CS261 - Optimization Paradigms
Lecture Notes for 2012-2013 Academic Year

©Serge Plotkin

January 2013

Contents

1 Steiner Tree Approximation Algorithm **6**

2 The Traveling Salesman Problem **10**

2.1 General TSP 10

2.1.1 Definitions 10

2.1.2 Example of a TSP 10

2.1.3 Computational Complexity of General TSP 11

2.1.4 Approximation Methods of General TSP? 11

2.2 TSP with triangle inequality 12

2.2.1 Computational Complexity of TSP with Triangle Inequality 12

2.2.2 Approximation Methods for TSP with Triangle Inequality 13

3 Matchings, Edge Covers, Node Covers, and Independent Sets **22**

3.1 Minumum Edge Cover and Maximum Matching 22

3.2 Maximum Independent Set and Minimum Node Cover 24

3.3 Minimum Node Cover Approximation 24

4 Intro to Linear Programming **27**

4.1 Overview 27

4.2 Transformations 27

4.3 Geometric interpretation of LP 29

4.4 Existence of optimum at a vertex of the polytope 30

4.5 Primal and Dual 32

4.6 Geometric view of Linear Programming duality in two dimensions 33

4.7 Historical comments 35

5 Approximating Weighted Node Cover **37**

5.1 Overview and definitions 37

5.2 Min Weight Node Cover as an Integer Program 37

5.3 Relaxing the Linear Program 37

5.4 Primal/Dual Approach . . . 38
5.5 Summary . . . 40

6 Approximating Set Cover **41**

6.1 Solving Minimum Set Cover . . . 41

7 Randomized Algorithms **46**

7.1 Maximum Weight Crossing Edge Set . . . 46
7.2 The Wire Routing Problem . . . 48
7.2.1 Overview . . . 48
7.2.2 Approximate Linear Program Formulation . . . 48
7.2.3 Rounding . . . 49
7.2.4 Analysis . . . 49

8 Introduction to Network Flow **53**

8.1 Flow-Related Problems . . . 53
8.2 Residual Graphs . . . 56
8.3 Equivalence of min cut and max flow . . . 57
8.4 Polynomial max-flow algorithms . . . 59
8.4.1 Fat-path algorithm and flow decomposition . . . 59
8.4.2 Polynomial algorithm for Max-flow using scaling . . . 62
8.4.3 Strongly polynomial algorithm for Max-flow . . . 62
8.5 The Push/Relabel Algorithm for Max-Flow . . . 65
8.5.1 Motivation and Overview . . . 65
8.5.2 Description of Framework . . . 65
8.5.3 An Illustrated Example . . . 68
8.5.4 Analysis of the Algorithm . . . 70
8.5.5 A better implementation: Discharge/Relabel . . . 74
8.6 Flow with Lower Bounds . . . 77
8.6.1 Motivating example - “Job Assignment” problem . . . 77
8.6.2 Flows With Lower Bound Constraints . . . 78

9 Matchings, Flows, Cuts and Covers in Bipartite Graphs **83**

9.1 Matching and Bipartite Graphs . . . 83

9.2 Finding Maximum Matching in a Bipartite Graph . . . 84

9.3 Equivalence Between Min Cut and Minimum Vertex Cover . . . 86

9.4 A faster algorithm for Bipartite Matching . . . 88

9.5 Another $O(m\sqrt{n})$ algorithm that runs faster in practice . . . 90

9.6 Tricks in the implementation of flow algorithms. . . . 92

10 Partial Orders and Dilworth's Theorem **94**

10.1 Partial orders . . . 94

10.2 Dilworth Theorem . . . 95

11 Farkas Lemma and Proof of Duality **99**

11.1 The Farkas Lemma . . . 99

11.2 Alternative Forms of Farkas Lemma . . . 99

11.3 Duality of Linear Programming . . . 101

12 Examples of primal/dual relationships **104**

12.1 Complementary Slackness . . . 104

12.2 Maximum bipartite matching . . . 104

12.3 Shortest path from source to sink . . . 105

12.4 Max flow and min-cut . . . 106

12.5 Multicommodity Flow . . . 108

13 Online Algorithms **111**

13.1 Introduction . . . 111

13.2 Ski Problem and Competitive Ratio . . . 111

13.3 Paging and Caching . . . 113

13.3.1 Last-in First-out (LIFO) . . . 113

13.3.2 Longest Forward Distance (LFD) . . . 113

13.3.3 Least Recently Used (LRU) . . . 115

13.4 Load Balancing . . . 117

13.5 On-line Steiner trees 118

1 Steiner Tree Approximation Algorithm

Given a connected graph $G = (V, E)$ with non-negative edge costs, and a set of "special" nodes $S \subset V$, a subgraph of G is a Steiner tree, if it is a tree that spans (connects) all the ("special") nodes in S.

The Steiner Tree problem is to find a Steiner Tree of minimum weight (cost).

Steiner Tree is an important NP-hard problem that is often encountered in practice. Examples include design of multicast trees, design of signal distribution networks, etc. Since it is NP-hard, we cannot expect to design a polynomial-time algorithm that solves it to optimality. In this lecture we will describe a simple polynomial time algorithm that produces an answer that is within a factor 2 of optimum.

Since a minimum cost spanning tree (MST) of the given graph is a Steiner tree, the intuitively easiest approximation is to find the MST of the given graph. Unfortunately, there are graphs for which this is a bad choice, as illustrated in fig. (1).

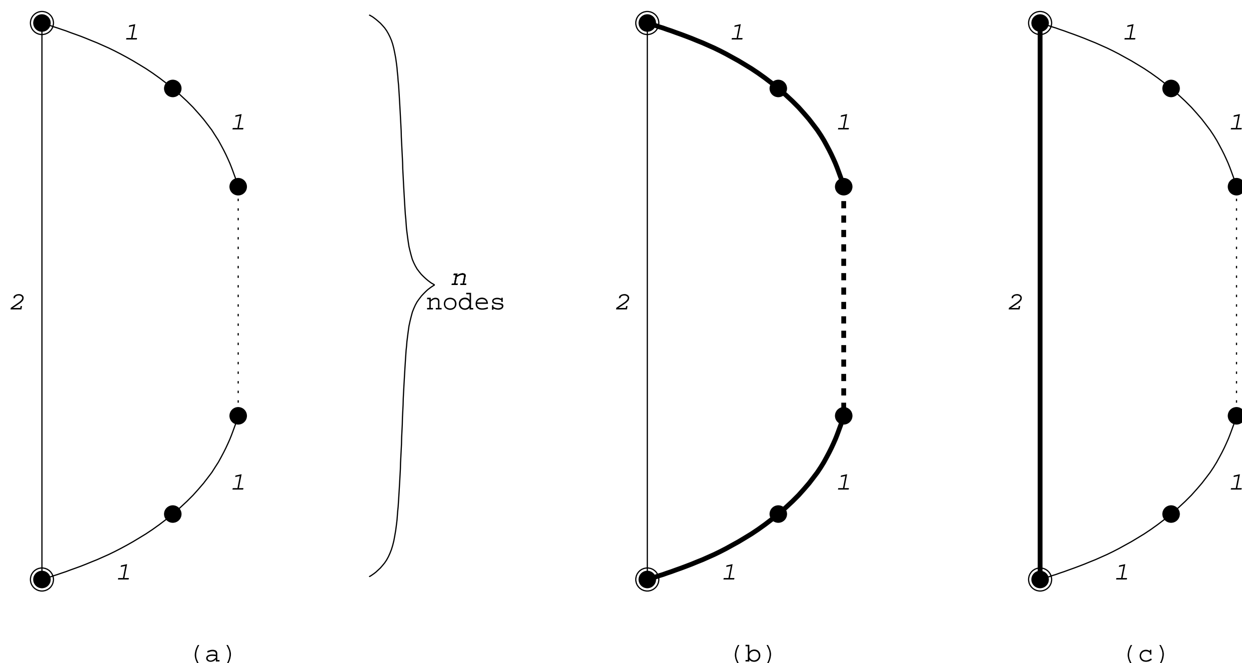


Figure 1: An example showing that MST is not a good heuristic for the minimum Steiner tree problem. *(a)* The original graph. Circled nodes are the special nodes. *(b)* The minimum cost spanning tree. *(c)* The minimum cost Steiner tree.

In this graph, the MST has weight $(n - 1)$. Also, in this case, the MST is minimal in the Steiner tree sense, that is, removing any edge from it will cause it to not be a Steiner tree. The minimum cost Steiner tree, however, has weight 2. The approximation is therefore only good to a factor of $(n - 1)/2$, which is quite bad.

Next we will describe a better approximation algorithm, which is guaranteed to produce a Steiner tree no worse than 2 times the optimal solution. The algorithm to find an approximate optimal Steiner tree for a graph $G = (V, E)$ and special nodes S is as follows:

1. *Compute the complete graph of distances between all of the special nodes.* The distance between two nodes is the length of the shortest path between them. We construct a complete graph $G' = (V', E')$, where V' corresponds to the special nodes S, and the weight of each edge in E' is the distance between the two corresponding nodes in G, i.e., the length of the shortest path between them.

2. *Find the MST on the complete graph of distances.* Let T' be the MST of G'.

3. *Reinterpret the MST in the original graph.* For each edge in T', add the corresponding path in G to a new graph G''. This corresponding path would be a shortest path between the two corresponding special nodes. G'' is not necessarily a tree yet is a subgraph of the original graph G.

4. *Find a spanning tree in the resulting subgraph.* T_{approx}, a spanning tree of G'' is an approximately optimal Steiner tree.

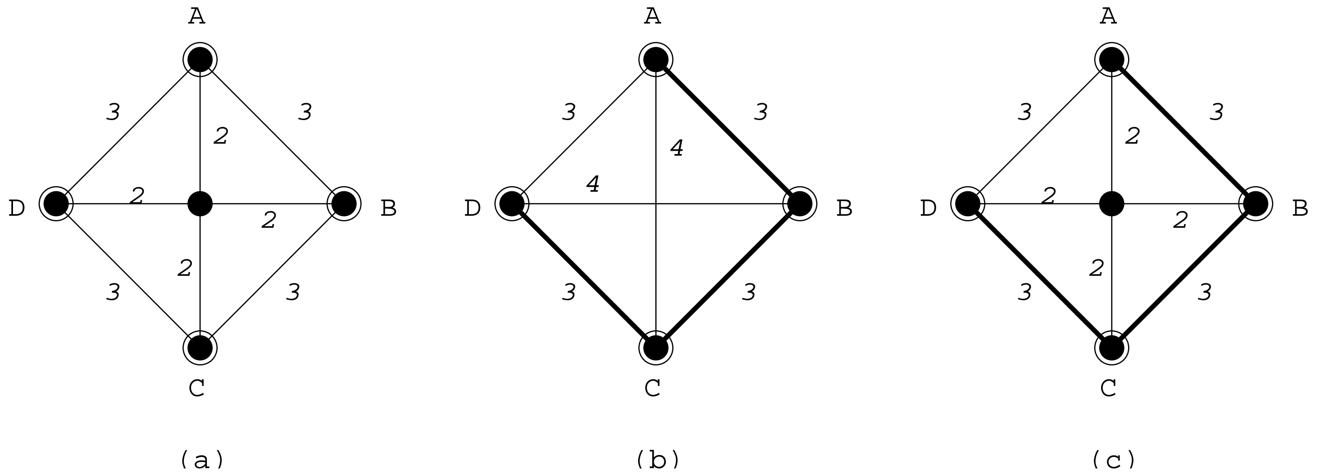


Figure 2: The Steiner tree approximation algorithm run on the graph in *(a)*. Circled nodes are the special nodes. *(b)* The corresponding complete graph of distances. The minimum spanning tree is drawn in bold. *(c)* The minimum spanning tree of the graph of distances reinterpreted as a subgraph of the original graph (drawn in bold). In this example, it is it's own spanning tree.

Theorem 1.1. *The above algorithm produces a Steiner tree.*

Proof. The algorithm returns a tree which is a subgraph of G, because step 4 returns a tree T_{approx} which is a subgraph of the reinterpreted MST G'', which is a subgraph of G.

To see that T_{approx} spans all the nodes in S, see that V' has a node corresponding to each node in S, and thus that the MST of G', T' spans nodes corresponding to all the nodes in S. Thus, after reinterpretation in step 3, G'' spans all the nodes in S. Since T_{approx} is a spanning tree of G'', it too spans all the nodes in S.

Thus, T_{approx} is a tree, a subgraph of G, and spans S. Thus, by definition, it is a Steiner tree.

∎

Theorem 1.2. *The Steiner tree returned by the above algorithm has weight at most twice the optimum.*

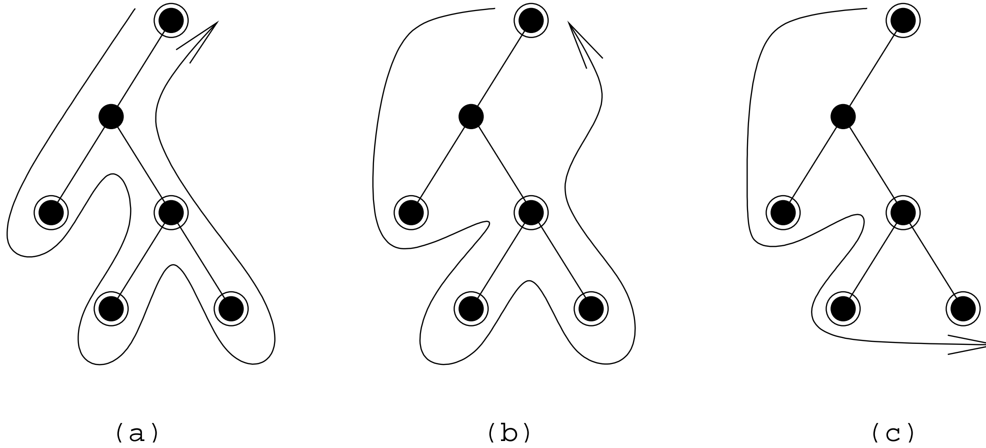


Figure 3: *(a)* A depth-first search walk on an example optimal tree T_{opt}. *(b)* The effect of taking only special nodes to form W'. (This is not W', but the special nodes corresponding to W', in order.)*(c)* The effect of dropping repeated nodes to form W''. (These are the special nodes corresponding to W'', in order.)

Proof. To show this, consider a minimum cost Steiner tree, T_{opt}, with weight OPT.

Now, do a depth-first search walk on this optimum tree T_{opt}. This can be described recursively by starting from the root, moving to a child, performing an depth-first search walk on that subtree, returning to the root, and continuing to the next child until there are no children left. Call this walk W.

Each edge in the tree is followed exactly twice: once on the path away from a node, and once on the path back to the node. The total weight of the walk $w(W) = 2OPT$.

Next, we relate this to the complete graph of distances G'. We replace the walk W in G by a walk W' in G', the complete graph of distances, in the following manner. We follow the original walk, and every time a special node is encountered we put this in the walk in the distance graph. One can view this operation as "shortcutting".

Consider a part of W between two successive special nodes. This is a path in G. The corresponding edge in W' is of the weight of the minimum path in G between the two special nodes, by construction in step 1. Thus, each edge of W' is of weight no greater than the corresponding path in W. Thus, the total weight of the new walk, $w(W') \leq w(W) = 2OPT$.

