints are satisfied as well as long as each capacity is multiplied by λ.

As in the case of single-commodity flow, we can define corresponding cut problems:

For an undirected graph $G = (V, E)$, let $F \subseteq E$ be a set of edges[6].

Def. 1. *Let Cap(F) $= \sum_{e \in F} cap(e)$. For a set $U \subseteq V$, we define Cap($U, \overline{U}$) as Cap($F_{U,\overline{U}}$), where $F_{U,\overline{U}}$ is the set of edges that have one endpoint in U and the other in $\overline{U}$.*

Let the demand pairs be $\mathcal{D} = (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ where d_i is the demand between s_i, t_i.

Def. 2. *We say that F* **separates** *pair (s_i, t_i) if all paths from s_i to t_i contain at least one edge of F.*

Def. 3. *F Define Demand(F) as*

$$Demand(F) = \sum_{(s_i,t_i) \in \mathcal{D}, F \text{ separates } (s_i,t_i)} d_i$$

Now, we define the following cut problems:

- MINIMUM MULTICUT: Find a multicut F s.t Cap(F) is minimized and F separates **all** demands in $\mathcal{D}$
- SPARSEST CUT: (also called "Ratio Cut") Find a cut F that minimizes:

$$\min_F \frac{\text{Cap}(F)}{\text{Demand}(F)}$$

  Intuitively, the sparsest cut, is that set whose edges have the lowest capacity per unit of demand disconnected.

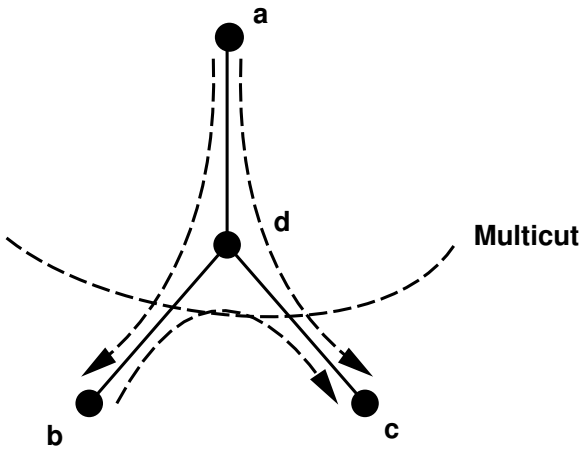
It should be noted that, in general, finding either the minimum multicut or the sparsest cut is NP-Hard.

Observe that weak duality holds for max sum flows and minimum multicuts i.e. the value of the max sum flow for an instance of MCF is at most the cost of any multicut. This is simply because all flow paths between the s-t pairs must traverse through a cut separating (all) these pairs, and therefore the maximum values of this flow is bounded by the capacity of the minimum such cut.

[6] We could also define the cut in terms of sets of vertices, but this formulation is more convenient.

13.2 Lower bound for min-cut max-flow ratio for MCF

Unlike the case for single commodity flow and min-cuts where we could prove exact min-cut/max-flow theorem – the optimal min cut has value equal to the value of the optimal max flow – for multicuts, such exact relationship does not hold.



Assume all edges in the figure have capacity 1 and we have to route commodities between the pairs marked by the arrows. As we can see, the minimum multicut in the graph has value 2, whereas the max sum flow has value 1.5 (assign each flow path – shown by dotted lines – to have value .5).

Analogous to max sum flows and minimum multicuts, in the case of concurrent flows and sparsest multicut weak duality always holds, but strong duality is not valid.

We will now use expander graphs to give a lower bound on the size of a minimum multicut relative to the size of the max sum flow. Before we do this we will describe expander graphs more in detail and prove some of their features required to give the above bound.

Expander graphs Consider a graph G with vertexes V and edges E.

Def. 4. *Let $U \subset V$. Then the **boundary** of U is defined to be all nodes in V connected to U, but not in U itself. i.e.*

$$boundary(U) = \{v \in (V - U) : (u, v) \in E \wedge u \in U\}$$

Def. 5. *If G is undirected with constant degree δ, then G is said to be an (δ, α) **expander** or have an α **expansion** iff*

$$\forall W \subset V \text{ such that } |W| \leq \tfrac{|V|}{2},\ boundary(W) \geq \alpha.|V|$$

Intuitively an expander graph is one where any small set of vertexes will have a large boundary. In general, any randomly created regular constant-degree graph will have expander properties.

Lemma 13.1. *The diameter of any expander graph is $O(\log |V|)$*

Proof. Consider any two arbitrary vertexes u and v in G an expander graph.

Let B_1 be the set of vertexes containing only u and the boundary of u. Let B_2 be the set of vertexes containing only B_1 and the boundary of B_1. Similarly let B_3 be the set of vertexes containing only B_2 and the boundary of B_2 ...

We have $B_1 \subseteq B_2 \subseteq B_3 \subseteq B_4 \ldots$

Here $B_1 = boundary(u) + u \Rightarrow |B_1| = |boundary(u)| + 1 \geq (1 + \alpha)$

Similarly $|B_2| = |boundary(B_1)| + |B_1| \geq (1 + \alpha)^2$

and $|B_3| = |boundary(B_2)| + |B_2| \geq (1 + \alpha)^3 \ldots$

This process will continue until we have a B_k where $|B_k| > \frac{|V|}{2}$.

The number of iterations that it will take until this B_k will be reached is bounded by $\log_{1+\alpha} \frac{|V|}{2}$. As in each iteration, the new nodes added are one more edge away from u than the nodes added in the previous iteration, the distance of the newly added nodes in B_k from u will be $\log_{1+\alpha} \frac{|V|}{2}$. All other nodes in B_k will be closer to u than this, so the distance of **all** nodes in B_k from u is bound by $\log_{1+\alpha} \frac{|V|}{2}$.

Now in a similar manner construct sets B'_1, B'_2, B'_3 ... around v. Let B'_l be the first set where $|B'_i| > \frac{|V|}{2}$. Analogous to the above case, the distance of all nodes in B'_l from v must be bounded by $\log_{1+\alpha} \frac{|V|}{2}$.

Now as $|B_k| > \frac{|V|}{2}$ and $|B'_l| > \frac{|V|}{2}$, B_k and B'_l must intersect. Let x be any vertex in this intersection.

$dist(u \to x) \leq \log_{1+\alpha}(\frac{|V|}{2}) \wedge dist(x \to v) \leq \log_{1+\alpha}(\frac{|V|}{2})$

$\Rightarrow dist(u \to v) \leq 2. \log_{1+\alpha} \frac{|V|}{2}$

Therefore the maximum diameter of G is $O(\log |V|)$ as required. ∎

The above lemma showed that in an expander graph, all vertices are "close to each other", i.e. no vertices are more than $O(\log |V|)$ edges apart. Since the graph has constant degree, we can also state that many vertex pairs are at distance at least $\Omega(\log |V|)$ from each other.

Lemma 13.2. *In an expander graph more than* $\frac{|V|}{2}$ *nodes are at a distance* $\Omega(\log |V|)$ *from any given node.*

Proof. Let G be an α expander with all nodes having constant degree δ.

We know that $\alpha \leq \delta$ – as as any set of vertexes $U \subset V$ cannot possibly grow faster than the degree of each node.

Now pick an arbitrary vertex v in the graph and consider all nodes within a distance $\frac{\log_\delta |V|}{2}$ from v.

Using the same construction of co-centric balls around v (as in the previous lemma), it is easy to see that:

$|\text{nodes within a distance of } \frac{\log_{1+\delta} |V|}{2} \text{ from } v| \leq (1 + \delta)^{\frac{\log_{1+\delta} |V|}{2}} < \frac{|V|}{2}$

This implies that more than $\frac{|V|}{2}$ nodes are at a distance $\geq \frac{\log_{1+\delta} |V|}{2} = \Omega(\log |V|)$ as required. ∎

Using expander graphs to derive a lower bound We will now use expander graphs to derive a lower bound on the size of the minimum multicut relative to the size of max sum flow. An analogous proof can be used to give an $\Omega(\log |V|)$ bound to the size of the sparsest cut relative to the size of the max concurrent flow.

Lemma 13.3. *There exists an instance of MCF where the minimum multicut is $\Omega(\log |V|)$ times the max sum flow.*

Proof. Let $G = (V, E)$ be a n-vertex bounded degree (each vertex has degree at most δ, for a constant δ) α expander. The lemma above implies that every vertex has more than $\frac{n}{2}$ vertices that are at distance $\Omega(\log n)$ from it. Therefore, there are $\Omega(n^2)$ pairs of vertices that are at a distance of at least $\Omega(\log n)$. Let these be the pairs for the MCF instance. Let each edge have capacity 1. The total capacity of the graph is $O(n)$, and each flow path is of length $\Omega(\log n)$, hence the max flow is $O(\frac{n}{\log n})$.

Consider the smallest capacity multicut. It separates the graph into connected components. Note that each component is strictly smaller than $n/2$ since otherwise it will include at least one source-sink pair. Thus, each component U_i has at least $\alpha|U_i|$ edges crossing its boundary. Thus, the capacity of the multicut is at least $\alpha n/2 = \Omega(n)$, where the division by 2 is due to each edge counted twice. This gives us cut-to-flow ratio of at least $\Omega(\log n)$, as desired. ∎

13.3 Minimum multicut in undirected graphs

Suppose we are given an instance of a multicommodity flow problem, with a capacitated graph $G = (V, E)$ and k demands $\mathcal{D} = \{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$, where demand d_i is associated with pair (s_i, t_i). Let $P_i = \{P_i^1, P_i^2, \ldots, P_i^l\}$ be the set of paths used to route the flow of commodity i from source to sink. Let $f(P_i^j)$ be the flow of commodity i on this path P_i^j.