Next, remove duplicate nodes, if any, from W' to get W''. Obviously, $w(W'') \leq 2OPT$.

Now, since T_{opt} spanned all the special nodes, so did W. Thus, W' and hence W'' span all the nodes in G'. Thus, W'' is a spanning tree of G'. Since T' is the minimum spanning tree of G', $w(T') \leq w(W'') \leq 2OPT$.

The reinterpretation of T' in step 3 adds edges of cumulative weight exactly equal to the edge of T' being considered, to G''. Thus, the total weight of G'' cannot be greater than that of T'. It could be less, though, because an edge might be present in G'' due to more than one edges of T and yet contribute it's weight only once to G''. Thus, $w(G'') \leq w(T') \leq 2OPT$.

Step 4 will just delete some edges from G'', possibly reducing it's weight further, to give T_{approx}. Thus, $w(T_{approx}) \leq w(G'') \leq 2OPT$. ∎

To summarize, a good way to compute the approximation bounds of an algorithm is to manipulate the optimal solution somehow in order to relate it to the output of the algorithm. In the present case, we related the weight of the Steiner tree given by the algorithm to the optimal Steiner tree, through a depth-first search walk of the optimal Steiner tree.

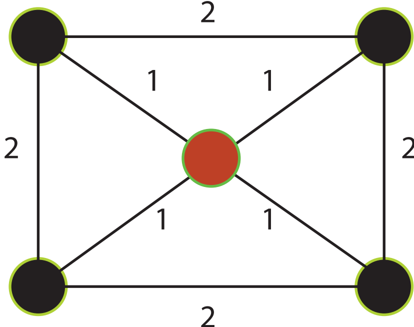


Figure 4: This example shows that the above algorithm sometimes indeed produces suboptimal solution. In this example, black nodes are "special" and red node is not. The algorithm described above will produce a tree of weight 6, while the optimum is clearly 4, giving 1.5 approximation ratio. (How would you generalize this example to get $2(n-1)/n$ ratio for any n ?)

Notes The algorithm presented in this section is from [8]. Best approximation factor for Steiner Tree problem is 1.39, presented in [2]. See also [13]. It is knowns that unless P=NP, it is impossible to produce an approximation factor better than $(1+\epsilon)$ for some constant ϵ.

2 The Traveling Salesman Problem

In this section we discuss several variants of the Travelling Salesman Problem (TSP). For the general TSP, we know that the problem is NP-hard. So we ask the natural question: can we find a reasonable approximation method? In this section we prove that the answer is no: there is no polynomial-time algorithm that solves the general TSP to within a given constant factor of the optimal solution.

If we constrain the general TSP by demanding that the edge weights obey the triangle inequality, we still have an NP-hard problem. However, now we *can* find reasonable approximation methods. Two are presented: one which obtains approximation factor of 2 and a more complicated one that achieves approximation factor of 1.5.

2.1 General TSP

2.1.1 Definitions

Before defining TSP, it is useful to consider a related problem:

Definition 2.1. Hamiltonian Cycle *is a cycle in an undirected graph that passes through each node exactly once.*

Definition 2.2. *Given an undirected* complete *weighted graph, TSP is the problem of finding a minimum-cost Hamiltonian Cycle.*

Let $G = (V, E)$ be a complete undirected graph. We are also given a weight function w_{ij} defined on each edge joining nodes i and j in V, to be interpreted as the cost or weight (of traveling) directly between the nodes. The general Travelling Salesman Problem (TSP) is to *find a minimal weight tour through all the nodes of G.* A tour is a cycle in the graph that goes through every node once and only once. We define the weight of a tour as the sum of the weights of the edges contained in the tour. Note that in general the function w_{ij} need not be a distance function. In particular, it need not satisfy the triangle inequality. For this section, we constrain w_{ij} to be nonnegative to make the proofs less complicated. All the results we show can also be proved when negative weights are allowed.

2.1.2 Example of a TSP

Here is a specific example of where the TSP arises. Suppose we are drilling some holes in a board in our favorite machine shop. Suppose it takes time p_j to drill the hole at position j and time w_{ij} to move the drill head from position i to position j. We wish to minimize the time needed to drill all of the holes and return to the initial reference position.

The solution to this problem can be represented as a permutation π, which represents the order in which the holes are drilled. The cost of the solution (the time required for drilling all holes) is

$$time = \sum_i (w_{i\pi(i)} + p_{\pi(i)}) = \sum_i w_{i\pi(i)} + \sum_i p_{\pi(i)};$$

Since p_i is fixed for each i, $\sum_i p_{\pi(i)}$ is the same for any permutation π of the holes. This part of the cost is fixed, or may be considered a sunk cost, so the objective is to minimize the sum of the $w_{i\pi(i)}$ terms, which is done by formulating the problem as a TSP and solving it.

2.1.3 Computational Complexity of General TSP

Recall that algorithmic complexity classes apply to *decision* problems: those with a yes/no type of answer. An algorithm to solve the TSP problem might return a minimum cost tour of an input graph, or the cost of such a tour. We can formulate a decision problem version of TSP as follows: We ask "Does the minimum cost tour of G have a cost less than k?" where k is some constant we specify.

The complexity class NP may be casually defined as the set of all problems such that solutions to specific instances of these problems can be verified in polynomial time. We can see that TSP is in NP, because we can verify whether the cost of a proposed tour is below a specified cost threshold k in polynomial time. Given any tour of the graph G, we can verify that the tour contains all of the nodes, can sum up the cost of the tour, and then compare this cost to k. Clearly these operations can be performed in polynomial time. Thus, TSP is in NP.

It turns out that the general TSP is, in fact, NP-complete (that is, it is in NP and all problems of class NP polynomially transform to it). Recall that since all NP-complete problems have polynomial transformations to TSP, if we could solve TSP in polynomial time we could solve all NP-complete problems in polynomial time. Since NP-complete problems have been studied for a long time and no poly-time solutions have been found, seeking a poly-time solution to TSP is very likely to be fruitless.

The proof of NP-completeness is not presented here but is similar to the proof in the next section and can be found in [4].

2.1.4 Approximation Methods of General TSP?

First we show that one cannot expect to find good approximation algorithms for general TSP. More formally:

Theorem *No polynomial time algorithm that approximates TSP within a constant factor exists unless P=NP.*

A general strategy to prove theorems of this type is to prove a polynomial-time reduction from a hard problem to our problem of interest. If we can quickly convert any instance of the hard problem to an instance of our problem, and can quickly solve any instance of our problem, then we can solve the hard problem quickly. This produces a contradiction if we know (or assume) that the hard problem cannot be solved quickly.

Proof. Suppose that we have a polynomial-time algorithm A that approximates TSP within a contant factor r. We show that A can be used to decide the Hamiltonian Cycle problem, which is known to be NP-complete in polynomial time. But unless P=NP, this is impossible, leading us to conclude A doesn't exist.

Consider an instance of Hamiltonian cycle problem in graph G. As defined above, the Hamiltonian cycle problem is the problem of finding a cycle in a graph $G = (V, E)$ that contains every node of G once and only once (in other words, a tour of G). The catch is that, unlike in the TSP problem, G is not guaranteed to be a complete graph. Note that it is trivial to find a Hamiltonian Cycle in a complete graph: just number the nodes in some arbitrary way and go from one node to another in this order.

Modify $G = (V, E)$ into a complete graph $G' = (V, E')$ by adding all the missing edges. For the cost function, if an edge is in the original graph G (meaning $e \in E$), assign to it weight $w(e) = 1$. If not (i.e. we had to add it to G to make G' complete, so $e \notin E$), give it weight $w(e) = r * n + 1$.

- If G has a Hamiltonian cycle, then this cycle contains the same edges as the minimum cost TSP solution in G'. In this case, the minimum cost tour's length is equal to n, and the algorithm A on G' will give an approximate solution of at most $r * n$.
- If G does not have a Hamiltonian cycle, then any TSP solution of G' must include one of the edges we added to G to make it complete. So, an optimal solution must have weight at least $r * n + 1$. Clearly, the weight of the cycle produced by A on G' must be at least $r * n + 1$, since the approximate solution cannot beat the optimal one.

Now notice that we can use A on G' to find whether G contains a Hamiltonian cycle. If A gives a solution of at most $r * n$, then G has a Hamiltonian cycle. If not, then we can conclude G does not have a Hamiltonian cycle. Since constructing G' and running A can be done in polynomial time, we have a polynomial-time algorithm which tests for the existence of a Hamiltonian cycle. Since the Hamiltonian cycle problem is known to be NP-complete, this is a contradiction under the assumption P≠NP, so such an A does not exist.

∎

2.2 TSP with triangle inequality

One tactic for dealing with hard problems is to use the following rule: "if you can't solve a problem, change it".

Let us consider the following modification to the TSP problem: we require that for any three nodes i, j, and k, the *triangle inequality* holds for the edges between them:

$$w_{ij} \leq w_{ik} + w_{kj} \qquad \forall \;\; i, j, k \epsilon V;$$

in other words, it is not faster to "go around". It turns out that with this constraint we can find good approximations to the TSP.

First, let's consider the usefulness of solving this modified TSP problem. Consider the drill press example described earlier. Upon initial examination, this problem seems to obey the triangle inequality because the drill moves between points on a plane. Suppose we want to move the drill from point A to point C. Further suppose there is an obstruction on the straight path from A to C, so we go through point B on the way from A to C (where B is not collinear with line AC). If the cost of going from A to C through B is equal to $w_{AB} + w_{BC}$, then the triangle inequality is obeyed with equality. However, if this were a real scenario, the drill might need to spend some time at B changing direction. In this case, moving from A to B to C would cost more than the summed cost of moving from A to B and moving from B to C, so the triangle inequality would not hold. In the majority of problems for which TSP is used the triangle inequality holds, but it is important to recognize cases in which it does not hold.

2.2.1 Computational Complexity of TSP with Triangle Inequality

Theorem TSP with triangle inequality is NP-hard.

Proof. Suppose we have a TSP problem without triangle inequality. Let us add some fixed value Q to the weight of each edge in the graph. If we do so, assuming the graph has n nodes, the total solution

weight will increase by $n * Q$ for all solutions. Clearly, this modification cannot change which path the TSP solution takes, since we increase the weights of all paths equally.

Now let us consider all triples of edges ($\{i, j\}$, $\{i, k\}$, and $\{k, j\}$) in the graph that violate the triangle inequality, and let us define

$$q_{ikj} = w_{ij} - w_{ik} - w_{kj}$$

We choose the largest value of q_{ikj} over all edge triples as the value Q that we will use.

What happens when we add this value of Q to the weight of each edge in the graph? Assuming that w'_{ij} denotes $w_{ij} + Q$ for an arbitrary edge $\{i, j\}$, we have:

$$w'_{ij} = w_{ij} + Q, \quad w'_{ik} = w_{ik} + Q, \quad w'_{kj} = w_{kj} + Q.$$

Now let us substitute the value of Q in the sum $w'_{ik} + w'_{kj}$:

$$w'_{ik} + w'_{kj} = w_{ik} + Q + w_{kj} + Q \geq w_{ij} + Q = w'_{ij}.$$

We have just demonstrated that adding a certain value Q to the weights of all edges removes triangle inequality violations in any graph. This transformation can be done in polynomial time since the number of triangles in a graph is polynomial in the graph size. So, we can reduce a TSP without triangle inequality (the general TSP problem) to a TSP with triangle inequality in polynomial time. Since general TSP is NP-complete, this completes the proof. ∎

Notice how our proof that constant-factor approximation of TSP is NP-hard does not apply to TSP with the triangle inequality restriction. This is because the triangle inequality may be violated in the graphs G' constructed by the procedure specified in the proof. Consider, for example,

$$w_{ij} = w_{ik} = 1, \quad and \quad w_{kj} = r * n + 1.$$

This would occur if edges $\{i, j\}$ and $\{i, k\}$ were in G but edge $\{k, j\}$ was not. Since our graph construction procedure does not in general produce graphs that satisfy the triangle inequality, we cannot use this procedure to reformulate any instance of the Hamiltonian cycle problem as an instance of the TSP approximation problem with triangle inequality.

In fact, there is no such polynomial time reformulation of the Hamiltonian cycle problem, because polynomial time approximation methods for TSP with triangle inequality do exist.

2.2.2 Approximation Methods for TSP with Triangle Inequality

Approximation Method within a factor of 2 We present a polynomial time algorithm that computes a tour within a factor of 2 of the optimal:

1. Compute the MST (minimum spanning tree) of the given graph G.

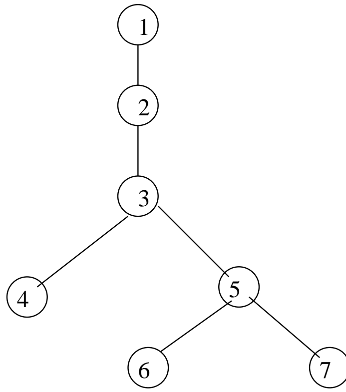


Figure 5: Example MST Solution

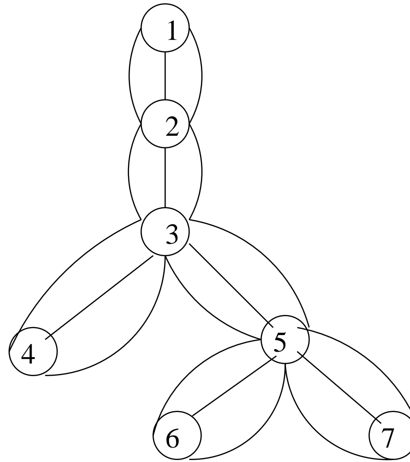


Figure 6: Pre-order Walk of MST

2. Walk on the MST:

   First do a pre-order walk of the MST: Transcribe this walk by writing out the nodes in the order they were visited in the pre-order walk. e.g., in the above tree the pre-order walk would be:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1;$$

3. Shortcut

   now compute a TSP tour from the walk using shortcuts for nodes that are visited several times; e.g., in the example above, the walk

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 5...$$

   will be transformed into

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5...,$$

   because in the subpath

$$...4 \rightarrow 3 \rightarrow 5...$$

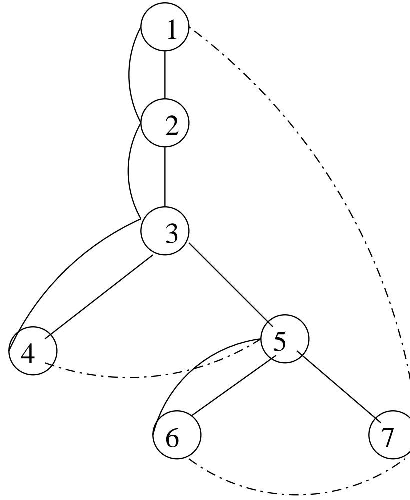


Figure 7: TSP tour with shortcuts

of the original walk, node 3 has already been visited.

Shortcutting is possible because we only consider complete graphs. Notice that after shortcutting the length of the walk cannot increase because of the triangle inequality. We can show this by induction: as a base case, if we shortcut a two-edge path by taking a single edge instead then the triangle inequality assures us that the shortcut cannot be longer than the original path. In the inductive case, consider the following figure in which we want to shortcut m edges:

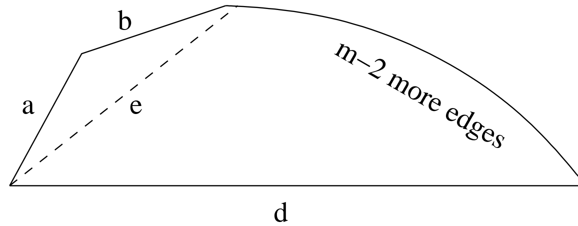


Figure 8: Triangle inequality induction

We can see that the length of edge e is less than or equal to the summed lengths of edges a and b, and the result of substituting e for a and b is a case in which d shortcuts a path of $m-1$ edges. By repeatedly constructing edges of type e, we can reach the base case because our inductive shortcutting procedure is only applied to paths of finite length.

Theorem 2.3. *The TSP tour constructed by the algorithm above is within a factor of* 2 *of the optimal TSP tour.*

Proof.

- The weight of the path constructed in step 2 above is no more than twice the sum of MST edge weights, since our walk of the tree traverses each edge of the MST at most twice. Also, as argued above, the shortcutting performed in step 3 cannot increase the summed weight.

Thus, $weight(TSP_{approx}) \leq 2 * weight(MST)$, where TSP_{approx} is the approximate TSP solution produced by our algorithm.

- If we delete one edge from the optimal TSP tour, we get a tree (all paths are trees) that connects all vertices in G. The sum of edge weights for this tree cannot be smaller than the sum of edge weights in the MST, by definition of MST. Thus $weight(MST) \leq weight(TSP_{OPT})$, where TSP_{OPT} is a minimum weight tour of the graph.

Thus, we've shown that $weight(TSP_{approx}) \leq 2 * weight(TSP_{OPT})$. ∎

Approximation Method Within a Factor of 1.5 A *Eulerian walk* on a graph is a walk that includes each edge of the graph exactly once. Our next algorithm depends on the following elementary theorem in graph theory: every vertex of a connected graph G has even degree iff G has a Eulerian walk.

As an aside, it's easy to see that a Eulerian walk can only exist if all nodes of a graph have even degree: Every time the walk passes through a node it must use two edges (one to enter the node and one to exit). No edges are traversed twice in the walk, so if a node is visited c times it must have degree $2c$, an even number.

Using the Eulerian walk theorem, we can get a factor 1.5 approximation. This approach is called the **Christofides algorithm**:

1. Find the MST of the given graph G.

2. Identify all odd-degree nodes in the MST

   Another elementary theorem in graph theory says that the number of odd-degree nodes in a graph is even. It's easy to see why this is the case: The sum of the degrees of all nodes in a graph is twice the number of edges in the graph, because each edge increases the degree of both its attached nodes by one. Thus, the sum of degrees of all nodes is even. For a sum of integers to be even it must have an even number of odd terms, so we have an even number of odd-degree nodes.

3. Do *minimum cost* perfect matching on the odd-degree nodes in the MST

   A *matching* is a subset of a graph's edges that do not share any nodes as endpoints. A *perfect matching* is a matching containing all the nodes in a graph (a graph may have many perfect matchings). A *minimum cost perfect matching* is a perfect matching for which the sum of edge weights is minimum. A minimum cost perfect matching of a graph can be found in polynomial time.

4. Add the matching edges to the MST.

   This may produce "doubled" edges, which are pairs of edges joining the same pair of nodes. We will allow doubled edges for now. Observe that in the graph produced at this step, all nodes are of even degree.

5. Do a Eulerian walk on the graph from the previous step.

   By the Eulerian walk theorem stated earlier, a Eulerian walk exists on this graph. Moreover, we claim without proof that it can be found in polynomial time.

6. Shortcut the Eulerian walk.

   Since the Eulerian walk traverses all nodes in the graph, we can shortcut this walk to produce a tour of the graph of total weight less than or equal to that of the Eularian walk. The proof of this is the same as the one used for shortcutting in the previous triangle inequality TSP approximation algorithm.

A graphical example of the Christofides algorithm is included at the end of this section.

Theorem 2.4. *The Christofides algorithm achieves 1.5 factor approximation.*

Proof. Clearly, the cost of the MST found in step 1 above is at most $weight(TSP_{OPT})$, for we can convert TSP_{OPT} into a spanning tree by eliminating any one edge. We now show that the process in steps 2-5 of converting this MST into a TSP solution adds weight no more than $.5 * weight(TSP_{OPT})$.

Assume we have a solution TSP_{OPT} to the TSP problem on G. Mark all odd-degree nodes found in step 2 of the approximation algorithm in TSP_{OPT}. This is possible because TSP_{OPT} contains all nodes in G. As has been shown above, there is now an even number of marked nodes.

Now let us build a cycle through only the marked nodes in TSP_{OPT}. The length of the resulting tour is still at most $weight(TSP_{OPT})$, since all we have done is shortcut a minimal tour and the triangle inequality holds.

Now consider matchings of the marked nodes. We can construct two perfect matchings from the cycle through the marked nodes, because the number of marked nodes is even. For example, if we have a cycle as in Figure 5

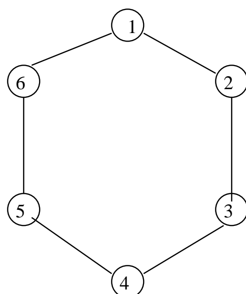


Figure 9: Cycle through 6 nodes

the two matchings will be as in Figure 6

Since by combining these two matchings one obtains the original marked cycle, the length of the lightest of these matchings is at most $.5 * weight(TSP_{OPT})$.

We have just shown that

$$\begin{aligned} mincost\ matching &\leq smaller\ matching \leq 0.5 * marked\ cycle \\ &\leq 0.5 * weight(TSP_{OPT}) \end{aligned}$$

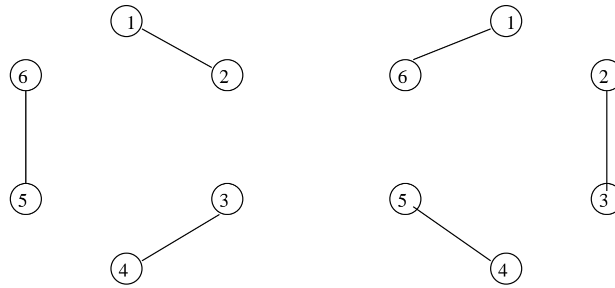


Figure 10: 2 Matchings from cycle

Thus, $.5 * weight(TSP_{OPT})$ is the maximum weight that we add to the weight of the MST in the algorithm to convert it to a tour of the graph. Therefore, the total weight of the solution found by the algorithm is at most $1.5 * weight(TSP_{OPT})$, with at most $weight(TSP_{OPT})$ for the MST and at most $.5 * weight(TSP_{OPT})$ to convert the MST to a tour. ▮

Notes There is a huge body of literature on TSP and its variants. One of the best sources for further reading is the book by Lawler et. al. [9].

The Christofides Algorithm - Example

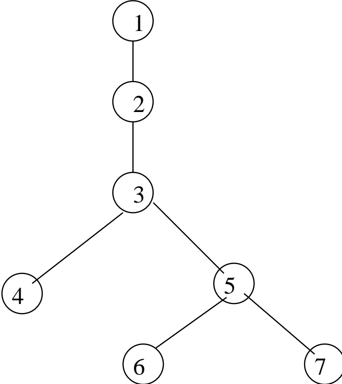


Figure 11: Step 1: MST Solution

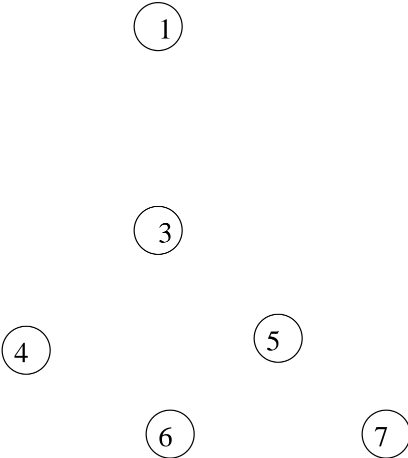


Figure 12: Step 2: Find odd-degree Nodes

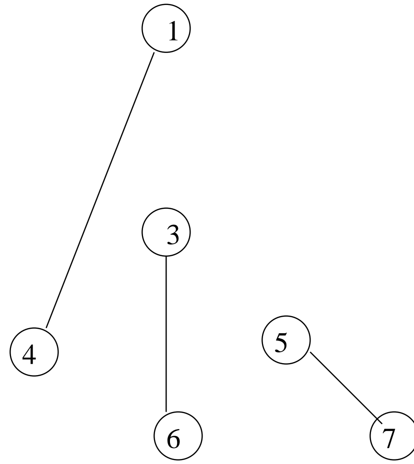


Figure 13: Step 3: Matching

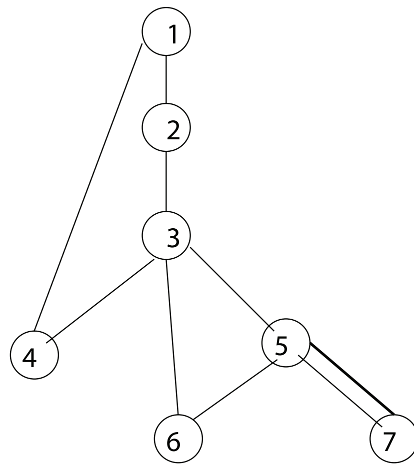


Figure 14: Step 4: Add matching edges

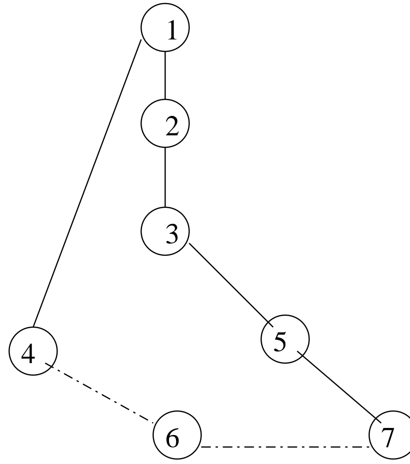


Figure 15: Step 5 and 6: Eulerian walk with Shortcuts

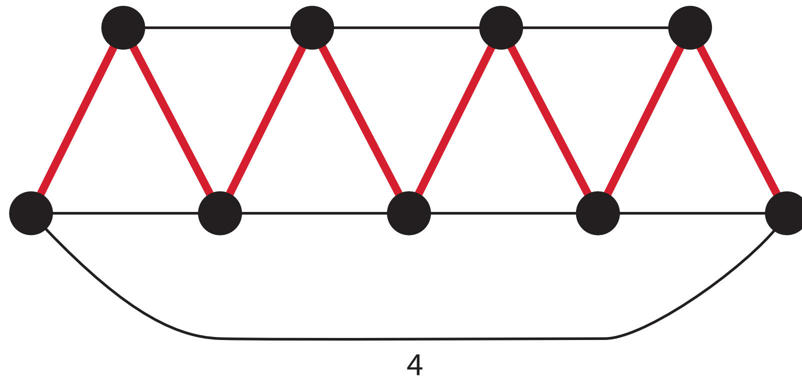


Figure 16: In the above graph all shown edges have weight 1, except the single edge with weight 4. The input to the TSP is a transitive closure of this graph, i.e. a complete graph with weights on edges defined by distances between the endpoints according to the graph shown in the figure. Red edges are the MST. It has only 2 odd-degree nodes. Connecting them gives TSP of weight 12. Optimum has weight 9. This can be generalized to give a ratio that is as close to 1.5 as desired.

3 Matchings, Edge Covers, Node Covers, and Independent Sets

A graph $G = (V, E)$ consists of the set of vertices (nodes) V and the set of edges E. $|S|$ denotes the cardinality (size) of a set S, and S^* denotes a set which is optimal in some sense to be defined. We consider only unweighted graphs here.

A *matching*, M, on a graph G is a subset of E which does not contain any edges with a node in common. A **maximum matching**, M^*, on a graph G is a matching on G with the highest possible cardinality. A **perfect matching** consists of edges which cover all the nodes of a graph. A perfect matching has cardinality $n/2$, where $n = |V|$.

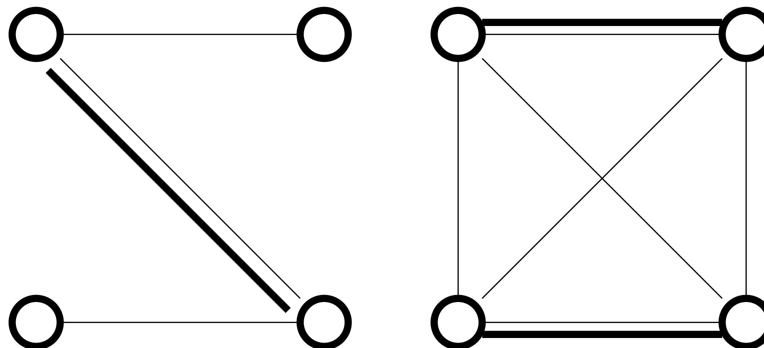


Figure 17: A matching and a maximum (and perfect) matching.

Matchings have applications in routing (matching between input and output ports) and job assignment (pairing workers to tasks). Besides direct applications, matching often arises as a subroutine in various combinatorial optimization algorithms (e.g. Christophedes TSP approximation).

An **edge cover**, ρ, of a graph G is a subset of E which contains edges covering all nodes of G. That is, for each $u \in V$ there is a $v \in V$ such that $(u, v) \in \rho$. We assume there are no isolated nodes, but if there are they can easily be handled separately, so this assumption does not have any cost in practice. A **minimum edge cover**, ρ^*, of a graph G is an edge cover of G with the smallest possible cardinality.

A **node cover**, S, of a graph G is a subset of V which contains nodes covering all edges of G. That is, for each $(u, v) \in E$, $u \in S$ or $v \in S$. A **minimum node cover**, S^*, of a graph G is a node cover of G with the smallest possible cardinality. An application of node cover is in placing equipment which tests connectivity on a network, where there must be a device on an end of each link.

An **independent set**, α, in a graph G is a subset of V which contains only unconnected nodes. That is, for each $v, w \in \alpha$, $(v, w) \notin E$. A **maximum independent set**, α^*, in G is an independent set in G with the highest possible cardinality. Independent set algorithms are useful as a subroutine in solving more complicated problems and in applications where edges represent conflicts or incompatibilities.

3.1 Minumum Edge Cover and Maximum Matching

For any graph G, the minimum edge cover size and the maximum matching size sum to the total number of nodes.

Theorem 3.1. *$|\rho^*| + |M^*| = n$.*

Proof. Consider some maximum matching, M^*. Let S be the set of nodes that are not covered by M^*, that is, those nodes which are not endpoints of any edge in M^*. S is an independent set. To see this, suppose $v, w \in S$ and $(v, w) \in E$. By our construction of S, neither v nor w are covered by M^*, so $M^* \bigcup \{(v, w)\}$ is a matching with cardinality that strictly larger than the cardinality of M^*. But this is a contradiction, since we started with a maximum matching. Thus, S must be an independent set, as claimed.

Now we can construct a valid edge cover, ρ, by taking the edges of M^* and adding an additional edge to cover each node in S. Clearly this accounts for all nodes, though it may contain more edges than needed to do so. But since ρ is *a valid* edge cover, we know it contains at least as many edges as the *minimum* edge cover, ρ^*. Thus,

$$|\rho^*| \leq |\rho| = |M^*| + |S| = |M^*| + (n - 2|M^*|) = n - |M^*|,$$

and therefore

$$|\rho^*| + |M^*| \leq n.$$

We proceed by symmetry to obtain the opposite bound. We now start by considering a minimum edge cover, ρ^*. A minimum edge cover cannot have chains of more than two edges, or some in the middle would be redundant. Therefore ρ^* induces a subgraph of G that consists of stars (trees of depth one), as in Figure 18.