The max-sum MCF flow problem is to maximize the total flow on all paths. Stated as an LP:

$$\begin{aligned}
\text{Maximize} \quad & \sum_{i=1}^{k} \sum_{j=1}^{|P_i|} f(P_i^j) \\
\text{subject to} \quad & \forall e \in E \quad \sum_{i=1}^{k} \sum_{e \in P_i^j} f(P_i^j) \leq \text{cap}(e) \\
\text{and} \quad & \forall P_i^j \quad f(P_i^j) \geq 0
\end{aligned}$$

The dual of this multicommodity flow problem is the minimum multicut problem, which can be written as follows.

$$\begin{aligned}
\text{Minimize} \quad & W = \sum_{e \in E} l(e)\text{cap}(e) \\
\text{subject to} \quad & \forall e \quad l(e) \geq 0 \\
\text{and} \quad & \text{dist}_l(s_i, t_i) \geq 1
\end{aligned}$$

where $l(e)$ is the variable corresponding to the capacity constraints and dist_l is the distance measure on G defined using edge cost $c(e) = l(e)$.

We can think of $l(e)$ as the length of a pipe e and $\text{cap}(e)$ as the cross-sectional area of e. Under this interpretation, W is the total volume of all the pipes. Observe that any multicut corresponds to a feasible dual – just assign $l(e) = 1$ for every edge e in the multicut. The opposite is not true, i.e. fractional dual does not necessarily correspond to a multicut.

To simplify notation, we drop the subscript l from dist_l.

An $O(\log k)$ approximation We present an algorithm to approximate the minimum multicut to within a $\log k$ factor of optimal. Since the relaxed version of the minimum multicut problem is just the dual of the max-sum MCF problem, we use the solution of the dual of max-sum MCF to guide our approximation to the NP-hard multicut problem. Specifically, we make use of the distance metric *dist* on G derived from this solution to give us information about the structure of the graph. The general idea is to show that around any source we can construct a ball of diameter less than 1 whose boundary intersects few edges. We then note that no region of diameter less than 1 may contain a source-sink pair. Thus, if we build such balls around every source, we end up with every source-sink pair separated by the boundary of a ball. Since we claim these boundaries intersect few edges, we have the desired cut.

We motivate this construction with an example. We claim that given any graph with m edges, we can cut it into regions of radius at most roughly $10 \log m$ with at most $\frac{1}{10}$ of the edges crossing region boundaries. The construction is as follows:

1. Pick a vertex S, and set a ball B equal to S.
2. While the number of edges leaving B is at least $\frac{1}{10}$ the number of edges in B, expand B to include its immediate neighbors: $B \leftarrow B \cup N(B)$.
3. Add B to a collection of balls $\mathcal{B}$, remove B from the graph, and repeat these steps until the graph is empty.

We have thus divided the graph into regions. We note that each stage of expanding a region increases the number of edges by a factor of $1 + \frac{1}{10}$ so that the maximum number of expansion stages a given region can take is $\log_{1+\frac{1}{10}} m \approx 10 \log m$. Since each stage merely adds to the region those points adjacent to the region, we have that the radius of the region is thus also at most roughly $10 \log m$. Also, a region stops expanding once the number of edges crossing the boundary is less than a tenth the number of edges inside it, so the boundary of any region intersects at most a tenth the number of edges in the region. Thus, the number of edges crossing any boundary is at most a tenth the total number of edges, which is the desired result.

We now show that given the *dist* metric from the relaxed problem we can construct a multicut.

Lemma 13.4. *For any $s \in V, \epsilon > 0, p > 0$, there exists $U \subseteq V$ such that*

$$
\begin{aligned}
& s \in U \\
\text{and} \quad & \text{cap}(U, \overline{U}) \leq \epsilon \left(\frac{W}{p} + \sum_{u \in U} l(uv)\text{cap}(uv) \right) \\
\text{and} \quad & \forall u \in U \quad \text{dist}(u, s) \leq \frac{\ln(p+1)}{\epsilon}.
\end{aligned}
$$

As we'll see, it is convenient to assign some initial volume to the node s, for instance the average volume per node divided by a constant, W/p. In this case,

$$G(U) \equiv \frac{W}{p} + \sum_{u \in U} l(uv)\text{cap}(uv)$$

can be thought of as the volume of all the pipes that have at least one endpoint in U; in other words, it is the initial volume of s plus the pipe volume of all the pipes within U plus the pipe volume of all the pipes that stick out of U.

Note that the upper bound on the radius of U tells us that the set U is small. This means that if l is feasible, then choosing $s = s_i, \epsilon > \ln(p+1)$ will ensure that U separates s_i and t_i, because we know that $\text{dist}_l(s_i, t_i) \geq 1$. Also, $\text{cap}(U, \overline{U})$ represents the total pipe volume of the cut, and the condition guarantees that this is small compared to the volume of U (which corresponds to $\sum_{u \in U} l(uv)\text{cap}(uv)$). As we shall see, these two facts will allow us to build cuts that are small in size and that progressively separate all demand pairs.

Proof. Let $V(x) \equiv \{v \mid \text{dist}(s, v) \leq x\}$, and define

$$F(x) \equiv \frac{W}{p} + \sum_{v,w \in V(x)} l(vw)\text{cap}(vw) + \sum_{v \in V(x), w \notin V(x)} \text{cap}(vw)(x - \text{dist}(s, v)).$$

Essentially, $F(x)$ is the "true" volume of $V(x)$: the first term is the initial volume of the node s, the first summation is the pipe volume that lies completely within $V(x)$, and the second is the fractional volume of the pipes that stick out of $V(x)$. Thus, this volume is no larger than than the volume of s plus the volume of all the pipes that have at least one endpoint in $V(x)$:

$$F(x) \leq G(V(x)).$$

Now, $F(x)$ is continuous and monotone in x. It is also differentiable except for values of x at which a vertex enters $V(x)$. We assume that the pipes are in "general position" i.e, there are no two vertices equidistant from s for any s (which could create a discontinuity). (This restriction can easily be satisfied by perturbing the function l slightly.)

Let us compute the derivative of F:

$$\frac{dF}{dx} = \sum_{v \in V(x), w \notin V(x)} \text{cap}(vw) = \text{cap}(V(x), \overline{V(x)}).$$

Suppose for the sake of contradiction that the statement of the lemma is false. Then there exist s, p and ϵ such that for all subsets U of V

$$\text{cap}(U, \overline{U}) > \epsilon \left(\frac{W}{p} + \sum_{u \in U} l(uv)\text{cap}(uv) \right).$$

This implies that for all $x \leq \ln(p+1)/\epsilon$, if we take $V(x)$ for U we have

$$\text{cap}(V(x), \overline{V(x)}) > \epsilon G(V(x)) \geq \epsilon F(x).$$

In other words,

$$\frac{dF}{dx} > \epsilon F(x).$$

Solving the differential inequality gives

$$\ln F(x) > \epsilon x + \ln F(0).$$

Substituting $\ln(p+1)/\epsilon$ for x and W/p for $F(0)$ yields

$$\ln F\left(\frac{\ln(p+1)}{\epsilon}\right) > \ln(p+1) + \ln(W/p) = \ln((1+1/p)W),$$

and so,

$$F\left(\frac{\ln(p+1)}{\epsilon}\right) > (1+1/p)W.$$

However, by construction F is bounded from above by the initial volume on the node s, W/p, plus the total volume of the system, W, yielding $F(x) \leq (1+1/p)W$. This is a contradiction, and the proof of the lemma is complete. ∎

We note that we can find such a set U by starting with s and growing the ball vertex by vertex until the conditions of the lemma are satisfied.

Taking δ to be any positive real number, p to be the number of commodities k, and fixing $\epsilon \equiv (2+\delta)\ln(k+1)$, the Lemma leads to the following algorithm:

1. Let $S = \{s_1, s_2, \ldots, s_k\}, \mathcal{U} = \emptyset$.
2. Choose $s \in S$ and apply Lemma 13.4 with s, p, and ϵ as above, obtaining us U as the cut. Add U to the collection $\mathcal{U}$.
3. Let S' be the set of all s_i for which (s_i, t_i) is not separated by $\mathcal{U}$. Set $S \leftarrow S'$.
4. If $S = \emptyset$, STOP and output $\mathcal{U}$; otherwise goto step 2.

Theorem 13.5. *$\mathcal{U}$ approximates the minimum multicut to within $O(\log k)$ of the optimal value.*

Proof. We first prove correctness of the algorithm: *Each $U \in \mathcal{U}$ separates at least one source-sink pair, and no source-sink pair is contained wholly in U.*

Consider s chosen in step 2. Let t be its corresponding sink. First, by the Lemma, $\forall x \in U$

$$\text{dist}(x, s) \leq \frac{\ln(p+1)}{\epsilon} = \frac{1}{2+\delta} < 1/2.$$

But $\text{dist}(s,t) \geq 1$, and so U separates at least one source-sink pair. Also, if U did contain a source-sink pair (x, y), we would have (by the triangle inequality),

$$\text{dist}(x, y) \leq \text{dist}(x, s) + \text{dist}(s, y) < 1/2 + 1/2 = 1,$$

but for any source-sink pair (x, y), $\text{dist}(x, y) \geq 1$, and so no source-sink pair can be wholly contained in U.