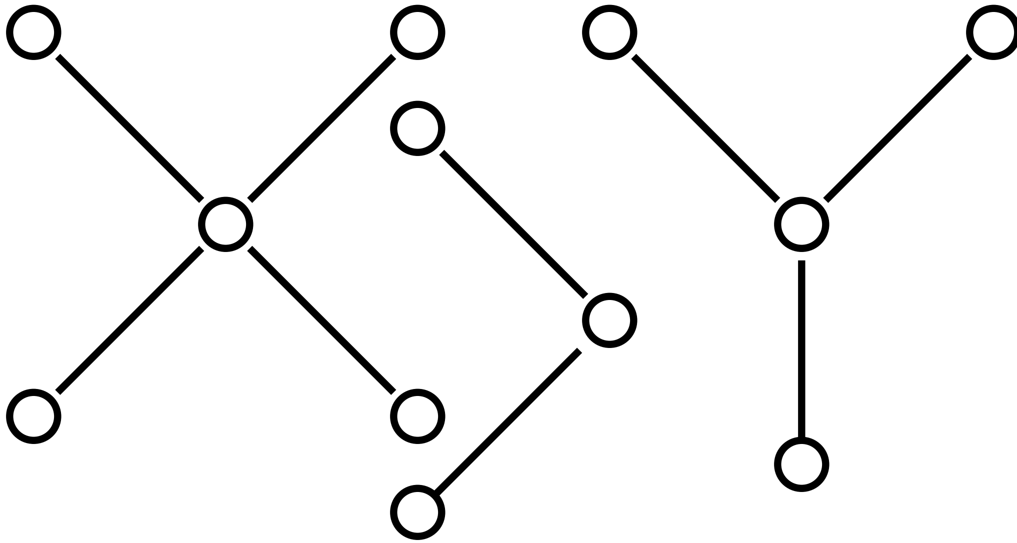


Figure 18: A minimum edge cover induces a star graph.

Since a star with k edges covers $k + 1$ nodes, we can conclude that the number of stars is $n - |\rho^*|$. We can construct a (not necessarily maximum) matching, M, by taking one edge from each star. By the same sort of reasoning used in the previous case, we see here that

$$|M^*| \geq |M| = n - |\rho^*|,$$

or,

$$|\rho^*| + |M^*| \geq n.$$

Combining the two inequalities gives us the equality we claim. ▮

3.2 Maximum Independent Set and Minimum Node Cover

For any graph G, the maximum independent set size and the minimum node cover size sum to the total number of nodes.

Theorem 3.2. $|\alpha^*| + |S^*| = n$.

Proof. Consider some minimum node cover, S^*. Let α be the set of vertices of G not included in the cover. Observe that α is an independent set since $(v, w) \in E \Rightarrow v \in S^*$ or $w \in S^*$. As in the previous proof, since the maximum independent set is at least as large as this independent set, we can conclude that $|\alpha^*| \geq |\alpha| = n - |S^*|$, and therefore

$$|\alpha^*| + |S^*| \geq n.$$

Next, consider some maximum independent set α^*. Let S be all nodes not in α^*. The set S constitutes a node cover since no edge has both endpoints in α^*, by definition of independence, and so each edge has at least one endpoint in the set of nodes not in α^*. This node cover is no smaller than the minimum node cover, so $|S^*| \leq |S| = n - |\alpha^*|$, and therefore

$$|\alpha^*| + |S^*| \leq n.$$

Again, combining the two inequalities gives us the equality we claim. ∎

3.3 Minimum Node Cover Approximation

Our goal is to find a minimum node cover for a graph G. Or, if we could find a maximum independent set, we could simply take the nodes not in it. But these problems are NP-hard, so we will develop an approximation algorithm for minimum node cover. In other words, instead of looking for S^*, we will try to construct a node cover, S, that is not much larger than S^*, and prove some bound on how close to optimal it is.

Consider this greedy approach:

1. Let $V' = \phi$.
2. Take the node v of highest degree, and let $V' = V' \bigcup \{v\}$.
3. Delete v from the graph (along with any edges (v, w)).
4. If there are no edges left, stop, else go back to step 2.

Clearly, the resulting set V' constitutes a node cover. We are going to prove later that this approach guarantees $\log n$ approximation, i.e., the node cover produced is no more than a factor of $\log n$ larger than the minimum one. Moreover, it is possible to show that there are cases where this approach indeed produces a cover $\log n$ times optimal.

We can do much better. Consider the following algorithm:

1. Compute maximum matching for G.

2. For each edge (u, v) in the maximum matching, add u and v to the node cover.

This is a node cover because any edge not in the maximum matching shares an endpoint with some edge in the maximum matching (or else this edge could have been added to create a larger matching). For an illustration, see Figure 19.

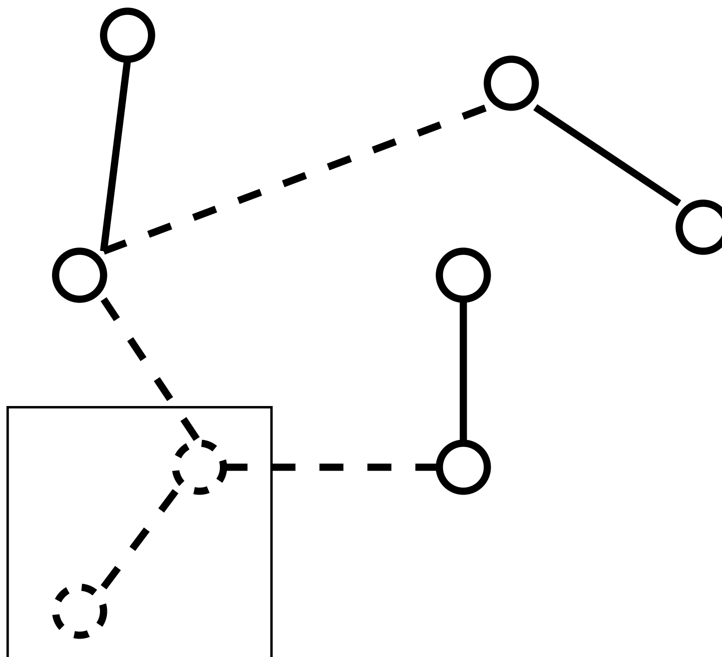


Figure 19: Converting a maximum matching into a node cover. Matching represented by solid lines. Two marked nodes are not covered by node cover, and thus can be added to the matching, proving that it was not maximum.

Theorem 3.3. *A node cover for a graph G produced by the above algorithm is no more than twice as large as a minimum node cover of G.*

Proof. Assume $|M^*| > |S^*|$. Then some $v \in S^*$ must touch two edges in M^*, but this is a contradiction, by construction of M^*. Thus, $|M^*| \leq |S^*|$. Therefore, our node cover (constructed in the above algorithm, which finds a node cover of size $2|M^*|$) is within a factor of 2 of the minimum node cover. ∎

Although one can find a maximum matching in polynomial time, the algorithm is far from trivial. We can greatly simplify the above algorithm by using *maximal* instead of *maximum* matching. A **maximal matching** is a matching that cannot be augmented without first deleting some edges from it. Here is an algorithm for computing a maximal matching:

1. Let $\tilde{M} = \phi$.
2. Pick an edge $(v, w) \in E$.
3. Add (v, w) to $\tilde{M}$, and delete nodes v and w and all edges adjacent to them from the graph.

4. If there are remaining edges, go back to step 2.

The set $\tilde{M}$ constitutes a matching because no two edges in the set share an endpoint: once (v, w) is added to $\tilde{M}$, v and w are deleted, so no other edges with these endpoints will be considered. It is maximal (meaning we can't add edges to it and still have a matching) because it continues until no remaining edges can be added. Observe that, by construction, every edge in the graph touches at least one of the nodes that are touched by the maximal matching. (If not, then this edge can be added to the matching, contradicting its maximality.) Thus, by taking both endpoints of the edges in the maximal matching we get a node cover. Moreover, we have $|\tilde{M}| \leq |M^*| \leq |S^*|$, and so this much simpler algorithm not only still comes within a factor of 2 of the minimum node cover size, but also never yields a worse approximation (in fact, usually better).

4 Intro to Linear Programming

4.1 Overview

LP problems come in different forms. We will start with the following form:

$$\begin{aligned} Ax &= b \\ x &\geq 0 \\ \min \ & cx \end{aligned}$$

Here, x and b are column vectors, and c is a row vector. A *feasible* solution of this problem is a vector x which satisfies the constraints represented by the equality $Ax = b$ and the inequality $x \geq 0$. An *optimal* solution is a feasible solution x that minimizes the value of cx.

It is important to note that not all LP problems have an optimal solution. Sometimes, an LP can have no solution, or the solution can be unbounded i.e. for all λ, there exists a feasible solution x_λ such that $cx_\lambda \leq \lambda$. In the first case we say the problem is over-constrained and in the second case that it is under-constrained.

4.2 Transformations

The LP that we would like to solve might be given to us in many different forms, with some of the constraints being equalities, some inequalities, some of the variables unrestricted, some restricted to be non-negative, etc. There are several elementary transformations that allow us to rewrite any LP formulation into an equivalent one.

Maximization vs minimization: Maximizing linear objective cx is equivalent to minimizing objective $-cx$.

Equality constraints to inequality constraints: Suppose we are given an LP with equality constraints and would like to transform it into an equivalent LP with only inequality constraints. This is done by replacing $Ax = b$ constraints by $Ax \leq b$ and $-Ax \leq -b$ constraints (doubling the number of constraints.)

Inequality constraints to equality constraints: To translate an inequality constraint $a_i x \leq b_i$ (where a_i is ith row of matrix A and b_i is ith coordinate of b) into an equality constraint, we can introduce a new slack variable s_i and replace $a_i x \leq b_i$ with

$$\begin{aligned} a_i x + s_i &= b_i \\ s_i &\geq 0 \end{aligned}$$

It is noted that as the number of slack variables introduced equals the number of the original inequalities, the cost of this transformation can be high (especially in cases where there exist many inequalities relative to the original number of variables).

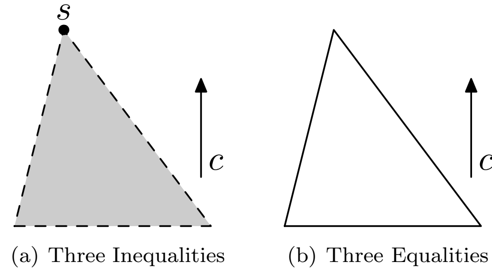


Figure 20: A simple LP with three constraints

The alternative approach where all inequalities are directly transformed to equalities, which is equivalent to obtaining a solution with all slack variables equal to zero is incorrect, since there can exist an LP for which none of its optimum solutions satisfies all of its inequality constraints as equalities.

Consider for example the simple LP of figure 20. While it is straightforward to detect an optimum solution if the three constraints are inequalities, there exists no feasible solution if the three inequalities become equalities, since such a feasible solution would have to lie on the intersection of all three lines.

From unrestricted to non-negative variables: Given an LP with no non-negativity constraints $x \geq 0$, we would like to translate it into an LP that has only non-negative variable assignments in its solutions. To do so, we will introduce two variables x_i^+ and x_i^- for every original variable x_i, and require

$$\begin{aligned} x_i &= x_i^+ - x_i^- \\ x_i^+ &\geq 0 \\ x_i^- &\geq 0 \end{aligned}$$

We observe that multiple solutions of the resulting LP may correspond to a single solution of the original LP. Moreover, this transformation doubles the number of variables in the LP.

Example: Using the above described transformations, we can transform

$$\begin{aligned} Ax &\leq b \\ \max \quad & cx \end{aligned}$$

into the form

$$\begin{aligned} A'x' &= b' \\ x' &\geq 0 \\ \min \quad & c'x' \end{aligned}$$

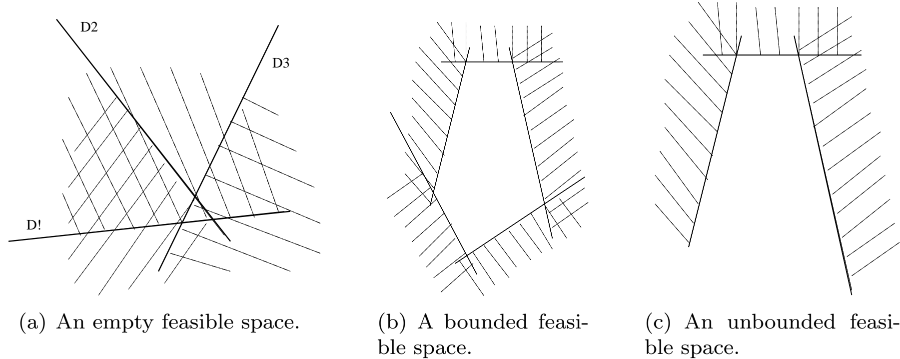
Assuming the dimension of A is $m \times n$, the result is as follows:

$$\begin{aligned} x' &= \begin{bmatrix} x^+ \\ x^- \\ s \end{bmatrix} \\ A' &= [A; -A; I] \\ b' &= b \\ c' &= [-c; c; 0^m] \end{aligned}$$

Where 0^m represents a vector of m zeros.

4.3 Geometric interpretation of LP

If we consider a LP problem with an inequality constraint $Ax \leq b$, the set of feasible solutions can be seen as the intersection of half planes. Each constraint cuts the space into two halves and states that one of these halves is outside the feasible region. This can lead to various kinds of feasible spaces, as shown in figure 21 for a two-dimensional problem.



(a) An empty feasible space. (b) A bounded feasible space. (c) An unbounded feasible space.

Figure 21: Feasible space for inequality constraints.

The feasible set is *convex*, ie. whenever the feasible space contains two points, it must contain all the segment joining these two points. This can be shown as follows: If x and y both belong to the feasible set, then $Ax \leq b$ and $Ay \leq b$ must hold. Any point z on the segment [xy] can be expressed as $\lambda x + (1 - \lambda)y$ for a given λ such that $0 \leq \lambda \leq 1$. Thus, $Az = \lambda Ax + (1 - \lambda)Ay \leq \lambda b + (1 - \lambda)b = b$. Hence, z is a feasible solution.

If we consider a LP problem with an equality constraint $Ax = b$, the feasible set is still a convex portion of space, of a lower dimension. This is the case for example with the space defined by the equality $x + y + z = 1$ in the 3-dimensional plane, as shown in figure 22, where the solution space is the 2-dimensional triangular surface depicted.

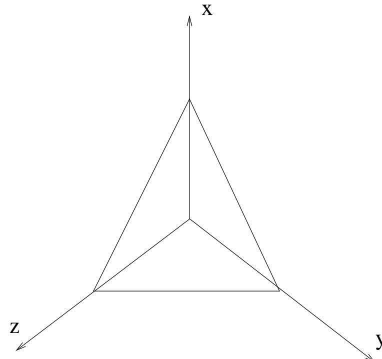


Figure 22: Feasible space for an equality constraint.

4.4 Existence of optimum at a vertex of the polytope

Our intuition is that an optimal solution can always be found at a vertex of the polytope of feasible solutions. In the 2-dimensional plane, the equation $cx = k$ defines a line that is orthogonal to the vector c. If we increase k, we get another line parallel to the first one. Intuitively, we can keep on increasing k until we reach a vertex. Then, we can no longer increase k, otherwise we are going out of the feasible set (see figure 23.)

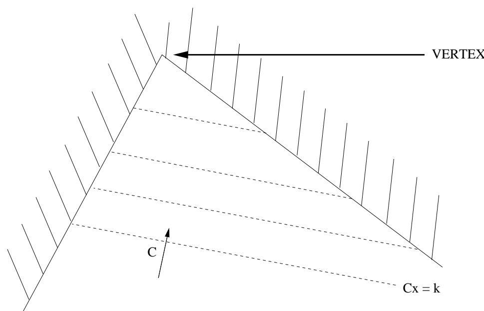


Figure 23: Location of the Optimum.

We can also observe that if we were given c that is orthogonal to an edge of the feasible set in the example, any point of that edge would have been an optimal solution. So our intuition is that, in general, optimal solutions are vertices but in certain circumstances, optimal solutions can also exist outside of vertices (e.g. at the faces of the polytope.) The observation is that, even if every optimal solution is not necessarily a vertex, *there exists* a vertex that is an optimal solution of the LP problem. Now we will try to prove this formally. First, we need a formal definition of a vertex.

Definition 4.1. *Let P be the set of feasible points defined by $P = \{x|Ax = b, x \geq 0\}$. A point x is a vertex of the polytope iff $\nexists y \neq 0$ such that $(x + y \in P) \wedge (x - y \in P)$.*

The following theorem implies that there is always an optimal solution at a vertex of the polytope.

Theorem 4.2. *Consider LP $\min\{cx|Ax = b, x \geq 0\}$ and the associated feasible region $P = \{x|Ax = b, x \geq 0\}$. If minimum exists, then given any point x that is not a vertex of P, there exists a vertex x' of P such that $cx' \leq cx$.*

Proof: The proof works by moving us around the polytope and showing that moving towards an optimal solution moves us towards a vertex. More formally, the proof constructs a sequence of points beginning with x, such that the value of the objective function is non-increasing along this sequence and the last point in this sequence is a vertex of the polytope.

Consider a point x in the polytope such that x is not a vertex. This implies that $\exists y \neq 0$ such that $(x + y \in P) \wedge (x - y \in P)$. We will travel in the polytope along this y. By the definition of P, $A(x + y) = b = A(x - y)$. Thus, $Ay = 0$. Also, assume $cy \leq 0$. Otherwise we can replace y by $-y$.

We will try to travel as far as possible in the direction of y. To see that this does not increase the value of the objective function, consider the points $x + \lambda y, \lambda \geq 0$. Substituting, we get:

$$c(x + \lambda y) = cx + \lambda cy \leq cx.$$

To check that the new point is still feasible, we need to verify first the equality constraint:

$$A(x + \lambda y) = Ax + \lambda Ay = Ax = b$$

So any point of the form $x + \lambda y$ satisfies the first constraint. However, we must be more careful when choosing a λ so that we do not violate the non-negativity of x. Let x_j denote the jth coordinate of x and y_j the jth coordinate of y. Notice that $\forall i$, $x_i = 0$ implies that $x_i + y_i = y_i \geq 0$ and $x_i - y_i = -y_i \geq 0$. Hence $x_i = 0 \Rightarrow y_i = 0$.

There are two cases to consider:

1. $\exists y_j : y_j < 0$

   Let S be the range of λ such that $\forall i, x_i + \lambda y_i \geq 0$. S is bounded because $\exists j$ such that for $\lambda > \left|\frac{x_j}{y_j}\right|$ we get $x_j + \lambda y_j < 0$ and thus this $\lambda \notin S$.

   Let λ' be the largest element of S. Then $x + \lambda' y$ can be interpreted as the first point where we reach a border of the feasible set. Clearly, $\lambda' \geq 1 > 0$ since $x + y$ is feasible.

   Also, necessarily $\exists k : x_k + \lambda' y_k = 0$ but $x_k \neq 0$. Intuitively, k corresponds to the direction in which we moved until we reached a border.

   Finally, notice $\forall i$, $x_i = 0 \Rightarrow y_i = 0 \Rightarrow x_i + \lambda' y_i = 0$.

   Hence the new point $x + \lambda' y$ must have at least one more zero coordinate than x and is still inside the feasible set.

2. $\forall j : y_j \geq 0$.

   This implies that $\forall \lambda \geq 0 : x + \lambda y \geq 0$. We don't have to worry about non-negativity being violated. We can move in y's direction as far as we want. There are two further cases.

   (a) $cy < 0$. In this case, we can make $c(x + \lambda y)$ arbitrarily small by increasing λ. This contradicts the assumption that LP had a bounded solution.

   (b) $cy = 0$. The set of solutions $x + \lambda y$ have equal objective function value. Since $y \neq 0$, one of the coordinates of y is positive. Thus, we can negate y without changing the value of $c(x + \lambda y)$ and we enter the first case.

This step produces a new point x' such that $cx' \leq cx$ and x' has at least one more zero coordinate than x. Once we have found the new point, we start the process all over again, choosing a new y. If we cannot find a new y, then we are at a vertex. We cannot loop indefinitely since each successive point has at least one more zero coordinate than the previous point, and there are a finite number of coordinates. Hence, we terminate at a vertex x'' such that $cx'' \leq cx$. ∎

The above theorem implies that at least one optimum solution lies on a vertex. One might suggest that a solution to a given LP can be found simply by examining which of its vertices minimizes the given objective. Such an approach cannot be efficient though, since the number of vertices may be exponential in the number of the constraints.

4.5 Primal and Dual

Consider the following LP:

$$\begin{array}{rcl} Ax & \leq & b \\ x & \geq & 0 \\ \text{maximize} & cx & \end{array}$$

We will call it ”primal LP”. The ”dual” LP is composed by the following syntactic transformation. [1] The dual LP is described by a new vector y of variables and a new set of constraints. Each new variable corresponds to a distinct constraint in the primal, and each new constraint corresponds to a distinct variable in the primal.

$$\begin{array}{rcl} y^tA & \geq & c \\ y & \geq & 0 \\ \text{minimize } y^tb & & \end{array}$$

Note that the variables and constraints in the dual LP may not have any useful or intuitive meaning, though they often do.

Calling the maximization problem the ”primal” LP turns out to be rather arbitrary. In practice, the LP formulation of the original problem is called the primal LP regardless of whether it maximizes or minimizes. We follow this practice for the rest of this presentation.

We say that x is *feasible* if it satisfies the constraints of the LP. In our case, this means that $Ax \leq b$ and $x \geq 0$. Feasible y can be defined in a similar way. Given any feasible x and y to the primal and dual LP's respectively, we can easily derive following inequality.

$$cx \leq (y^tA)x = y^t(Ax) \leq y^tb$$

The inequality $cx \leq y^tb$ is called *weak duality theorem.* Weak duality is useful in order to prove “quality” of a particular feasible solution: Given any maximization LP, if we can somehow compute a feasible solution to the dual LP, we can establish an upper bound on the primal LP solution. Similarly, given any minimization LP, if we can somehow compute a solution to the dual LP, we can establish a lower bound on the primal LP.

The weak duality inequality applies to any feasible primal and dual solutions x, y. If we consider optimum primal and dual solutions x^* and y^*, one can prove the *strong duality theorem* that says

$$cx^* = (y^*)^tb$$

Combining these two theorems, we have that

$$cx \leq cx^* = (y^*)^tb \leq y^tb$$

[1] It is important to note that the following dual corresponds to the primal LP form above. Changing to a different form, e.g. minimization instead of maximization, results in a different dual.

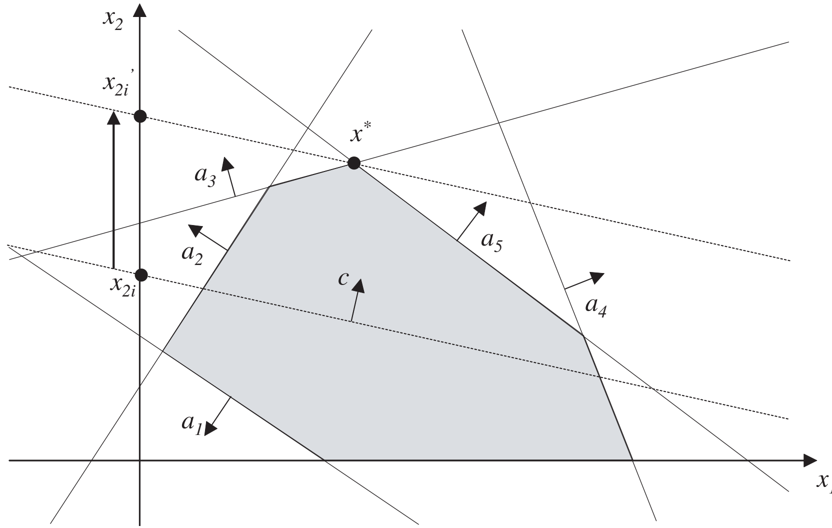


Figure 24: An example showing that how the polygonal structure can be obtained from a series of inequalities

With this property, for any feasible x and y, the optimum solution cx^* and $(y^*)^t b$ lie between cx and $y^t b$. This observation is very useful for many approximation algorithms. We will come back to strong duality later in the course. The approximation algorithms presented below use weak duality claim only.

Note that the above discussion is incomplete. In particular, it does not address cases where primal or dual are infeasible or where the optimum value is infinite.

4.6 Geometric view of Linear Programming duality in two dimensions

To better understand LP duality, we will consider two-dimensional case. In two-dimensional case LP can be re-written as

$$\begin{gathered} x = (x_1, x_2),\ x_1 \geq 0,\ x_2 \geq 0 \\ a_{11}x_1 + a_{12}x_2 \leq b_1 \\ a_{21}x_1 + a_{22}x_2 \leq b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 \leq b_n \\ \text{maximize } c_1x_1 + c_2x_2 \end{gathered}$$

Here, A is an $n \times 2$ matrix and b is an n-tuple.

For each inequality we can draw a line in an x_1 - x_2 coordinate system, and finally we will get some convex polygonal structure like fig. (24). Suppose we pick up arbitrary intercept $(0,\ x_{2i})$ and draw a line which is orthogonal to the vector $c = (c_1, c_2)$ going through the point. If the line meets the polygon, $x = (x_1, x_2)$ is feasible where x_1 and x_2 are on the line and inside the polygon. Then, we can increase x_{2i} and move the line up while it meets the polygon. When x_{2i} is maximized, $c_1x_1 + c_2x_2$ is also maximized. In this case the line will hit one vertex (or an edge when the edge is parallel to the line).

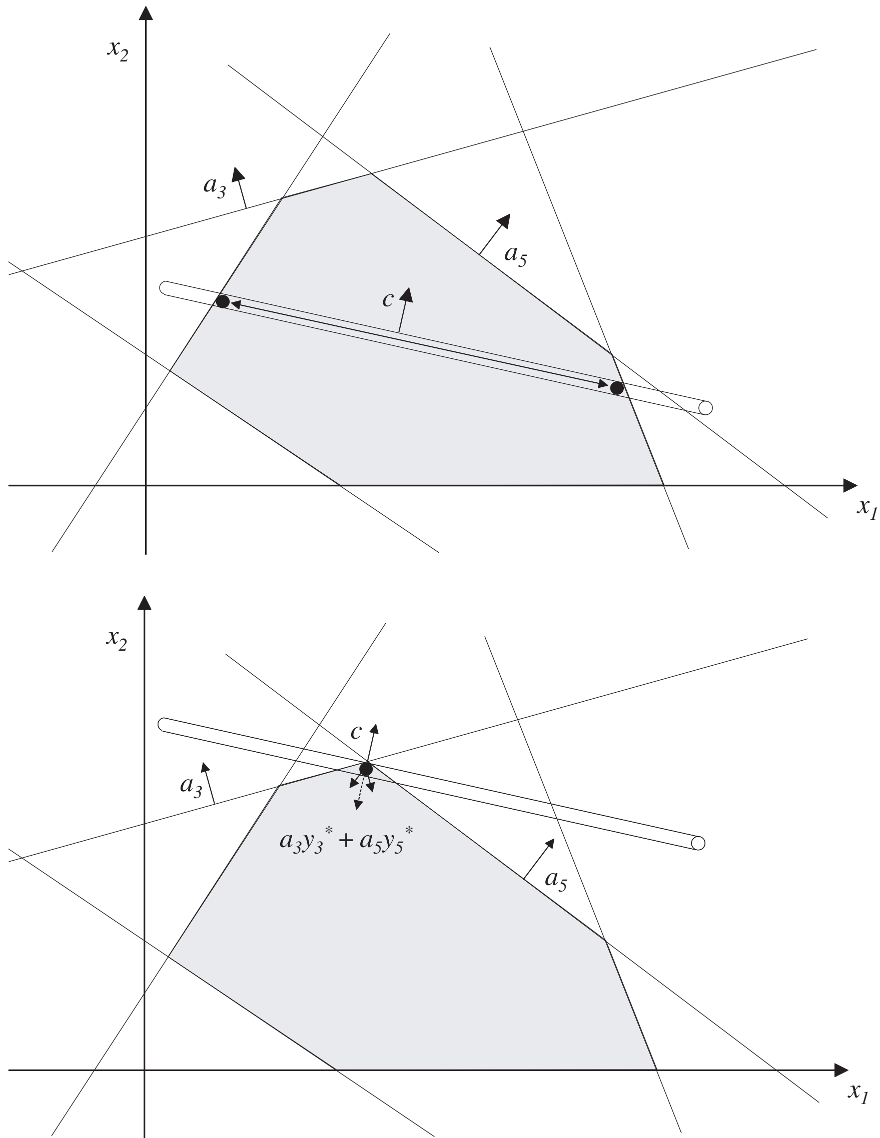


Figure 25: Pipe and Ball example

Now, suppose that there is a pipe which is parallel with the line inside the polygon, and a ball is moving inside the area which is restricted by the polygon and the pipe as in fig. (25). If we move the pipe up, at a certain point the ball is stuck in the corner and cannot move any more. At this point, cx is maximized. Suppose that the corner is made by lines orthogonal to $a_3 = (a_{31}, a_{32})$ and $a_5 = (a_{51}, a_{52})$. The ball is subject to three forces which are from the pipe and two edge of the polygon. These forces achieve an equilibrium, so the sum of three forces upon the ball is zero.

$$\begin{aligned} c - a_3 y_3^* - a_5 y_5^* &= 0 \\ c &= a_3 y_3^* + a_5 y_5^* \end{aligned}$$

Let $y^* be$ the vector with all 0 except ${y_3}^*$, ${y_5}^*$. Observe that y^* defined this way is indeed a feasible dual solution. Moreover, we have $c = (y^*)^t A$, and $y^* \geq 0$.

Using this observation, we will prove that y^* is not only feasible but optimum.

$$\begin{aligned} cx^* &= ((y^*)^t A)x^* \\ &= (y^*)^t (Ax^*) \\ &= y_3^*(a_3 x^*) + y_5^*(a_5 x^*) \\ &= y_3^* b_3 + y_5^* b_5 \\ &= (y^*)^t b \end{aligned}$$

Weak duality claims that $cx^* \leq y^t b$ for optimum primal x^* and for any feasible y. Since we proved that $cx^* = (y^*)^t b$ and our y^* is feasible, we have proved that our $y*$ indeed minimizes $y^t b$. This argument can be viewed as an informal proof of the strong duality theorem in the 2-dimensional case.