The algorithm clearly terminates within at most k steps (for at each step we remove at least one source-sink pair.) This means that $|\mathcal{U}| \leq k$. From Lemma 13.4,

$$\begin{aligned}\sum_{U\in\mathcal{U}} \text{cap}(U,\overline{U}) &\leq \sum_{U\in\mathcal{U}} \epsilon\left(\frac{W}{k} + \sum_{u\in U} l(uv)\text{cap}(uv)\right) \\ &\leq (2+\delta)\ln(k+1)\left(W + \sum_{U\in\mathcal{U}}\sum_{u\in U} l(uv)\text{cap}(uv)\right)\end{aligned}$$

Now, consider a given edge. It can be counted in at most *two* distinct $U \in \mathcal{U}$: one endpoint is deleted from S when the first U is produced; the other endpoint is deleted from S when the second U is produced. Thus, the summation in the last line above is bounded by twice the total volume of the system, $2W$. Therefore,

$$\sum_{U\in\mathcal{U}} \text{cap}(U,\overline{U}) < (2+\delta)\ln(k+1)(3W) = O(W\log k),$$

where W, the solution to the relaxed problem, is a lower bound on the minimum multicut. ∎

14 Sparsest multicut in undirected graphs

To refresh our memory, the SPARSEST MULTICUT or "Ratio Cut" problem on an instance of MCF can be defined as the following:

Find a multicut $F \subseteq E$ that minimises $\frac{\text{Cap}_{(F)}}{\text{Demand}_{(F)}}$

As before, we define an LP for the concurrent flow problem and express the sparsest cut problem as the dual of this LP.

$$\begin{aligned}\text{maximise} \quad & z \\ \text{subject to} \quad & \sum_{i=1}^{k}\sum_{e\in P_i^j} f(P_i^j) \leq \text{cap}(e) \quad \forall e \in E \\ \text{and} \quad & \textstyle\sum_j f(P_i^j) = z d_i \quad \forall i \\ & z \geq 0\end{aligned}$$

The dual gives us :

$$\begin{aligned}\text{minimise} \quad & \sum_{e\in E} l(e)\text{cap}(e) = W \\ \text{subject to} \quad & \textstyle\sum_i d_i dist(s_i,t_i) \geq 1 \\ \text{and} \quad & l(e) \geq 0 \forall e \in E\end{aligned}$$

14.1 An $O(\log k \log D)$ approximation

Using the results of Section 13.3 for minimum multicuts, we will obtain an $O(\log k \log D)$ approximation for the sparsest cut, where $D = \sum_i d_i$. To show this, we first need a lemma, due to Nabil Kahale.

Lemma 14.1. *Given $\delta_1, \delta_2, \ldots, \delta_k \geq 0, d_1, d_2, \ldots, d_k$ integers, and $D = \sum_i d_i$, then $\exists Q \subseteq \{1, 2, \ldots, k\}$ s.t*

$$\sum_{i=1}^{k} \delta_i d_i \leq H_D \sum_{i \in Q} d_i \min_{i \in Q} \delta_i$$

(Recall that H_D is the Harmonic series, $\sum_{i=1}^{D} \frac{1}{i}$).

Proof. Assume that $\delta_1 \geq \delta_2 \geq \ldots \geq \delta_k$. Let $\alpha_j = \sum_{i=1}^{j} d_i$.

Claim: $\exists j$ s.t $\delta_j \geq \frac{\sum_{i=1}^{k} \delta_i d_i}{H_D \alpha_j}$.

Proof of the claim: Assume the lemma is false, Then for all j,

$$\delta_j < \frac{\sum_{i=1}^{k} \delta_i d_i}{H_D \alpha_j}$$

Summing $d_j \delta_j$ over all j, we obtain

$$\begin{aligned}
\sum_{j=1}^{k} d_j \delta_j &< \sum_{i=1}^{k} \frac{\delta_i d_i}{H_D} \sum_{j=1}^{k} \frac{d_j}{\alpha_j} \\
&< \sum_{i=1}^{k} \frac{\delta_i d_i}{H_D} \sum_{j=1}^{k} \frac{\alpha_j - \alpha_{j-1}}{\alpha_j} \\
&< \sum_{i=1}^{k} \frac{\delta_i d_i}{H_D} \sum_{j=1}^{k} \sum_{l=\alpha_{j-1}+1}^{\alpha_j} \frac{1}{l} \\
&< \sum_{i=1}^{k} \frac{\delta_i d_i}{H_D} H_D \\
&< \sum_{i=1}^{k} \delta_i d_i
\end{aligned}$$

But this is a contradiction, therefore such a j exists. This concludes the proof of the claim. The lemma follows from the claim by choosing $Q = \{1, 2, \ldots, j\}$. ∎

Let l be the distance function yielding the optimal dual value. Set q_i to be $\text{dist}(s_i, t_i)$. Note that by construction, $\sum_i d_i q_i \geq 1$. From Lemma 14.1, we know that there exists a set $Q \in \{1, 2, \ldots, k\}$ s.t. if

$q = \min_{i \in Q} q_i$,

$$\begin{aligned} q \sum_{i \in Q} d_i &\geq \frac{\sum_{i=1}^{k} d_i q_i}{H_D} \\ &\geq \frac{1}{H_D} \end{aligned}$$

which gives

$$\sum_{i \in Q} d_i \geq \frac{1}{H_D q}$$

Define $l'(e) = l(e)/q$.

Looking back to Section 13.3, notice that $l'(e)$ yields a feasible solution to the dual of the max-sum multicommodity flow problem restricted to the set Q. After transformation, $l'(e) \geq 0 \forall e \in E$. Additionally, all $dist_{l'}(s_i, t_i)$ are greater than 1. If a $dist_{l'}(s_i, t_i)$ was less than 1, that would imply that there exists $dist_l(s_i, t_i)$ less than q. This is a contradiction of minimality of q, so the new solution is feasible. It achieves an objective function value equal to $\frac{W}{q}$. We use our earlier algorithm to produce a multicut which separates all demands $i \in Q$ and has capacity $\frac{W}{q} logk$. Therefore, we can approximate W (which is also the capacity of the sparse cut induced by l) to within $O(\frac{\log k}{q})$.

$$\frac{\text{Capacity of cut F}}{\text{Total demand separated by F}} \leq \frac{\frac{W}{q} \log k}{\frac{1}{H_D q}}$$

The optimal sparsity ratio is $\min \frac{\sum_{e \in E} l(e) \text{cap}(e)}{\sum_{i=1}^{k} d_i dist(s_i, t_i)}$. Notice that $\sum_i d_i dist(s_i, t_i) = 1$ in the optimal solution. If it was greater than 1, we could decrease some $dist(s_i, t_i)$, decrease some $l(e)$, and achieve a smaller objective function value, violating optimality. Therefore, $\sum_i d_i dist(s_i, t_i) = 1$ and we have optimal sparsity ratio $z_{OPT} = W$. Thus, we have

$$\frac{\text{Capacity of cut F}}{\text{Total demand separated by F}} \leq O(z_{OPT} H_D \log k)$$

an approximation ratio of $O(\log k \log \sum_i d_i)$, because $H_D \approx \log \sum_i d_i$.

We can convert this multicut to a cut with the same approximation ratio by observing the following:

Given a multicut $F \subseteq E$, let $\mathcal{S} = \{S_1, S_2, \ldots, S_r\}$ be the connected components of $G' = (V, E - F)$. Let $I(\mathcal{S})$ be the set of terminal pairs disconnected by the multicut. Denote

$$\rho(\mathcal{S}) = \frac{\sum_{e \in F} \text{cap}(e)}{\sum_{i \in I(\mathcal{S})} d_i}$$

From above, we know that $\rho(\mathcal{S}) = O(z_{OPT} \log k \log \sum_i d_i)$.

Let $\delta(S) = \{e = (u, v) | u \in S, v \in V - S\}$ be the edges in the cut defined by S. And let $I(S)$ be the set of terminal pairs disconnected by S. Now define for each i, $1 \leq i \leq r$,

$$\rho(S_j) = \frac{\sum_{e \in \delta(S_j)} \text{cap}(e)}{\sum_{i \in I(S_j)} d_i}$$

Now,

$$\sum_{j=1}^{r} \sum_{e \in \delta(S_j)} \text{cap}(e) = 2 \sum_{e \in F} \text{cap}(e)$$

and

$$\sum_{j=1}^{r} \sum_{i \in I(\mathcal{S}_j)} d_i = 2 \sum_{i \in I(\mathcal{S})} d_i$$

because each capacity and demand is counted twice, once for each endpoint.

Therefore, since all capacities are non-negative and all demands are positive integers, we can conclude that

$$\min_{S \in \mathcal{S}} \rho(S) \leq \rho(\mathcal{S})$$

Hence, we can choose the cut to be one of $(S_j, \bar{S}_j)$ for some j. The ratio of cut capacity to the demands separated by this cut will be bounded by $O(z_{OPT} \log k \log \sum_i d_i)$.

15 Sparsest Multicut using graph embeddings

Previously, we showed how to obtain a $O(\log k \log \sum_i d_i)$ approximation to find the sparsest cut. Here we show how to use some results on the embedding of metric spaces to improve this ratio. Observe that the dual of concurrent flow implies that the minimum of the following ratio over all possible assignments of the length function $l(e)$ is equal to z_{OPT}:

$$z_{\texttt{OPT}} = \min_{l \geq 0} \frac{\sum_{e \in E} l(e) \text{cap}(e)}{\sum_i d_i \text{dist}_l(s_i, t_i)}$$

The division by $\sum_i d_i \text{dist}_l(s_i, t_i)$ is instead of requiring this value to be at least 1. Observe that if $l(e)$ is 0/1 for all e, we get a cut and this ratio corresponds to the capacity of this cut divided by the demand separated by this cut.

The general idea of the algorithm is to start with optimum l, embed the vertices of the graph into an L_1 metric space, and then use this embedding to get a 0/1-valued $l(e)$ without changing the above ratio by more than a logarithmic factor.

Lemma 15.1. *In the optimal solution to the LP, $l(\cdot)$ is a metric.*

Proof. First of all, $l(e) \geq 0 \; \forall e \in E$, directly from the LP. Assume triangle inequality is not always true. Then there exists $l(uv) + l(vw) < l(uw)$. Notice that $l(uw)$ does not lie on the shortest path between any s-t pair because it can be shortcutted. Therefore, we can adjust $l(uw)$ to be $l(uv) + l(vw)$. Note that all s-t shortest path lengths are preserved. LP inequalities are still satisfied, so the new solution is feasible. However, the objective function value decreases, contradicting the assumption that LP solution was optimal. Therefore, triangle inequality must be satisfied. Optimal $l(\cdot)$ is a metric. ∎

In our embedding, we will use a metric function ψ that will assign a vector of $\Re^{\log^2 n}$ to each node $v \in V$. The function will approximate l(uv) as $\|\psi(u) - \psi(v)\|$ for each edge uv.

15.1 Building a length function ψ

The following lemma (which we will not prove here) can be made:

Lemma 15.2. *Given a finite metric space $(X, d), |X| = n$, there exists a function $\psi : X \to \Re^{\log^2 n}$ s.t. $\forall x, y \in S \subseteq X, |S| = k$,*

$$\frac{d(x,y)}{\Omega(\log k)} \leq \|\psi(x) - \psi(y)\|_1 \leq d(x,y)$$

Here, $\|\cdot\|_1$ is the L_1 norm -the 'Manhattan distance'- defined by $\sum_{i=1}^{d} |x_i - y_i|$ for points defined by vectors $\vec{x}$ and $\vec{y}$ in space of dimension d. We can use this approximate embedding to approximate l. We lose a factor of $O(\log k)$ in the approximation, but we do not incur any other losses. Specifically, let $(X, d) = (V, l)$, and let $S = \mathcal{D}$. Let ψ be the function built from the lemma and let $x_i = \psi(v_i), \forall i = 1 \ldots |V|$. Also denote $x_{s_i} = \psi(s_i), x_{t_i} = \psi(t_i)\ \forall (s_i, t_i) \in \mathcal{D}$. Replacing our distances metric l by L_1, we get

$$z_1 = \frac{\sum_{e=(v_i,v_j)\in E} \|x_i - x_j\|_1 \text{cap}(e)}{\sum_i d_i \|\psi(x_{s_i}) - \psi(x_{t_i})\|_1}$$

15.2 Analysis

From the previous lemma, $\|\psi(x) - \psi(y)\|_1 \leq l(xy)$, and $\|\psi(x_{s_i}) - \psi(x_{t_i})\|_1 \geq \frac{\text{dist}_{l(s_i,t_i)}}{\log k}$. Hence, $z_1 = O(z_{\texttt{OPT}} \log k)$.

Set $d = \log^2 n$. Let x_i^r be the r^{th} coordinate of x_i.

$$z_1 = \frac{\sum_{e=(v_i,v_j)\in E} \sum_{r=1}^{d} |x_i^r - x_j^r| \text{cap}(e)}{\sum_i d_i \sum_{r=1}^{d} |x_{s_i}^r - x_{t_i}^r|}$$

Exchanging summations, and since all elements are non-negative, we know that for at least one r,

$$\frac{\sum_{e=(v_i,v_j)\in E} |x_i^r - x_j^r| \text{cap}(e)}{\sum_i d_i |x_{s_i}^r - x_{t_i}^r|} \leq z_1$$

Clearly, the above equation is unchanged if ψ is translated or multiplied by a scaling constant. Hence, if we suppose all the x_i^r are bivalued i.e. $\forall i, x_i^r \in \{a, b\}, a, b \in \Re$, we can assume that $\{a, b\} = \{0, 1\}$, by applying scaling and translation operations.

Now, let $S = \{v_i | x_i^r = 1\}$. This is a cut for which $z = z_1 = O(z_{\texttt{OPT}} \log k)$.

Suppose however that x_i^r is not bivalued. Assume without loss of generality that $x_1^r \leq x_2^r \leq \ldots \leq x_n^r$. Consider $x_{i-1}^r < x_i^r < x_{i+1}^r$. If we vary the value of x_i^r within the range $[x_{i-1}^r, x_{i+1}^r]$, the function $z_1(x_i^r)$ consists of one linear function divided by another. Hence, its extreme points are achieved at the endpoints of the interval, implying that we can set $x_i^{'r} \in \{x_{i-1}^r, x_{i+1}^r\}$ s.t $z_1(x_i^{'r}) \leq z_1(x_i^r)$. Repeating this procedure, we can ensure that all the x_i^r are bivalued.

To summarize, the above method builds an $O(\log k)$ approximation of the sparsest cut, based on a metric approximation of the optimum LP solution.

16 Cuts in directed graphs

We solve the following problem:

Problem 16.1. *Given a directed graph $D = (V, E)$ with weights $w(e)$ on edges, determine a minimum weight set of edges K s.t. the graph $D' = (V, E - K)$ has no directed cycles.*

We can state this problem in two different ways: we are either maximizing the weight of the remaining edges, or minimizing the weight of the removed edges (in each case, leaving no cycles in the graph). If we are looking for an exact solution, than of course these problems are equivalent. However, approximating the solution for one problem does not necessarily give a good approximation for the other problem.

We show this using a trivial approximation algorithm. First, put all the nodes in a horizontal line, in some order. Now, some edges go right-to-left, and some edges go left-to-right. Let L be the set of edges that go right-to-left, and R be the set of edges that go left-to-right. Removing either all the edges of L or all the edges of R obviously makes the rest of the graph acyclic. We remove the lower-weight set of L and R.

This is a 2-approximation for the maximization problem, since we removed a set of edges whose combined weight is no more than half the combined weight of all the edges in E. Since the optimal solution's total weight is no more than the total weight of E, our algorithm gives a solution whose total weight is at least half that of the optimum.

However, this is not necessarily a good approximation for the minimization problem. It could be that the entire graph consists of one cycle through all the vertices. In this case, the optimal solution would be to remove only the minimum-weight edge. But our algorithm could remove a number of edges whose total weight is up to half the total weight of E.

We now modify out problem slightly:

Problem 16.2. *Given a directed graph $D = (V, E)$ with weights $w(e)$ on edges and a set of source-sink pairs (s_i, t_i), determine a minimum-weight set of edges K s.t. in the graph $D' = (V, E - K)$, every source-sink pair has the property that source and sink are in different strongly-connected components.*

Any solution to the first problem is a solution to this problem as well, since each vertex is in a different strongly-connect component if there are no cycles in the graph.

At this time, we have no satisfactory way to handle this problem. The best we can do is to work with the following variation:

Problem 16.3. (Symmetric multicommodity flow) *Given a directed graph $D = (V, E)$ with weights $w(e)$ on edges and a set of source-sink pairs (s_i, t_i), determine a minimum-weight set of edges K s.t. for every source-sink pair, the result of deleting K from E removes all paths from source to sink or all paths from sink to source.*

We can write an LP for this problem and find its dual. If we do so, we will find constraints that look like $dist(s_i, t_i) + dist(t_i, s_i) \geq 1$. Compare with the corresponding dual constraints for max-sum multicommodity flow, $dist(s_i, t_i) \geq 1$.

The technique is similiar to the case for undirected graphs. We will show that small separators exist that have small incoming/outgoing capacity, and will use these separators to construct a multicut.

Let W denote (as always) the volume of the system i.e $W = \sum_{e \in E} l(e)\text{cap}(e)$. Assume all round

trips from source to sink and back are of distance $\geq \delta$. Note that the LP for the undirected case has $\delta = 1$.

As before, we state the following lemma:

Lemma 16.4. *For any s, ϵ, $\exists U, s \in U$ s.t $\forall x \in U$, dist$(s,x) \leq \ln(k+1)\epsilon$, and*

$$cap(\Gamma^{out(U)}) \leq \epsilon(W/k + \sum_{u \in U} l(uv)cap(uv))$$

where $\Gamma^{out(U)}$ is the set of edges going out from U.

Proof. Similiar to the proof of lemma 13.4 in Section 13.3. ∎

We can prove an analogous statement for $\Gamma^{\text{in}(U)}$, the set of edges entering U.

Choose $\epsilon = 4\ln(k+1)/\delta$. Choose a source s and find the sets constructed by lemma 16.4. These are "balls" similar to those in the undirected problem. Call these be $R^i(s)$ (for the incoming-edge ball) and $R^o(s)$ (for the outgoing-edge ball). $\forall x, y \in R^i(s) \cap R^o(s), \text{dist}(x,y) \leq \delta/4$. Therefore, there are no source-sink pairs in $R^i(s) \cap R^o(s)$. Therefore, $R^i(s), R^o(s)$ cannot both have more than $k/2$ source-sink pairs. Let $U(s)$ be the smaller of these sets. Remove $U(s)$ and delete all edges entering it if $U(s) = R^i(s)$. Similarly, remove $U(s)$ and delete all edges leaving it if $U(s) = R^o(s)$.

Repeating this procedure until all sets have at most $k/2$ source-sink pairs, , we obtain a sequence of $U(s)$s. Maintain them in a list, each new set U being added in the beginning if $U = R^i$, or in the end if $U = R^o$. If R_1^i and R_2^i are both incoming-edge balls in this sequence, and R_2^i was removed after R_1^i, then there can be no edges from R_2^i to R_1^i. We can make a similar argument for outgoing-edge balls. Thus, our list is an acyclic graph on the Us.

At each step we cut edges of total volume $\epsilon(W/k + \sum_{u \in U} l(uv)\text{cap}(uv))$. Summing this over all Us, and noting that there are at most k Us, we obtain the total volume of edges cut to be

$$\sum_U 4\ln\frac{k+1}{\delta}(W/k + \sum_{u \in U} l(uv)\text{cap}(uv)) = O(W \log k/\delta)$$

At this point, all sets have at most $k/2$ unseparated source-sink pairs. In other words, we have reduced the number of unseparated source-sink pairs by a constant factor. Therefore, $O(\log k)$ iterations of this basic step will ensure that all pairs are separated. Hence, the total weight of edges cut is $O(W \log^2 k/\delta)$.

Note however that W/δ is the volume of the pipe system with all $l(e)$s scaled by δ. This is therefore a lower bound on the optimal multicut value. Hence, the above scheme yields an $O(\log^2 k)$ approximation.