4.7 Historical comments

Linear Programming (or LP) was first introduced in the 1940's for military applications. The term *programming* here is different from the interpretation that we normally use today. It refers to a conceptually principled approach, as opposed to software development. The initial algorithm, called Simplex, was shown require exponential time in the worst case.

In the 1970's, the first provably polynomial time algorithm for solving linear programs was invented: it was called the ellipsoid algorithm. But this algorithm actually had huge constant factors and was much slower than existing implementations of Simplex. This caused bad publicity for polynomial time algorithms in general.

In the 1980's a better polynomial time algorithm called the Interior Point Method was developed. Nowadays, commercial packages (e.g. CPLEX) implement both Simplex and interior point methods and leave the decision of which method to use to the user. The quality of these packages is such, that they are even able to detect solutions to Integer Linear Programs (ILPs) within acceptable time margins, even though they do behave badly under some conditions.

Simplex algorithm is illustrated in Figure 26. Roughly speaking, the algorithm starts by transforming the problem into an equivalent one with an obvious (non-optimum) vertex, and iteratively moves from

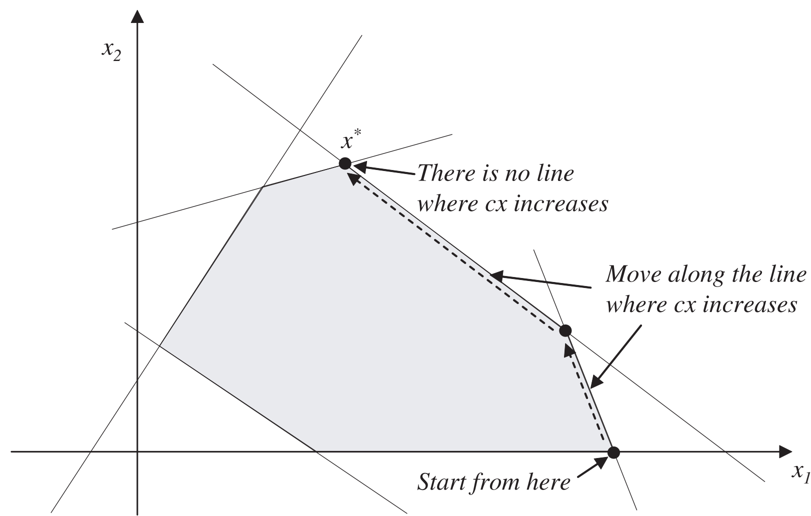


Figure 26: Simplex method example

one vertex to another, always trying to improve the value of the objective. It is important to note that the above description is quite incomplete. For example, sometimes it is not possible to improve objective value at every step and the algorithm has to be able to deal with this.

5 Approximating Weighted Node Cover

5.1 Overview and definitions

Given any graph $G = (V, E)$ with non-negative weights $w_i,\ 1 \leq i \leq n$ associated with each node, a node cover is defined as a set of nodes $S \subseteq V$ such that for every edge $(ij) \in V$, either $i \in S$ or $j \in S$, or both of them are in S (i.e. $\{i, j\} \cap S \neq \emptyset$). The minimum cardinality node cover problem is to find a cover that consists of the minimum possible number of nodes. We presented a 2-approximation algorithm for this problem.

In the *weighted node cover*, each node is associated with a weight (cost) and the goal is to compute a cover of minimum possible weight. The weight of a node cover is the sum of weights of all the chosen nodes.

Since min-cardinality node cover is a special case, weighted node cover is NP-hard as well. We will present two approaches to this problem, both based on the ideas of Linear Programming (LP).

5.2 Min Weight Node Cover as an Integer Program

Our first step is to write the Minimum Node Cover problem as an Integer Program (IP), that is a linear program where some of the variables are restricted to integral values. In our IP, there is a variable x_i corresponding to each node i. $x_i = 1$ if the vertex i is chosen in the node cover and $x_i = 0$ otherwise. The formulation is as follows:

$$\begin{aligned} \forall (ij) \in E : x_i + x_j &\geq 1 \\ \forall i \in V : x_i &\in \{0, 1\} \\ \text{minimize} \sum_{i \in V} x_i w_i & \end{aligned}$$

It is easy to see that there is a one-to-one correspondence between node covers and feasible solutions to the above problem. Thus, the optimum (minimum) solution corresponds to a minimum node cover. Unfortunately, since node cover is an NP hard problem, we cannot hope to find a polynomial time algorithm to the above IP.

5.3 Relaxing the Linear Program

General LPs with no integrality constraints are solvable in polynomial time. Our first approximation will come from relaxing the original IP, allowing the x_i to take on non-integer values[2]:

[2] There is no need to require $x_i \leq 1$; an optimal solution will have this property anyway. A quick proof: Consider an optimal solution x' with some $x'_i > 1$. Construct another set x'', identical to x' except with $x''_i = 1$. Clearly x'' still satisfies the problem constraints, but we have reduced a supposedly optimal answer. Thus, no optimal solution will have any $x_i > 1$.

$$\begin{aligned} \forall (ij) \in E : x_i + x_j &\geq 1 \\ \forall i \in V : x_i &\geq 0 \\ \text{minimize} \sum_{i \in V} x_i w_i & \end{aligned}$$

Suppose we have a solution x^* to the relaxed LP. Some x_i^* may have non-integer values, so we must do some adaptation before we can use x^* as a solution to the Minimum Weighted Node Cover problem.

The immediate guess is to use *rounding*. That is, we would create a solution x' by rounding all values x_i^* to the nearest integer: $\forall i \in V : x'_i = min(\lfloor 2x_i^* \rfloor, 1)$.

We should prove first that this is a solution to the problem and second that it is a good solution to the problem.

Feasibility Proof The optimum fractional solution x^* is feasible with respect to the constraints of the relaxed LP. We constructed x' from x^* by rounding all values x_i^* to the nearest integer. So we must show that rounding these values did not cause the two feasibility constraints to be violated.

In the relaxed LP solution x^*, we have $\forall (ij) \in V.x_i^* + x_j^* \geq 1$. In order for this to be true, at least one of x_i^* or x_j^* must be $\geq 1/2$ and must therefore round up to 1. Thus, in our adapted solution x', at least one of x'_i or x'_j must be 1, so $\forall (ij) \in V.x'_i + x'_j \geq 1$. Thus, x' satisfies the first constraint.

Clearly, since $\forall i \in V.x_i^* \geq 0$, and x'_i is obtained by rounding x_i^* to the nearest integer, $\forall i \in V.x'_i \geq 0$. x' satisfies the second constraint also and is thus feasible.

Relationship to the Optimum Solution We proved that x' is feasible, but how good is it? We can prove that it is within a factor of two of the optimum.

Consider OPT_{int}, the value of the optimum solution to the Minimum Weighted Node Cover problem (and to the associated IP), and OPT_{frac}, the value of the optimum solution to our relaxed LP. Since any feasible solution to the IP is feasible for the relaxed LP as well, we have $OPT_{frac} \leq OPT_{int}$.

Now, the weight of our solution x' is:

$$\sum_{i=1}^{n} x'_i w_i = \sum_{i=1}^{n} min(\lfloor 2x_i^* \rfloor, 1) w_i \leq \sum_{i=1}^{n} 2x_i^* w_i = 2OPT_{frac} \leq 2OPT_{int}$$

Thus, we have a approximation algorithm within a factor of 2.

5.4 Primal/Dual Approach

The above approach is simple, but the disadvantage is that it requires us to (optimally) solve a linear program. Although, theoretically, solving LPs can be done in polynomial time, it takes quite a lot of time in practice. Thus, it is advantageous to try to develop an algorithm that is faster and does not require us to solve an LP.

The variables x_i in the primal correspond to nodes, so the dual will have variables l_{ij} corresponding to the edges. The primal is a minimization problem, so the dual will be a maximization problem. The dual formulation is as follows:

$$\begin{aligned} \forall i \in V : \sum_{j:(ij)\in E} l_{ij} &\leq w_i \\ \forall (ij) \in E : l_{ij} &\geq 0 \\ \text{maximize} \sum_{(ij)\in E} l_{ij} & \end{aligned}$$

Let's look at the primal and dual LP we have compared to the "standard" form. A is an $n \times m$ matrix where n is the number of nodes and m is the number of edges, and the rows and columns of A correspond to nodes and edges respectively. b is the vector of weights, that is $(w_1, w_2, \ldots, w_n)^t$, and c is $(1, 1, \ldots, 1)$. We cannot say what l_{ij} is exactly because it's the result of syntactic transformation. However, we can guess that it is a certain value for each edge.

We will now construct an algorithm that tries to find both primal and dual approximate solutions *at the same time*. The intuition is as follows: We start from a feasible dual solution and infeasible primal. The algorithm iteratively updates the dual (keeping it feasible) and the primal (trying to make it "more feasible"), while always keeping a small "distance" between the current values of the dual and the primal. In the end, we get a feasible dual and a feasible primal that are close to each other in value. We conclude that the primal that we get at the end of the algorithm is close to optimum.

1. Initially set all l_{ij} to 0 and $S = \emptyset$ (or $\forall i : x_i = 0$). Unfreeze all edges.

2. Uniformly increment all unfrozen l_{ij} until for some i we hit the dual constraint $\sum_{j:(ij)\in E} l_{ij} \leq w_i$.[3] We call node i saturated.

3. Freeze edges adjacent to the newly saturated node i.

4. $S = S \cup \{i\}$ $(x_i = 1)$

5. While there are still unfrozen edges, go back to step 2

6. Output S.

Analysis First we claim that the algorithm computes a feasible primal, i.e. the set S output by the algorithm is indeed a node cover. This is due to the fact that, by construction, the algorithm continues until all edges are frozen. But an edge is frozen only when at least one of its endpoints is added to S. The claim follows.

Now we claim that this algorithm achieves a factor 2 approximation. To see this, observe that, by construction:

$$\sum_{i\in S} w_i = \sum_{i\in S} \sum_{j:(ij)\in E} l_{ij}$$

[3] In the "real life" implementation of this algorithm, we can merely figure out which l_{ij} is closest to its respective w_i and choose it immediately in step 2, instead of going through a series of small increments.

For each edge $(ij) \in E$, the term l_{ij} can occur at most twice in the expression on the right side, once for i and once for j. So,

$$\sum_{i \in S} \sum_{j:(ij) \in E} l_{ij} \leq 2 \sum_{(ij) \in E} l_{ij}$$

According to the weak duality theorem, we know that any feasible solution to the dual (packing) LP here will be less than the optimum solution of the primal (covering) LP. That is,

$$\sum_{(ij) \in E} l_{ij} \leq OPT_{frac}$$

where OPT_{frac} is value of the optimum (fractional) solution to the LP. Let OPT_{int} be the value of the optimum (integral) solution to the IP. Because $OPT_{frac} \leq OPT_{int}$, we say

$$\sum_{i \in S} w_i = \sum_{i \in S} \sum_{j:(ij) \in E} l_{ij} \leq 2 \sum_{(ij) \in E} l_{ij} \leq 2OPT_{frac} \leq 2OPT_{int}$$

Thus, this algorithm produces a factor 2 approximation for the minimum node cover problem.

5.5 Summary

These notes describe a method that is commonly used to solve LP problems. In general, the strategy involves:

- Start with a feasible solution to the dual.
- Tweak the dual to get it closer to the optimum solution, while making related changes to the primal.
- Stop when the primal becomes feasible.

Notes Further reading: Chapter 3 in [7].

6 Approximating Set Cover

6.1 Solving Minimum Set Cover

Set cover is an extremely useful problem of core importance to the study of approximation algorithms. The study of this problem led to the development of fundamental design techniques for the entire field, and due to its generality, it is applicable to a wide variety of problems. In much the same manner as the approximation technique used in the Minimum Node Cover problem, we will look at the primal and dual LP formulation of the minimum set cover problem and derive an approximation algorithm from there. Using the weak duality theorem, we will prove that our approximation is within a factor of $\ln(n)$, where n is the cardinality of the set we are covering. Somewhat surprisingly, this is asymptotically the best algorithm that we can get for this problem.

Motivation To motivate this problem, consider a city with several locations where hospitals could be built. We want any given inhabitant to be within, say, ten minutes of a hospital, so that we can deal with any emergency effectively. For each location where we could put a hospital, we can consider all inhabitants within ten minutes' range of this location. The problem would then be to choose locations at which to build hospitals minimizing the total number of hospitals built subject to the constraint that every inhabitant is within ten minutes of (covered by) at least one hospital. Actually, this example is an example of a more specific sub-problem of general set cover, for which there exist better algorithms than in the general case, because it possesses a metric (the distances between customers and hospitals satisfy the triangle inequality). Nevertheless, it provides some motivation for a solution to this problem.

The Minimum Set Cover problem Let $S_1, S_2, \ldots S_m$ be subsets of $V = \{1, \ldots, n\}$. The set cover problem is the following: Choose the minimum number of subsets such that they cover V. More formally, we are required to choose $I \subseteq \{1, \ldots, m\}$ such that $|I|$ is minimum and

$$\bigcup_{j \in I} S_j = V.$$

In the more general problem, each subset can be associated with some weight, and we may seek to minimize the total weight across I instead of the cardinality of I. We can also view this as an edge cover problem in a hypergraph, where each hyperedge can connect any number of nodes (a hyperedge just corresponds to some $S_i \subseteq V$).

Solving Minimum Set Cover as a Linear Program As before, we begin by trying to write an LP. Let x_j, a $0-1$ variable, indicate whether the jth subset is chosen into the candidate solution (1 corresponds to chosen). Throughout the rest of this section, we will use the index i to denote elements and the index j to denote sets. The LP includes the above constraints on x_j and the following:

$$\forall i \in \{1, \ldots, n\} \sum_{j: i \in S_j} x_j \geq 1$$

$$\text{minimize} \sum_{j=1}^{m} x_j$$

The above constraint states: For every element i in our set, we want to sum the indicator variables x_j corresponding to the subsets S_j that contain i. Constraining this sum to be greater than or equal to one guarantees that at least one of the "chosen" subsets contains this element i.

Observe that the vertex cover problem is a special case of the set cover problem. There, the sets correspond to vertices and the elements to edges. The set corresponding to a vertex contains the edges incident on that vertex. Every edge is incident on two vertices. In the corresponding set cover instance, every element is contained in two sets.

We could try to solve the set cover problem in the same way as we solved the vertex cover problem - begin by relaxing the integral constraint in order to obtain a general LP, then utilize a similar rounding procedure. This time, however, there may be multiple sets containing any element i. Specifically, what if S_1, S_2, and S_3 are the only sets that contain the element 17, then one of our inequalities is $x_1+x_2+x_3 \geq 1$. What if our solver says that $x_1 = x_2 = x_3 = 1/3$?

We can no longer apply the same rounding procedure used in the vertex cover problem since our solution may not be a feasible one afterwards. This is shown by taking the above example: using the previous rounding procedure, we would end up setting $x_1 = x_2 = x_3 = 0$ causing the constraint, $x_1 + x_2 + x_3 \geq 1$, to be violated. We can however, use a $1/k$ rounding scheme, where k is the maximum number of sets that any node appears in - that is, multiply all variables by k and truncate the fractional part. This is good if k is very small (e.g., if $k = 2$, this is just the vertex cover problem). Unfortunately, in general k could be an appreciable fraction of m, in which case this results in a factor m approximation, which is trivial to obtain by simply picking all m sets.