Sparse cuts in directed graphs We can use the technique of Section 14 to reduce sparse cuts to multicuts, in this case on directed graphs. This yields, as before, an $O(\log^2 k \log \sum_i d_i)$ approximation to the sparsest cut ratio. Note that we cannot use the embedding technique from Section 15 (we do not have a metric space).

17 The Linear Layout Problem

In this section, we will discuss an application of the algorithm for finding an approximate sparsest cut which was presented in the previous lecture. Consider the following problem. Given a graph G, where $|G| = n$, we want to map its vertices onto the set $\{1, 2, \cdots, n\}$ of integers. We may think of it as designing a bus with n slots: each vertex corresponds to a device, edges correspond to pairs of devices that need to communicate. We want to assign devices to different slots on the bus so that devices that heavily communicate are *close* to each other. Formally, a *linear layout* of a graph G, where $|G| = n$, is a bijection $l : V(G) \to \{1, 2, \cdots, n\}$. We are interested in two measures associated with a layout:

1. Cutwidth: Given a linear layout l for a graph G, its cut width is defined as the size of the maximum cut of G. Thus, if $U_i = \{1, 2, \cdots, i\}$ and $e(X, V \setminus X)$ denotes the number of edges crossing the cut $(X, V \setminus X)$, then:
$$cutwidth(G) = max_{U_i} e(U_i, V \setminus U_i) \tag{21}$$
2. Wirelength: It is the total length of wires between all pairs of nodes. Thus,
$$wirelength(G) = \sum_{(u,v)\epsilon G} |l(u) - l(v)| \tag{22}$$

For example the layout in Figure 15 has cutwidth 3 and wirelength 8. Our goal is to generate a layout for which the cutwidth and wirelength values are as small as possible. In the subsequent sections, we will describe an algorithm which generates a layout for which both the cutwidth and the wirelength are within a factor of $O(\log^2 n)$ of the optimal.

It may be noted here that the layout problem, like the wire routing problem discussed earlier, is an important problem in VLSI circuit design. Note that in the wire routing problem, the layout of the blocks was fixed. We were trying to optimally route the wires in order to achieve the necessary interconnection. On the other hand, in this problem, the interconnects are fixed and our objective is to optimally place the blocks.

17.1 Approximating Graph Bisection

In this section, we discuss an approximation algorithm for the graph bisection problem. The algorithm developed here will be used as a subroutine in the solution of the layout problem.

The problem is as follows: Given graph G, find a set U, $|U| = \lfloor \frac{n}{2} \rfloor$ that minimizes $e(U, V \setminus U)$, the number of edges between U and $V \setminus U$. This problem is known to be NP hard. A rough idea for an approximation algorithm would be to apply the approximation algorithm from the previous lecture for min-ratio cut. If it splits the graph into two roughly equal parts, we are happy. If not, apply the algorithm to the bigger part. Keep chopping slices off the biggest chunk until the remaining part is roughly half of the original graph. Put all the small pieces into one partition and the big chunk into the second.

The problem with this approach is that we don't get the graph spliced into exact halves. To circumvent this problem, we relax our requirements slightly. A *b-balanced separator* is a set $U \subseteq V$ such that

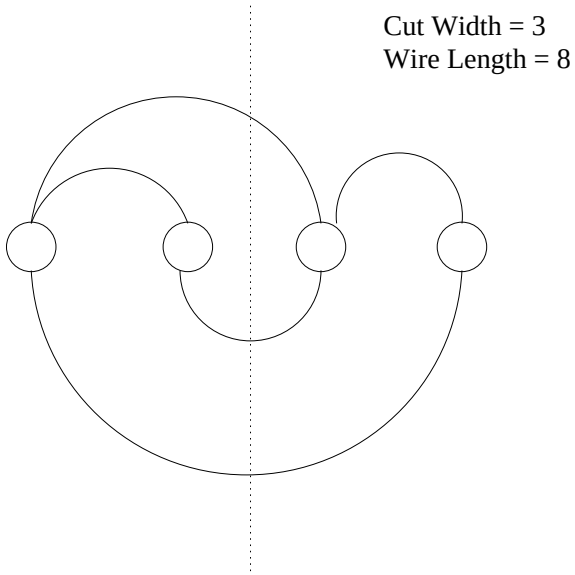


Figure 15: A Linear Layout Example

$|U|, |V \setminus U| \geq bn$.We proceed to show that an algorithm based on the idea above finds a $\frac{1}{3}-$balanced separator whose cut size is within a factor of $O(\log n)$ of the size of minimal bisection.

Our algorithm works as follows. We construct a sequence of graphs $G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_k$ such that $\frac{n}{3} \leq |G_k| \leq \frac{2n}{3}$. For each $i \geq 0$, we find an approximate min-ratio cut in the graph G_i. Set G_{i+1} to the bigger partition of G_i induced by the cut and recur. Stop when the size of G_i drops below $\frac{2n}{3}$ and call this graph G_k. Clearly, $|G_k| \geq \frac{n}{3}$. Let s_i be the size of the cut in G_i and let $\alpha_i n$ be the size of $G_i \setminus G_{i+1}$ (see Figure 16).

Let S^* be the number of edges in the minimum bisection cut. Since the size of each G_i ($i < k$) is atleast $\frac{2n}{3}$, it must contain at least $\frac{n}{6}$ nodes from each partition of the bisection of G. In other words, if we take the set of edges from the minimum bisection of G, it induces a cut in each G_i with cut ratio at most $\frac{S^*}{\frac{n}{6} \cdot \frac{n}{2}}$. This is clearly an upper bound on the ratio of the min-ratio cut. The cut ratios of our cut i is $\frac{s_i}{\alpha_i n(|G_i| - \alpha_i n)}$, which is bounded from below by $\frac{s_i}{\alpha_i n^2}$. Since this cut is within $O(\log n)$ factor of the optimal min-ratio cut, we have:

$$\frac{s_i}{\alpha_i n^2} = O(\frac{S^*}{n^2} \log n) \tag{23}$$

Rearranging and summing over all $i < k$:

$$\sum_{i=0}^{k-1} s_i \leq O(S^* \log n \sum_{i=0}^{k-1} \alpha_i) = O(S^* \log n) \tag{24}$$

Thus, we have proved that the $\frac{1}{3}-$balanced separator produced by this algorithm has a cut size which is within a factor of $O(\log n)$ of the size of minimal bisection which can be viewed as a $\frac{1}{2}-$balanced separator. We can slightly generalize it to solve the following problem: let S_b^* be the minimum number of edges in a $b-$*balanced separator* of graph G. For a $b' < b, b' \leq \frac{1}{3}$, find a $b'-$*balanced separator* with number of edges within a factor of $O(\log n)$ of S_b^*. Using basically the same analysis as above, we can show that the size of the cut found by the modified algorithm is $O(\frac{S_b^* \log n}{b - b'})$.

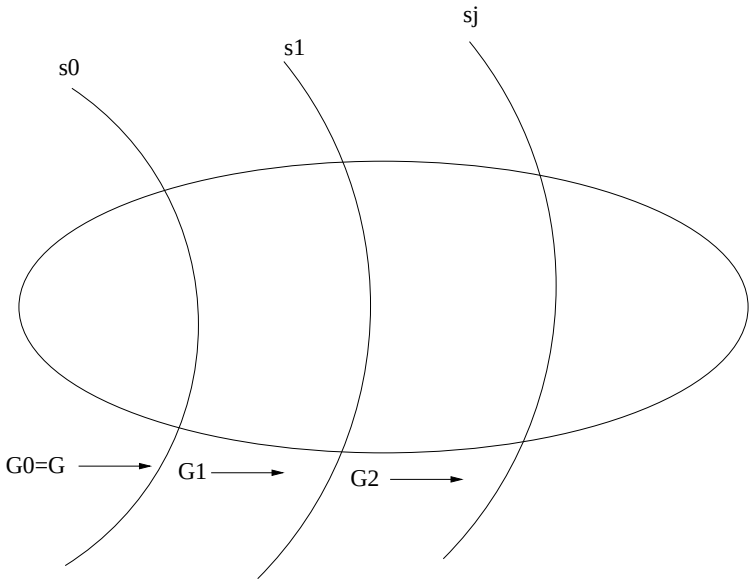


Figure 16: The Graph Sequence

17.2 The Linear Layout Algorithm

The algorithm works by finding a $b\prime-$*balanced separator* (say, $b\prime = \frac{1}{4}$) of G using the algorithm from the previous section. We then recurse on the left and right partitions induced by the cut to find an approximate optimal layout for each. The complete layout is obtained by "gluing" together the layouts for the left and right partitions. The process is illustrated in Figure 17. Observe that since $b\prime$ is a constant, the depth of the recursion tree is $O(\log n)$.

Before we analyse the approximation guarantees of this algorithm, it would be worthwile to discuss the intuition behind it. We first note that separating the graph along its min-cut and then recursing on the two partitions induced is not a very good idea. This is because while the cut size is small, it may not partition the graph too well. Consequently, the depth of the recursion tree may be $O(n)$ in the worst case. So, a balance has to be achieved between the number of edges in a cut and the number of nodes separated by that cut. A $b'-$*separator* achieves this balance as it corresponds to a cut which partitions the graph into two "nearly" equal parts *and* whose cut size is within a logarithmic factor of the minimum bisection width.

17.3 Analysis

17.3.1 Cutwidth

Let $C^*(G)$ be the minimum cutwidth of a graph G and let $S^*(G)$ be its minimum bisection. Clearly, $C^*(G) \geq S^*(G)$. Actually, $C^*(G) \geq S_{max} = max_{G' \subseteq G} S^*(G')$. In each recursive call of the algorithm, we make a cut of size $O(S_{max} \log n)$. Consider the cut between slots i and $i+1$ for some i. Let u and v be the vertices assigned to those slots. We will now bound the number of edges crossing this cut. Since our choice of i is arbitrary, this value also bounds the cutwidth.

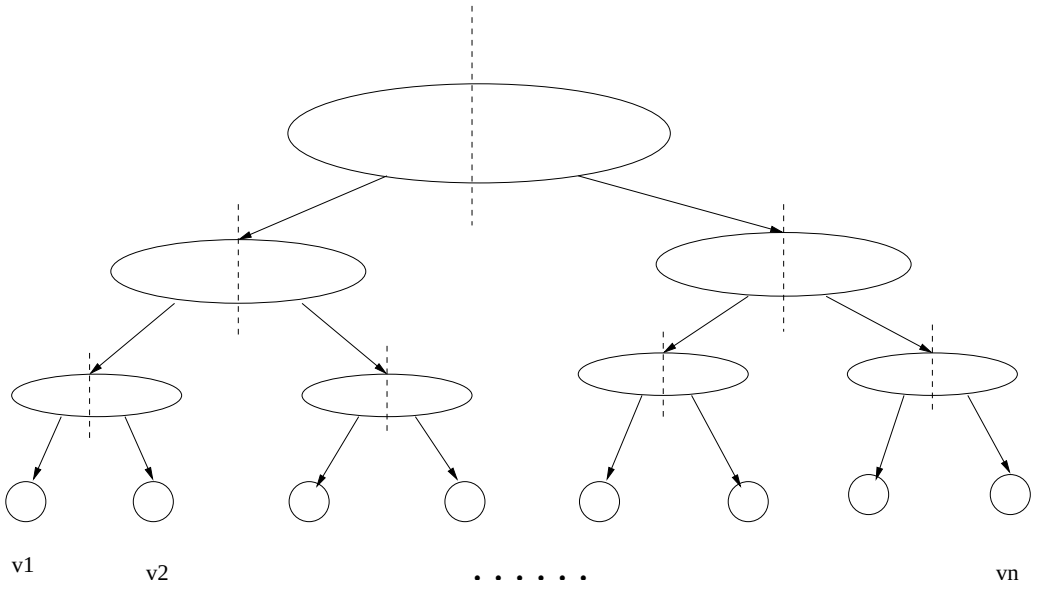


Figure 17: The Recursion Tree

Consider the following sequence of subgraphs of G: $H_k \subseteq H_{k-1} \subseteq \cdots \subseteq H_1$, where H_k is the least common ancestor of u and v and H_{i-1} is the parent of H_i in the recursion tree. Here, $k = O(\log n)$. Note that the edges passing between slots i and $i+1$ must have been cut while splitting one of the $H_i's$. Since each H_i is split by a cut of size $O(S_{max} \log n)$, it follows immediately that the number of edges crossing the cut between slots i and $i+1$ is bound by $O(S_{max} \log^2 n)$.

Theorem 17.1. *The cutwidth of the linear layout generated by the algorithm is within a factor of* $O(\log^2 n)$ *of the optimal.*

17.3.2 Wirelength

For the sake of concreteness, we assume that $b = \frac{1}{3}$ and $b' > b$ is some constant. Let $w^*(G)$ be the minimal wirelength of G and $S^*_{\frac{1}{3}}$ be the size of a minimal $\frac{1}{3}-$balanced separator of G. Then, the following two lemmas hold.

Lemma 17.2. *Let* $H_1, H_2, \cdots, H_k$ *be disjoint subgraphs of* G. *Then* $w^*(G) \geq \sum_{i=1}^{k} w^*(H_i)$.

Proof. Every layout of G gives a layout of the graph $\cup_i H_i$ with wirelength not exceeding the wirelength of G since $\cup_i H_i$ is a subgraph of G. ∎

Lemma 17.3. *For any* G, $w^*(G) \geq \frac{1}{3} n S^*_{\frac{1}{3}}$.

Proof. For any linear layout of G, consider the $\frac{n}{3}$ cuts between slots $\frac{n}{3}$ and $\frac{n}{3}+1$, $\frac{n}{3}+1$ and $\frac{n}{3}+2, \cdots, \frac{2n}{3}$ and $\frac{2n}{3}+1$. Each such cut has size atleast $S^*_{\frac{1}{3}}$ and each edge crossing the cut contributes 1 to the wire length. ∎

We are now in a position to prove the following theorem.

Theorem 17.4. *The total wire length of the linear layout generated by the algorithm is within a factor of $O(\log^2 n)$ of the optimal.*

Proof. Consider the recursion tree of G (see Figure 17). Let us denote by H_i an arbitrary subgraph at level l of the recursion tree. Note that $H_1, H_2, \cdots, H_{2^l}$ are disjoint subgraphs of G. Also, an edge in the cut that splits H_i contributes atmost $|H_i|$ to the total wirelength. Hence, the total contribution of level l to the wirelength is given by:

$$\begin{aligned}
&\sum_{i=1}^{2^l}(\text{contrib. of edges of } H_i \text{ cut at level } l) \\
&\leq \sum_{i=1}^{2^l} |H_i| \cdot (\#\text{ edges cut to separate} H_i) \\
&\leq \sum_{i=1}^{2^l} |H_i| \cdot S^*_{\frac{1}{3}}(H_i) \cdot \log n \\
&\leq \sum_{i=1}^{2^l} O(w^*(H_i)) \cdot \log n \quad \text{(from Lemma 2)} \\
&\leq O(w^*(G) \log n) \quad \text{(from Lemma 1)}
\end{aligned}$$

Since, there are $O(\log n)$ levels in the recursion tree, the total wirelength is bounded by $O(w^*(G) \log^2 n)$. ∎

18 Facility location and related problems

18.1 Introduction

Facility location problems are a class of problems in which the input is a set of points in a metric space and the goal is to open *facilities* at some subset of these points, such that every point is near some facility. Of course we could easily make such a guarantee by opening a facility at *every* point, but we are constrained in the number of facilities we may open.

These problems have wide-ranging applications including classification of data elements into clusters of similar elements, design of computer networks, and various problems in operations research.

In general, facility location problems aim to select "centers" at which to open facilities in such a way as too meet two goals:

1. All points should be near a center
2. There should not be too many centers

We will consider several different ways of formalizing these two objectives. They differ in the way in which "near" is measured and in the manner in which the number of facilities is restricted.

18.2 The k-Center Problem

18.3 The Problem Defined

In this variation of facility location problem, the input is a set of points $X = \{x_1, x_2, \ldots x_n\}$, a distance metric $dist(x_i, x_j)$, and a number of centers k. We will sometimes use i to mean the point x_i.

We are asked to place exactly k facilities (centers) in such a way as to minimize the maximum distance from any point to the nearest facility.

The k-Center variation of the facility location problem formulates the constraints as follows:

1. For every point x_i, there is a center x_j such that $dist(x_i, x_j) < D$
2. At most k points are chosen as centers.

The goal is to find the minimal possible D for which k centers can be chosen so as to satisfy the constraints, and to specify which points should be chosen as centers to satisfy the constraints with that value of D.

We'll define D^* as the value of D in the optimal solution.

18.4 An Attempt That Fails: The Greedy Approach

How might we go about solving such a problem? Suppose we know the optimum value D^*. We could apply a greedy approach. Simply place a center at some point, remove all points within D^* of that

location, and repeat. Unfortunately, this will not work very well. For example, consider the case where $k = 1$ and there are six points that are arranged in the following pattern: one point is in the center of a circle of radius D^* and the other five points are placed at equally spaced intervals along the outside of the circle. The difference between the central point and each of the points on the circle is D^*, and the distance between the points on the circle is at least $\sqrt{2 - 2\cos(2\pi/5)}D^* > D^*$.

The optimal solution chooses the center point, but the greedy approach could end up picking all five exterior points instead.

In fact the k-Center problem is NP-Hard. (It is even NP-Hard to approximate to better than a factor of 2, as we will show below.) We could solve it by trying all $O(n^k)$ possible choices of centers, but instead we will look for a polynomial-time approximation to the problem. We can modify the greedy scheme slightly to provide such an approximation.

18.5 The Greedy 2-Approximation Algorithm

Suppose again that we know the optimum value of $D^* = \max_i d(i)$. Simply place a center at some point, remove all points within $2D^*$ of that location, and repeat. When we finish, we guarantee that all points are within $2D^*$ of some center. But how many centers have we placed? We will show that we have placed at most k centers, meaning that we have a 2-approximation to the k-Center problem.

Assume that the contrary is true, and that the number of centers opened by our greedy algorithm is $m > k$. Consider some optimal solution for the k-Center problem. The optimal solution used k centers such that all points (including the m points selected for our solution) were within distance D^* of a center. Since our solution uses $m > k$ centers, then by the pigeonhole principle we know that *any* optimal solution must have some center that is within a distance D^* of more than one of our m centers. The distance between those two of our centers that are served by the same center in the optimal solution can be at most $2D^*$, by the triangle inequality. One of those two centers must have been opened first by our algorithm. However, since all points within a distance of $2D^*$ of that center were removed from consideration when that center was opened, and the other center is within a distance $2D^*$, the other center would have been removed from consideration and could not have been selected by our algorithm as a center! This is a contradiction, and so the original assumption is wrong, and in fact the solution found by our algorithm used at most k centers.

There is one thing unexplained by the algorithm: it assumes that we know the optimum value D^* from the beginning, whereas in fact this value is unknown. To get around this difficulty, we will show that D^* must be one of a relatively small number of possible values (polynomial in the number of points, n) and that therefore we can perform a binary search, running our algorithm using different possible values of D^* until we discover the correct value of D^*. (If we run our algorithm and it produces more than k centers, then our guess for D^* was too small. The smallest value of D^* for which our algorithm can identify k or fewer centers is the correct solution.)