A Greedy (and Better) Algorithm As before, we take the LP and formulate the dual LP. We now have a dual variable corresponding to each constraint in the primal. Let y_i be a variable corresponding to the constraint for element $i \in V$.

$$\begin{aligned} \forall j \in \{1, \ldots, m\} : \sum_{i \in S_j} y_i &\leq 1 \\ \forall i \in \{1, \ldots, n\} y_i &\geq 0 \\ \text{maximize} \sum_i y_i & \end{aligned}$$

Consider the following greedy algorithm:

1. Let $V' = V$, and $I = \emptyset$

2. Find S_j with the largest $|S_j \cap V'|$

3. Set $I = I \cup \{j\}$ and $V' = V' - S_j$

4. While $V' \neq \emptyset$ go back to step 2

5. Output I.

We will now construct a lower bound for the optimal as we proceed through the algorithm. We will prove the following lemma:

Lemma Let I be the set produced by the greedy algorithm. Then $\exists y \geq 0$ such that

$$\sum_{i=1}^{n} y_i = |I| \qquad \text{and} \qquad \forall j \in \{1, \ldots, m\} \sum_{i \in S_j} y_i \leq H(|S_j|),$$

We use $H(x) = 1 + \frac{1}{2} + \ldots + \frac{1}{x}$ to denote the *harmonic function.* Since $|S_j| \leq n$, $H(|S_j|) \leq H(n) = \Theta(\log n)$.

First assume that the lemma is true (we will prove it later). Observer that by dividing each variable y_i by $H(n)$, we get a feasible solution to the dual LP, since

$$\forall j \in \{1, \ldots, m\} \sum_{i \in S_j} \frac{y_i}{H(n)} \leq 1,$$

Using the weak duality theorem, we get

$$\begin{aligned} OPT_{integer} &\geq OPT_{fractional} \\ &\geq \text{feasible dual value} \\ &= \frac{\sum y_i}{H(n)} \\ &= \frac{|I|}{H(n)} \\ &= (\text{our greedy solution value})/H(n) \end{aligned}$$

Thus, the lemma implies that we have a $O(\log n)$ approximation for our problem.

Proof of the Lemma In the algorithm given above, let's consider the step that chooses S_j. Assign:

$$\forall i \in S_j \cap V' : y_i = \frac{1}{|S_j \cap V'|}$$

where V' corresponds to its value during that iteration (since V' changes with every iteration). We will show that this assignment of y values satisfies the conditions of the lemma.

Choose a set and look at the elements that are not yet covered. Due to the way we assign the values we end up satisfying the first property above. In each iteration, since to each node we are covering we assign weight equal to the inverse of the number of additional elements we are covering, the total increase in $\sum y_i$ is 1; the increase in $|I|$ is also 1. That is, we are increasing by one; we divide this 1 up equally amongst all new elements that we are covering. The main difficulty is to show that we can satisfy the second property as well.

Consider some set S after we have already chosen one other set. Some of the elements belonging to S might have been part of the first subset chosen, but these have already been covered. Denote the set of elements of S covered by the first chosen subset by Q_1. Similarly, we have a set Q_2 and so on. Define Q_0 as the empty set, since no elements have been chosen before the algorithm starts.

Observe that $S = Q_1 \cup Q_2 \cup Q_3 \cup \ldots \cup Q_k$, where k is the number of the iteration at which all elements of S were first covered (i.e., when the algorithm terminates).

Denote the set that was chosen by the algorithm at iteration j as S_j and let V^j be the set of elements that were not yet covered when S_j was chosen (i.e., the value of V' at the beginning of the iteration). This set covers $Q_j = S \cap S_j \cap V^j$ elements in S. Following elements of S are not yet covered at the beginning of this iteration:

$$S \cap V^j = Q_j \cup Q_{j+1} \cup \ldots \cup Q_k \tag{1}$$

Thus, there are $|S| - \sum_{i=1}^{j-1} |Q_i|$ such elements.

Now, here is the crucial observation which is the key to this analysis: the greedy algorithm chose S_j instead of S at this iteration because S_j is locally optimal, which means that it covers at least as many elements not covered by the sets chosen thus far by the greedy algorithm as S does; i.e. at the beginning of the iteration we have:

$$|S_j \cap V^j| \geq |S \cap V^j|. \tag{2}$$

During this iteration we increment some of the coordinates of the y vector corresponding to the elements of S that are covered during this iteration (Q_j). Each increment is by one over the total number of elements covered during this iteration, which is $|S_j \cap V^j|$. Thus, equation 2 implies that the *total* increase in $\sum y_i, i \in S$ can be bounded by:

$$\frac{|Q_{current}|}{|S_j \cap V^j|} \leq \frac{|Q_{current}|}{|S \cap V^j|}$$

Summing up all the increments and using equation 1, we can see that, at the end of the algorithm, we get:

$$\sum_{i \in S} y_i \leq \sum_{l=1}^{k} \frac{|Q_l|}{|S - \bigcup_{q=0}^{l-1} Q_q|}$$

At the first iteration, $V^1 = V$ and $S \cap V^1 = S$ which satisfies the above. At step 2, the intersection set is $S - Q_1$ and so on. Let's look at the worst case scenario; since all Q_i should sum up to S, we get the worst case when all Q_i are of size 1. That gives us the value

$$\leq \sum_{k=0}^{|S|-1} \frac{1}{(|S| - k)} = H(|S|)$$

and we have a $\ln(n)$ approximation.

To give a little intuition to this last proof, suppose that we were looking at the sets S_1, S_2, and S_3 which happened to be the first, second, and third sets to be chosen by the algorithm, respectively. For this example we'll focus the analysis on set S_3. Suppose that S_1 has 10 elements, of which two overlap with S_2 and four overlap with S_3. We give those four elements in S_3 the values 1/10. Suppose S_2 has 8 elements (this can't be more than 10, otherwise it would have been chosen first). Two of its elements were already covered by the first round, and of the remaining six elements, one is in S_3 as well. This common element gets the value 1/6. Finally, S_3 gets chosen, and suppose that it has 9 elements, of which five were assigned to in previous rounds of the algorithm (Notice that because S_2 was chosen before S_3, the number of elements in S_3 is no more than $4 + 6$). The remaining four elements each get value 1/4. So the sum of the y_i in S_3 is

$$1/10 + 1/10 + 1/10 + 1/10 + 1/6 + 1/4 + 1/4 + 1/4 + 1/4$$

But in the first round $|S_3 \cap V'| = 9$, in the second round $|S_3 \cap V'| = 5$, and in the final round $|S_3 \cap V'| = 4$ thus the sum is bounded by:

$$\leq 1/9 + 1/9 + 1/9 + 1/9 + 1/5 + 1/4 + 1/4 + 1/4 + 1/4$$

$$\leq 1/9 + 1/8 + 1/7 + 1/6 + 1/5 + 1/4 + 1/3 + 1/2 + 1/1$$

which is just $H(|S_3|)$, the desired result.

It is easy to check whether this algorithm is useful for a given instance of the problem. We simply need to check $H(n)$ for that instance and see if the approximation is acceptable to us. For example, if all subsets are order of 20, we get $H(20)$ which is almost 3. If the problem has additional structure like a metric (as in the hospital example), we can do better than this; but in the general case, this is basically the best algorithm that can be hoped for.

Notes For further reading see Chapter 3 in [7], Chapters 2,13,14 in [15].

7 Randomized Algorithms

7.1 Maximum Weight Crossing Edge Set

Randomization is a powerful and simple technique that can be use to construct simple algorithms to solve seemingly complex problems. These algorithms perform well in the expected sense, but we need to use inequalities from statistics (e.g. Markov inequality, Chernoff bound etc.) to study the deviations of the results produced by these algorithms from the mean and to impose bounds on performance. We will illustrate the application of this technique with a simple example.

Consider a connected undirected graph $G(V, E)$, with associated weight function $w : E \to \mathcal{R}^+$. Consider any cut of the graph, say $(S, V - S)$, and let E_c denote the corresponding set of crossing edges induced by this cut. The cumulative weight of the crossing edge set is given by

$$W_c = \sum_{(i,j) \in E_c} w(i, j). \tag{3}$$

The objective is to find a cut such that W_c is maximized.

Now consider the following randomized algorithm: For each vertex $v \in V$, toss an unbiased coin, and assign v to set S if the outcome is a head, else assign v to set $V - S$. This produces *a* solution to the problem. The coin tosses are fair and independent, thus each edge can belong to the set of crossing edges E_c with probability 0.5. Thus, the expected cumulative weight produced by the algorithm is half the total cumulative weight of all edges of G. Formally,

$$E\left(\sum_{(i,j) \in E_c} w(i, j)\right) = \frac{1}{2} \sum_{(i,j) \in E} w(i, j). \tag{4}$$

Since the expected weight is half the total weight of all edges, there exists at least one solution with weight equal to (or more than) half the total weight of all edges. This follows because the mean of a random variable can be at most equal to its maximum value. Now, it is obvious that even the optimum weight $W_c^\star$ cannot exceed the sum of weights of all edges, i.e.,

$$W_c^\star \leq \sum_{(i,j) \in E} w(i, j). \tag{5}$$

Thus, the presented approach produces 2x approximate solution to our problem *in expectation.*

How often will this simple randomized algorithm produce a good solution? In other words, can we convert this approach to an algorithm with some reasonable guarantees on performance ? First, consider a situation when all edges in the graph, except one, have zero weight. Since the randomized algorithm will pick the edge with non-zero weight with probability 0.5, we will obtain a meaningful solution only half the time, on an average.

One possible bound on the deviation of a random variable from its mean can be obtained using Markov's inequality. Thus, if Y is a random variable with $E(Y) = \mu$,

$$P(Y \geq \beta\mu) \leq \frac{1}{\beta}. \tag{6}$$

The disadvantage of Markov's inequality is that it produces very loose bounds. The advantage is that we need to know only the first order statistics of the random variable in order to bound its deviation from

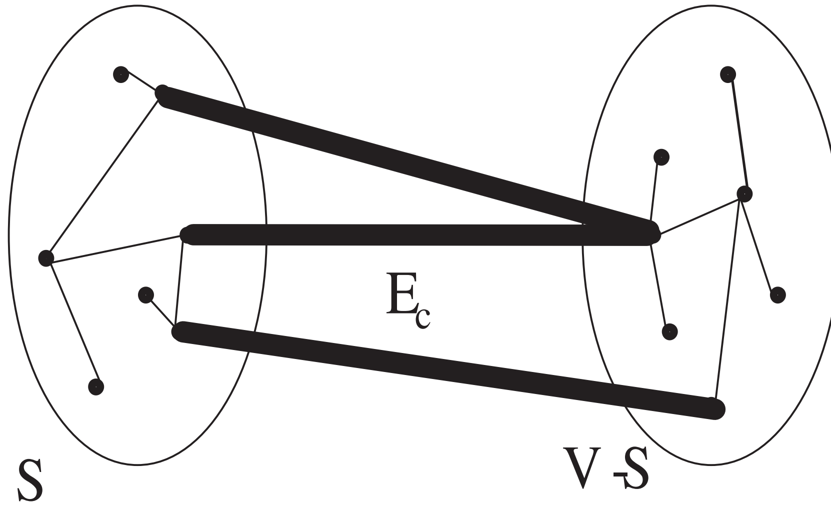


Figure 27: The figure shows the crossing edge set E_c, whose weight we are interested in maximizing

the mean. As discussed below, Markov's inequality can be used to derive tighter bounds like Chebychev's bound and Chernoff bound, which are based on higher order statistics.

A simple proof of Markov's inequality for discrete Y is as follows: Assume the the inequality does not hold. Now,

$$E(Y) = \sum_{y \in \mathcal{Y}} y \cdot P(Y = y) \geq \sum_{y > \beta\mu} y \cdot P(Y = y) > \frac{1}{\beta} \cdot \beta\mu, \tag{7}$$

which is a contradiction since we $E(Y) = \mu$.
QED.

To use Markov's inequality, define Y as the random variable equal to the sum of weights of edges that are *not* crossing the cut. As before, it is easy to see that $E(Y) = W/2$, where W is the sum of the weights of all the edges. Using Markov's inequality, we see that the probability of Y exceeding (say) $1.2E(Y) = 0.6W$ is at most $1/1.2$, so with probability at least 0.17 we get a cut with weight at least $(W - 0.6W) = 0.4W$. Consider the following algorithm:

- Randomly assign nodes to S and $V - S$
- Compute weight of crossing edges, compare with W
- Repeat until weight of crossing edges is at least $0.4W$

Our calculation above implies that expected number of iterations is $1/0.17 = 6$ we get cut that includes at least $0.4W$ edges, in other words we have 2.5-approximation. Note that the constants were chosen to make the example specific. We can choose alternative constants. For example, choosing 1.1 instead of 1.2 will result in a bit better approximation and slower (expected) running time of the resulting algorithm. Note that this approach inherently cannot improve the approximation factor beyond 2.

It is useful to note that one can use tighter bounds (e.g. Chernoff discussed in the next section), that will result in better performance. We are going to skip this due to the fact that the local search algorithm presented below is better and simpler anyway.

Next we consider the *Local Search* approach to the above problem. Conceptually, this approach can be viewed as a walk on a graph, whose vertices are comprised of feasible solutions to the problem at hand. Transforming one solution to another is then equivalent to moving from a vertex on a graph to one of its neighboring vertices. Typically, we start with a reasonable guess, and apply our transformation iteratively until the algorithm converges. This will occur when we reach a "local optimum". However, for this method to be useful, is is critical that the number of neighboring vertices (or solutions) is relatively small, in particular not exponential. We will illustrate this technique for the maximum cardinality crossing edge set problem.

In the maximum cardinality crossing edge set problem we can construct a neighboring solution by moving one node from set S to $V - S$, or vice-versa. In particular, we choose a node that has more edges inside its parent set than crossing into the other set. Since the number of crossing edges increases with each iteration, this algorithm converges in at most $|E|$ iterations (i.e., in polynomial time).

We claim that the produced solution is guaranteed to be a 2x approximation to the optimum. At termination, by construction every vertex will have at least as many edges crossing into the other set as within its set. Thus, the number of edges in the crossing edge set will be at least half the total number of edges. In other words, at least half of all edges cross the cut, implying 2-approximation. The algorithm can be extended to the weighted case where edges have non-negative weights.

7.2 The Wire Routing Problem

7.2.1 Overview

The problem of optimal wire routing subject to constraints comes up in several contexts, e.g., layout in VLSI, routing in MPLS, shipping goods to multiple destinations etc. Here we consider an abstraction of the problem.

Suppose we want to connect various pairs of points on a real circuit with physical wires. For each index i, we have a given pair of points $\{(s_i, t_i)\}$, which we wish to connect with a single wire. In practice, these wires must pass through some existing channels, each of a predetermined width. Therefore, the size of the channel poses a restriction on the number of wires that can pass through it. We can model this constraint with a graph $G(V, E)$, where each edge $e \in E$ can hold at most $cap(e)$ wires. So, the paths between several pairs of vertices may pass through a single edge e as long as the total number of such paths does not exceed $cap(e)$.