In fact, the value of D^* must be equal to the distance between some pair of points x_i and x_j from the problem input. (There are $O(n^2)$ such pairs of points.) Suppose that the contrary were true and the value for D^* were not equal to the distance between some pair of input points. Then we could decrease D^* by some very small positive amount ϵ and the same assignment of centers would still be a solution using the $D = D^* - \epsilon$. This contradicts the assumption that D^* was actually the minimum value of D for which a solution could be found. It's only if D^* is equal to the distance between some pair of points that decreasing the value of D^* by any amount whatsoever would change some points that were previously "near" to a center into points that are "far" from all centers.

Thus we only need calculate the $O(n^2)$ distances between pairs of points, sort them, and binary search over these possible values of D^* until the true value is found, using $O(\log n)$ iterations of the algorithm.

18.6 The Linear Program Rounding Algorithm

Another approach to approximating the k-Center problem uses linear program rounding. While this approach is no better than greedy for k-Center, it is more easily extended to different types of facility location problem. We can write the following integer program for k-Center:

- For each point x_i: $y_i \in \{0, 1\}$
- For each pair of points x_i, x_j: $f_{ij} \in \{0, 1\}$
- For each pair of points x_i, x_j: $f_{ij} \leq y_i$
- For each point x_j: $\sum_{i=1}^{n} f_{ij} = 1$
- If $dist(x_i, x_j) > D^*$ then $f_{ij} = 0$
- $\sum_{i=1}^{n} y_i = k$

In this integer program, y_i will be one if we open a center at point i. If i is the closest open center to point j, then f_{ij} will be one. We have an inequality guaranteeing that the closest open center is actually open, and guaranteeing that each point has an open center within D^*. Again we will perform a binary search on the value of D^* to find a *feasible* set of constraints with the smallest D^*.

Since we cannot solve integer programs in polynomial time, we will instead solve the linear relaxation. This leads to a situation where some centers are fractionally open (have $0 < y_i < 1$) and points are fractionally assigned to many different facilities. We need to round this linear program to integrality.

First, we consider all the facilities with $y_i = 1$. If $f_{ij} > 0$ for some point j and one of these facilities i, then we set $f_{ij} = 1$ and $f_{i'j} = 0$ for all $i' \neq i$. This doesn't change the value of any y_i and so the sum of the y_is is unchanged; $f_{ij} \leq y_i$ still holds because we never increased f_{ij} past 1 and we only increased it at points where $y_i = 1$; $\sum_{i=1}^{n} f_{ij}$ is unchanged; and we didn't change any f_{ij} from zero to non-zero. Therefore, the constraints are all still satisfied.

We are now left with (possibly many) fractionally open facilities. We find some point x_j which is sent fractionally ($f_{ij} > 0$) to fractionally open facilities ($y_i > 0$). Consider all the facilities x_i for which $f_{ij} > 0$ for some such point j. $\sum y_i : f_{ij} > 0$ must be at least one for these centers, because $y_i \geq f_{ij}$ and $\sum_{i=1}^{n} f_{ij} \geq 1$. We will set $y_j = 1$ and set $y_i = 0$ for all the facilities with $f_{ij} > 0$. We then send all points within $2D^*$ of j to facility j; that is, for every point x_k such that $dist(x_j, x_k) \leq 2D^*$ we set $f_{jk} = 1$ and we set $f_{pk} = 0$ for all $p \neq j$. Notice that the value of $\sum_{i=1}^{n} y_i$ can only decrease, and every point is still sent somewhere (or in other words, $\sum_{i=1}^{n} f_{ij}$ is unchanged). The number of fractionally open facilities has decreased, and so has the number of points fractionally assigned to facilities. Therefore we can repeat this step until either no more fractionally open facilities exist (in which case any remaining fractionally assigned points can be fully assigned to any of the facilities where they are currently partially assigned) or else no more fracitionally assigned points exist (in which case any remaining fractionally open facilities can have no points assigned to them, so they can be closed).

18.7 A Lower Bound: Reduction To Set Cover

We have found two algorithms that produce a 2-approximation to k-Center in polynomial time. Is it possible to do better? In fact it is not. If we could approximate k-Center to better than a factor of two in an arbitrary metric space, then we could solve the *Set Cover* problem exactly. Since Set Cover is $NP-Hard$, it is $NP-Hard$ to approximate k-Center to better than a factor of two. (In the Euclidean plane, it is possible to improve the approximation slightly.)

In the Set Cover problem, we are given as input a set $Z = \{z_1, z_2, \ldots z_n\}$ and a set $S = \{s_1, s_2, \ldots s_m\}$ of subsets of Z and a number k. The task is to choose k subsets from S such that the union of the k subsets is equal to Z.

We can reduce the Set Cover problem to finding a 2-approximation of the k-Centers problem as follows:

Let the set of points X consist of one point for each element of Z plus one point for each element of S. Define the distance metric as follows:

- $dist(z_i, s_j) = 1$ if z_i is an element of s_j
- $dist(z_i, s_j) = 2$ if z_i is not an element of s_j
- $dist(z_i, z_j) = 2$ if $i \neq j$
- $dist(s_i, s_j) = 1$ if $i \neq j$

The value of k, the number of facilities that can be opened, is the same as k in the Set Cover problem, the number of subsets to select. (We will assume without loss of generality that each element z_i is part of at least one subset that is present in S. If this is not the case, then there is clearly no solution to that instance of Set Cover.)

Since all distances between points are either 1 or 2 we know that the optimal solution to the k-Center problem has $D^* = 1$ or $D^* = 2$.

If there is a solution to the Set Cover problem, then the optimal solution to the k-Center problem has $D^* = 1$. The centers should be placed at the points corresponding to the k subsets that solve the Set Cover problem. Since all elements of Z are part of one of these subsets, all the points that were defined based on elements of Z are distance 1 from some center, and so are all points based on elements of S since all such points are distance 1 from each other.

If there is a solution to the k-Centers problem with $D^* = 1$ then there is also a solution to the corresponding Set Cover problem. The Set Cover solution consists of the subsets whose points were chosen as centers. (If any points based on elements of Z were chosen as centers, then for each such point, pick any arbitrary $s_i \in S$ that contains the point to use in the Set Cover solution.) We know that all points z_i are within a distance of 1 of a center, so that means that the k subsets chosen as centers include all the elements of Z since points z_i are only within distance 1 of subsets that contain them.

Suppose that there were an algorithm that provided an x-approximate solution to the k-Centers problem in polynomial time where $x < 2$. In that case, we could find an exact solution to Set Cover in polynomial time by running the approximation algorithm on the k-Centers instance produced from a Set Cover instance as described above. If the algorithm produced a solution with $D \geq 2$, then we would know that there was no solution to the Set Cover instance, because the existence of a Set Cover solution implies a k-Centers solution with $D^* = 1$ and thus an x-approximation algorithm with $x < 2$

would have found a solution with $D < 2$. What about the case when the algorithm produced a solution with $D < 2$? The algorithm obviously couldn't find a solution with D better than the optimal D^*, and so if the algorithm's solution had $D < 2$ the optimal D^* could not be 2, so by process of elimination D^* must be 1, and so we can transform the algorithm's solution into a solution to Set Cover.

We have shown that a polynomial-time x-approximation algorithm for the k-Centers problem with $x < 2$ would imply that Set Cover could be solved exactly in polynomial time. Set Cover is NP-Hard, so we should not expect to find better than a 2-approximation to k-Centers. (Doing so would prove that $P = NP$.)

18.8 The Facility Location Problem

Instead of limiting the number of centers and trying to optimize the maximum distance from point to center, we can assign a cost for each point and center and try to optimize that. In the facility location problem, we can open any number of facilities but each facility opened incurs some cost. This cost is added to the sum (over all points) of the distance to the closest open facility. We will assume this cost is one for each facility for simplicity.

Facility location has certain advantages over the k-center formulation. It mirrors natural situations in clustering and network design where we are willing to create extra clusters (or servers) provided they help a large number of the data points (clients). Also, facility location is less affected by a small number of outliers; the large distance of these points will be shadowed by the sum of the many small-distance points.

Since there is no "optimum distance" for us to guess (and we cannot perform a search to find the distance for each of n points), the greedy technique for k-center will not work for facility location. Therefore, we will try the linear program rounding method. We can write the following integer program for the facility location problem:

- For each point i: $y_i \in \{0, 1\}$
- For each pair of points i, j: $f_{ij} \in \{0, 1\}$
- For each pair of points i, j: $f_{ij} \leq y_i$
- For each j: $\sum_{i=1}^{n} f_{ij} = 1$
- Minimize $\sum_i y_i + \sum_j \sum_i f_{ij} d(i, j)$

This integer program is quite similar to the one for k-center. We attempt the same rounding scheme, selecting a point and merging its facilities together. We observe that this merging can only reduce the value of $\sum_i y_i$. However, consider a point j which was nearest to some fractional facility i which has now been merged to point j'. Point j has replaced $d(i, j)$ with $d(j', j) \leq d(i, j) + d(i, j')$, but $d(i, j')$ might be very large! The $\sum_i f_{ij} d(i, j)$ part of the cost might increase by a lot.

How can we fix this problem? Suppose that for any point j, we had $d(i, j)$ about the same for all i with $f_{ij} > 0$. Then we could choose the point j with the *smallest* such distance and merge its facilities together.

We will apply a technique called *filtering* to our linear program solution. For each point j, we compute the average distance from j to its facilities: $d_j = \sum_i f_{ij} d(i, j)$. For each i, if $d(i, j) > 2d_j$ then we set $f_{ij} = 0$. Of course, we could now have $\sum_i f_{ij} < 1$ since we set some variables to zero. However, Markov's

Inequality shows that after our updates: $\sum_i f_{ij} \geq \frac{1}{2}$. We consider doubling all f_{ij}, to guarantee that $\sum_i f_{ij} \geq 1$. However, now $f_{ij} \leq y_i$ might be violated. So we must double all the y_i as well. After filtering, we produce a solution which has the property that if $f_{ij} > 0$ then $f_{ij} \leq 2d_j$ for some d_j values such that $\sum_j d_j$ is the fractional total distance from points to centers. This filtered solution has $\sum_i y_i$ equal to exactly twice its fractional value.

We now find the point j with smallest d_j. We consider closing all its facilities and instead opening a facility at j. This does not increase the value of $\sum_i y_i$, since the facilities we close must sum to at least one. Now consider all points which are sent fractionally to facilities which were closed. We will send them (completely) to j. Let j' be such a point. It is now being sent at most a distance of $d(i,j') + d(i,j)$ to j. We know that $d(i,j') \leq 2d_{j'}$ since otherwise $f_{ij'}$ would be zero. We also know that $d(i,j) \leq 2d_j$. Since we selected j with the smallest d_j, we can conclude that $d_(j,j') \leq d(i,j') + d(i,j) \leq 2d_{j'} + 2d_j \leq 4d_{j'}$. We remove point j and all points sent to the facility at j and continue.

When this process terminates, we guarantee that the value of $\sum_i y_i$ is at most its post-filtering value, which is twice the fractional value. We also guarantee that for any point j which is sent to facility i, $d(i,j) \leq 4d_j$, so $\sum_j \sum_i f_{ij} d(i,j) \leq 4 \sum_j d_j$. We can conclude that our solution has cost at most 4 times the cost of the fractional solution, and thus we have a 4-approximation in polynomial time.

Ye et. al. [12] presented a 1.52-approximation algorithm for the metric uncapacitated facility location problem, and a 2-approximation algorithm for the metric capacitated facility location problem.

18.9 The k-Median Problem

Suppose we would like to optimize the sum of distances as in the facility location problem, while maintaining that we open at most k centers. Once again, we cannot apply the greedy algorithm because there is no fixed optimum distance D^*. If we consider the linear program rounding approach, we run into the same problem where a point may be sent very far away after rounding. Can we apply filtering again? If we do, we will have to double the value of $\sum_i y_i$, meaning we will open $2k$ medians instead of only k. While there are ways to perform linear program rounding for this problem, such methods are quite complex. Instead, we will consider yet another approach to approximating NP-Hard problems, *Local Search*.

Local Search is essentially a *hill-climbing* technique. We start with some feasible solution to our problem, and we try to find some small change we can make which will improve our solution. If there is such a small change, we make it and repeat. If there is not, then we halt. Such techniques are common in the field of artificial intelligence. The problem with hill-climbing is, we may become "stuck" in a local optimum; we may find a solution which really is not very good, but such that no small change can improve it. Since we cannot search for *all* possible changes (there are exponentially many), we will be stuck. A number of heuristics are known to avoid these scenarios in practice. Of course, we would like something which produces provably good results.

For k-Median, we will use the following local search algorithm. We start with some set of k medians. We can compute the cost of our solution by finding the closest median to each point. We will search for a swap $< i, j >$ which replaces some median i which we have opened with some other median j. Since there are k medians we have opened and n total demand points, there are only $k(n-k)$ possible swaps and we can test them all in polynomial time. If there is a swap which improves the cost of our solution (reduces the sum of distances) then we will make that swap and continue. If not, we will halt. We need to show that when this process finishes, we have a good solution; it is also not clear that we will terminate in a polynomial number of steps.

Let us consider what happens when the process finishes. We have some solution such that no swap can improve it. Suppose at this time our medians are $m_1, m_2, ...m_k$ and the optimum medians are $m_1^*, m_2^*, ...m_k^*$. We will construct a set of possible swaps, none of which can improve our solution.

We will say that our facility m_i *captures* optimum facility m_j^* if at least half the points which optimum sends to m_j^* are sent to m_i in our solution. Notice that one of our centers might capture *many* optimum centers, but any single optimum center can be captured by *at most one* of ours.

Our medians can be divided into three groups. There are those which capture no optimum medians, those which capture one, and those which capture two or more. Suppose there are M_0 in the first group, M_1 in the second, and M_2 in the third. Of course $M_0 + M_1 + M_2 = k$. For each of our medians i which captures *exactly one* optimum median j, we will construct a swap $< m_i, m_j^* >$. There are now $M_0 + M_2$ of our medians which have not appeared in a swap, and the same number of optimum medians which have not appeared in a swap. Notice that $M_1 + 2M_2 \leq k$ since no optimum median can be captured by more than one of ours. This means $M_1 + 2M_2 \leq M_0 + M_1 + M_2$ so $M_2 \leq M_0$. Thus, there are at most $2M_0$ left over optimum medians which are not involved in a swap. For each of our medians which capture zero optimum ones, we will define two swaps. We can do this in such a way that our set of swaps has the following properties:

1. Each optimum median appears in exactly one swap
2. Each of our medians appears in at most two swaps
3. If we have some swap $< m_i, m_j^* >$ then there does not exist any $j' \neq j$ such that m_i captures $m_{j'}^*$.

None of the swaps in the set we have constructed could possibly improve our solution; we assumed we had reached a local optimum. Let's look at some particular swap $< m_i, m_j^* >$. All the points which the optimum solution sends to center m_j^* can now be sent there, paying the optimum cost. On the other hand, there are some points which we sent to center m_i which now have to be sent elsewhere. We will describe one possible way to send these points elsewhere; since the actual algorithm will send them to the closest open center it can only do better.

For each point x which is sent to our center m_i and to some optimum center m_j^* which is not captured by m_i, we will define $\pi(x)$ such that $\pi(x)$ is also sent to m_j^* by the optimum solution but is sent to some $m_{i'}$ by our solution. Because of the non-capture assumption, there must exist enough points with the desired properties for us to design a one-to-one mapping π.

When center m_i is swapped out, we will consider sending x to $m_{i'}$ along the path from x to m_j^* to $\pi(x)$ to $m_{i'}$. The total cost of this looks like $OPT(x) + OPT(\pi(x)) + OUR(\pi(x))$ where OPT represents the optimum distance for a point and OUR represents the distance in our solution.

What is the total change in the distances if we perform a swap $< m_i, m_j^* >$? We can write it out as at most:

$$\sum_{x \in m_j^*} [OPT(x) - OUR(x)] + \sum_{x \in m_i} [OPT(x) + OPT(\pi(x)) + OUR(\pi(x)) - OUR(x)] \geq 0$$

If we sum this over all k of our defined swaps, we observe that each x appears in the first summation exactly once. Each of our centers may appear in two swaps, so a single x might appear in the second summation twice. Noticing that $\pi(x)$ is one-to-one, we can conclude that:

$$OPT - OUR + 2(OPT + OPT + OUR - OUR) = 5OPT - OUR \geq 0$$

From this a five-approximation follows.

Is the algorithm polynomial time? Actually not, since we might make many small improvements to the solution. Each improvement (local swap) is polynomial time but there could be many of them. This may be fixed by searching for swaps which improve the solution by a factor of $1 + \epsilon$ instead of simply swaps which improve the solution. This yields a $5 + \epsilon$ approximation to optimum.

The algorithm can be improved by permitting p for p swaps instead of only one for one. This increases the running time since we have more possible swaps to search over. The approximation ratio improves to $3 + \frac{2}{p} + \epsilon$.

References

[1] V. Chvátal. *Linear Programming.* W.H. Freeman and Co., New York, 1980.

[2] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[3] H. Gabow. Scaling algorithms for network problems. *J. Computer and System Sc.*, 31:148–168, 1985.

[4] A. Goldberg and R. Tarjan. Finding minimum-cost circulations by cancelling negative cycles. *Journal of the ACM*, 36:873–886, 1989.

[5] C. C. Gonzaga. Polynomial affine algorithms for linear programming. *Mathematical Programming*, 49:7:21, 1990.

[6] L. G. Khachiyan Grigoriadis, M. D. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optimization*, 4:86–107, 1991.

[7] É. Tardos J.K. Lenstra, D. B. Shmoys. Approximation algorithms for scheduling unrelated parallel machines. *Math. Programming*, 24:259–272, 1990.

[8] L.G. Khachian. A polynomial algorithm for linear programming. *Doklady Akad. Nauk USSR*, pages 1093–1096, 1979. English translation: *Soviet Math. Doklady*, 20, no.1, pp.191–194.

[9] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problem. In *Proc. 23th ACM Symposium on the Theory of Computing*, pages 101–111, May 1991.

[10] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50:228–243, 1995.

[11] A.Y. Levin. On an algorithm for convex functions minimization. *Doklady Akad. Nauk USSR*, 160:1244–1247, 1965. English translation: *Soviet Math. Doklady* 6, pp.286–290.

[12] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Improved approximation algorithms for metric facility location problems. In *In Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 229–242, 2002.

[13] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993.

[14] J. B. Orlin, S. A. Plotkin, and É. Tardos. Polynomial dual network simplex algorithms. *Mathematical Programming*, 60:255–276, 1993.

[15] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[16] S. Plotkin, D. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, pages 495–504, October 1991.

[17] S. Plotkin, D. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math of Oper. Research*, 20(2):257–301, 1995.

[18] F. Shahrokhi and D. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318–334, 1990.

[19] D. Shmoys and É. Tardos. Scheduling unrelated machines with costs. In *4th Symposium on Discrete Algorithms*, pages 448–454, 1993.

[20] N.Z. Shor. Utilization of the operation of space dilatation in the minimization of convex functions. *Kibernetika*, 6:6–12, 1970. English translation: *Cybernetics* 13, pp.94–96.

[21] B. Yamnitsky and L. Levin. An old linear programming algorithm runs in polynominal time. In *Proc. of 23rd IEEE Symp. on Foundation of Computer Science*, pages 327–328, 1982.

[22] D.B. Yudin and A.S. Nemirovskii. Informational complexity and effective methods for solving convex extremum problems. *Econ. i. Math. Metody*, 12:357–369, 1976. English translation: *Maketon* 13 (3) (1977), pp. 25-45.