The wire routing problem involves finding a set of paths connecting a given set of vertex pairs that is consistent with the capacity restrictions on all edges. In general, this problem is known to be NP-Hard, but we can approximate a solution for it by using a linear program.

7.2.2 Approximate Linear Program Formulation

Let $P_i^{(j)}$ denote the jth path from s_i to t_i. Define an indicator variable $f_i^{(j)}$ corresponding to $P_i^{(j)}$ as follows:

$$f_i^{(j)} = \begin{cases} 1 & \text{if we select } P_i^{(j)} \text{ to connect } s_i \text{ to } t_i \\ 0 & \text{if we do not select } P_i^{(j)} \end{cases}$$

We can formulate the following constraints:

$$\forall i : \sum_j f_i^{(j)} = 1 \tag{8}$$

$$\forall e : \sum_{e \in P_i^{(j)}} f_i^{(j)} \leq cap(e) \tag{9}$$

$$f_i^{(j)} \in \{0, 1\} \tag{10}$$

The first constraint implies that we desire exactly one path from s_i to t_i, $\forall i$. The second constraint is the capacity constraint on the edges, while the third constraint stems from the definition of the indicator variables. So far, we do not have a linear program because we have no objective function and we have an integer constraint. To tackle the first issue, we relax the integer constraint on $f_i^{(j)}$, and replace it with the constraint $f_i^{(j)} \geq 0$. It may be noted that constraints of the form $f_i^{(j)} \leq 1$ are redundant. Next we introduce a dummy objective λ, and modify our capacity constraint as follows:

$$\forall e : \sum_{e \in P_i^{(j)}} f_i^{(j)} \leq \lambda cap(e). \tag{11}$$

Our objective is now

$$min.\ \lambda, \tag{12}$$

subject to the above constraints. Thus, we converted a feasibility IP into a regular LP. Note that, in general, our LP might have exponential number of variables. We will deal with this issue later.

If solving our LP produces an optimum $\lambda^\star > 1$, it implies that no feasible solution exists for the (original) wire routing problem. If $\lambda^\star \leq 1$, it indicates that there is a chance that the original problem has a solution. No guarantees, since we are solving a relaxed problem. Note that in this case fractional solutions are not acceptable to us, because physically it would mean splitting wires, which is clearly impractical.

7.2.3 Rounding

To "round", we treat the combination of $f_i^{(j)}$'s between two points as a probability distribution. This is easy, since $\sum_j f_i^{(j)} = 1$ and $f_i^{(j)} \geq 0$. We select $P_i^{(j)}$ from the distribution over all paths:

$$\forall i : \text{Randomly choose } P_i^{(j)} \text{ with probability } f_i^{(j)}.$$

This can be visualized as juxtaposing the $f_i^{(j)}$s (for each i) along a unit interval, realizing a $U[0, 1]$ random variable, and picking the $f_i^{(j)}$ corresponding to the interval in which this random variable lies.

7.2.4 Analysis

The expected number of paths through edge e will be:

$$\sum_{e \in P_i^{(j)}} Pr(P_i^j\ chosen) \cdot 1 = \sum_{e \in P_i^{(j)}} f_i^{(j)\star} \cdot 1 \leq \lambda^\star cap(e).$$

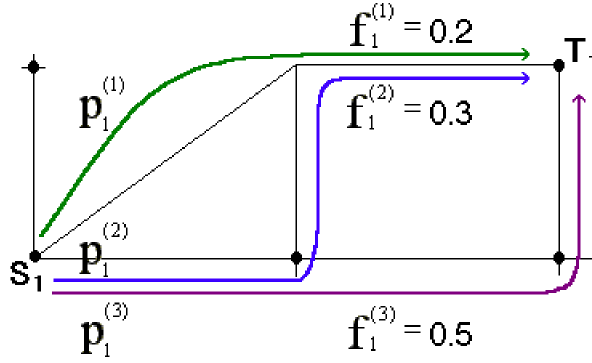


Figure 28: The above shows the part of the graph which contains all paths from S_1 to T_1, and the associated values in the optimum fractional solution.

As we can see, at least in the expected case, we did not overflow the capacity. However, we also must consider the "spread" of the distribution around the expected value; if the spread is large, we may still have a large probability of exceeding the edge capacity.

It is useful to try to use Markov's inequality to claim that probability of overflowing an edge by "a large factor" is small. In particular, given a specific edge, Markov's inequality can be used to claim that probability of overflowing $\lambda cap()$ by a factor of 2 is at most 1/2. Unfortunately this is not enough - we need to bound probability of overflowing by some factor of *any one* of the edges. In other words, we need $1/(constant \cdot m)$ bound on the probability of significantly overflowing a specific edge.

Markov inequality is quite general and does not take into account that our random variable (total number of paths allocated on a specific edge) consists of a sum of several 0-1 random variables. Thus, we will use *Chernoff Bound* instead.

The Chernoff bound theorem applied to our case will look as follows:

Theorem 7.1. *Let $X_1, X_2, ..., X_n$ be independent Bernoulli trials such that, for $1 \leq i \leq n, P(X_i = 1) = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_i X_i, \mu = E[X] = \sum_i p_i$,*

$$Pr(X \geq (1+\beta)\mu) \leq \begin{cases} e^{\frac{-\beta^2\mu}{4}} & \text{for } \beta \in (0, 2e-1] \\ 2^{-(1+\beta)\mu} & \text{for } \beta > 2e-1 \end{cases}$$

In our problem, each X_i corresponds to a path $P_i^{(j)}$ and $p_i = f_i^{(j)}$. We exceed the expected number of paths through an edge by a factor of $1+\beta$.

Example: Let $\beta = 0.1$. If the expected capacity $\mu = 1000$, what is the probability of overflow by a factor of $\beta = 10\%$?

$$P(X \geq 1100) \leq e^{\frac{-0.01*1000}{4}} = e^{-2.5} = 0.0821$$

As β increases, the probability bound drops off exponentially. Probability of overflow by $\beta = 20\%$ is:

$$P(X \geq 1200) \leq e^{\frac{-0.04*1000}{4}} = e^{-10} = 0.0000454$$

The above formula restricts the probability of capacity overflow of a single edge. Let ϵ be the desirable upper bound on the probability with which the capacity of *any* edge in the graph is exceeded. If we restrict the probability of overflow for a specific edge to be less than or equal to ϵ/m, where m is the number of edges in the graph, then for a single edge we have:

$$P(X \geq (1+\beta)\mu) \leq \epsilon/m$$

Note that we can always find a suitable value for β to achieve this bound. Using union bound, the total probability of overflow of any edge is bounded by $\sum_m \epsilon/m = \epsilon$ (from the union bound). Overall, we have

$$P(\text{any } X \geq (1+\beta)\mu) \leq \epsilon$$

which is the bound we were trying to obtain. It shows that the integer solution we obtain by randomly selecting $P_i^{(j)}$ with probability $f_i^{(j)}$ is most likely (probability of at least $1 - epsilon$) a "good" solution for our problem.

So how good is the solution that we have obtained ? It depends on the value of β. Note that β has to be large enough to ensure that the probability of failure of a single edge (where "failure" means that this edge is overcommitted by a factor above $1+\beta$) should be below ϵ/m. The Chernoff bound formulas above can be used to compute β as a function of μ.

Now consider a specific edge e with capacity $cap(e)$ that was loaded to some value $\mu(e) = \sum_i \sum_{j:e \in P_j^{(i)}} f_j^{(i)}$ in the fractional solution. By construction, we have $\mu(e) \leq \lambda^* cap(e)$, where λ^* is the smallest "capacity stretch" λ that permits us to solve the fractional formulation of the problem.

In order to simplify the calculations, lets just assume that $\mu(e) = cap(e)$. If it is significantly smaller, we can add a dummy 0-1 random variable that is 1 with probability $cap(e) - \mu(e)$. After rounding, we will just disregard this variable. This can only improve our solution.

First consider the case where $cap(e)$ is small, say constant. Then we can set $\beta = \Theta(\log n\epsilon^{-1})$. Substituting into the second Chernoff bound immediately tells us that this value is "sufficiently large". Thus, in this case we get a solution that does not overflow the optimum capacities by more than a logarithmic factor.

Now consider the case where capacities are large, say $10 \log n$ or above. Then we can use a much smaller β ! In particular, substituting $\beta = 1$ into the first Chernoff bound tells us that our rounding with overflow by more than a factor of 2 with probability below 1/2.

Additional issues There are a couple of additional issues that we need to address. First, the application of Chernoff bounds for $\sum X_i$ requires that the variables X_i be independent. The paths between a pair of vertices are not independent, because choosing path $P_i^{(j)}$ excludes every other path between these vertices from consideration. However, we can set each variable X_i to represents whether a selected path between vertices s_i and t_i passes through the edge we apply the bound on (essentially aggregating all the f-values for different paths of the same pair that use this edge). Since random choices of paths between different pairs of vertices are independent from each other, the required independence condition is indeed satisfied.

Second, since our LP requires separate variables for all paths between all given pairs of vertices, its size is exponential. However, it is possible to solve the above LP in polynomial time. We will discuss the techniques needed for this in subsequent sections. In fact, we will show that there exists an optimum solution with only a polynomial number of variables having non-zero value footnoteThis arises from the dimensionality of the problem, and the fact that a solution of the LP must lie at the vertex of a polytope..

Importance of vertex solution In section 4 we showed that if the problem is feasible, then optimum objective value is achievable in a vertex of a polytope. Lets apply this claim to a variant of wire routing problem. Specifically, consider the following LP:

$$\begin{aligned} \forall i : \sum_j f_i^{(j)} &= 1 \\ \forall e : \sum_{e \in P_i^{(j)}} f_i^{(j)} &\leq \lambda cap(e) \\ \forall i, j : f_i^{(j)} &\geq 0 \\ \text{minimize } \lambda \end{aligned}$$

We will show that there exists a solution with only a polynomial number of non-zero variables. (Note that this still does not explain how to actually solve this LP; this question is left for the homework.)

Let the number of f_i^j variables in the LP, the number of (s_i, t_i) pairs and the number of edges in G be k, l, m respectively. Note that k can be exponential in the size of the graph.

Now as we have k variables (f_i^j), the feasible region of the LP must lie in a k dimensional space. This polytope will be constrained by $k + l + m$ hyperplanes, each representing the boundary value of the inequality that it corresponds to (i.e. if an inequality is $x_1 \geq 0$, the hyperplane corresponding to it will be $x_1 = 0$.) Now a vertex in this polytope must satisfy k equalities since a vertex must lie on the intersection of at least k-hyperplanes.

We know that one of the optimum solutions to the LP lies on a vertex of the (feasible region) polytope. Denote this solution by f^*. The fact that f^* is a vertex implies that it satisfies with equality at least k (problem dimension) of the constraints. Thus, there can only be a maximum of $m + l (= k + m + l - k)$ constraints that are not satisfied as equalities.

Now consider the following inequalities of the LP:

$$\forall i, j : f_i^j \geq 0$$

We have just shown that in f^*, at most $m + l$ of *all* inequalities are not satisfied as equalities. This implies that in f^*, at most $m + l$ f_i^j can be greater than 0. Hence there exists an optimum solution where only a polynomial number of the f_i^j are greater than 0.

Notes The randomized rounding approach described in this section was introduced in [12]. To further study the subject of randomized algorithms see [1, 11, 10]. To learn more about approximation algorithms for max-cut, see [5].

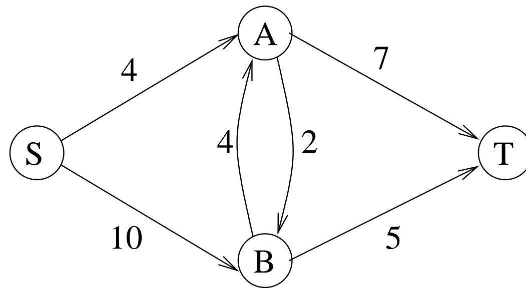


Figure 29: Each edge in this graph is labeled with its capacity.

8 Introduction to Network Flow

8.1 Flow-Related Problems

The optimization problems we have examined so far have had no known algorithms that solved the problems exactly, requiring us to devise approximation algorithms. We will now examine a class of problems that deal with flow in networks, for which there exists efficient exact algorithms. One can view these flow-related problems as tools. As we will see later, many problems can either be reduced directly to flow, or can be approximated by solving an appropriately constructed flow problem.

Maximum flow problem: Given a directed graph $G = (V, E)$, we assign to each edge $uv \in E$ a nonnegative (and integral in our discussion) *capacity* $cap(uv)$. If $uv \notin E$ we assume $cap(uv) = 0$. One of the nodes of G is designated as the source s, and another as sink t. The goal of the problem is to maximize the flow from s to t, while not exceeding the capacity of any edge.

Note: Although the *max-flow* problem is defined in terms of directed graphs, almost all techniques we will discuss apply to undirected graphs as well.

Maximum flow problems arise in a variety of contexts. For example:

1. *Networks*

   Every edge represents a communication link in the network. Information has to travel from s to t. Capacity of an edge edge is equal to its bandwidt, i.e. number of bits per second that can travel along this edge.

2. *Transportation*

   A truck is loaded at s and has to deliver its cargo to t. The edges of the graph are roads, and the capacities are the number of trucks per hour that can travel on that road.

3. *Bridges*

   Each vertex represents a physical location and each edge represents a bridge between two locations. The capacity of an edge is the cost to block the bridge it represents. The goal is to disconnect s and t while incurring the minimum cost in blocked bridges.

There are two sides to our flow optimization problem. We will state them informally first.

1. Transport maximum rate of material from s to t, as in examples 1 and 2 above. This is known as the *max-flow* problem. For example, as in the graph in Figure 29, we can construct the following flow:
   - Send 4 units along the path $s \to a \to t$.
   - Send 3 units along the path $s \to b \to a \to t$.
   - Send 5 units along the path $s \to b \to t$.

   The total flow is 12, which happens to be the maximum flow for this graph.
2. Cut edges of G to disconnect s and t, minimizing the cost of the edges cut, as in example 3 above. This is known as the *min-cut* problem.

   In the graph in Figure 29, it is easy to see that cutting edges at and bt will disconnect s and t, since there will be no edges going into t. The capacity of these edges is $cap(at) + cap(bt) = 12$. This happens to be the lowest capacity cut that separates s and t.

We will show the formal equivalence of *max-flow* and *min-cut* problems later.

We will now define the *max-flow* problem formally. For notational convenience, consider "mirror" edges. For each edge $e = (u, v)$ in $G = (V, E)$ with capacity $cap(e)$ and flow $f(e)$, we will add a "mirror" edge $e' = (v, u)$ with capacity 0 and flow $-f(e)$. Note that if both uv and vu are present in E, we will now have four edges between u and v.

Definition 8.1. *A* flow *is a real-valued function f defined on the set of edges $uv \in V \times V$ of the graph $G = (V, E)$ subject to two constraints:*

1. Capacity constraint. $f(uv) \leq cap(uv) \forall u, v \in V$, *where $cap(uv)$ is the capacity of edge uv.*
2. Conservation constraint. *The following equation holds for all $u \in V - \{s, t\}$*

$$\sum_{v:uv,vu \in E} f(uv) = 0.$$

The capacity constraint limits the flow from above. Note that the "mirror" edges added to the graph satisfy this constraint if the flows through edges of E are nonnegative.

The conservation constraint forces inflow and outflow for a node to sum to 0 for all nodes, where inflow is measured in negative numbers and outflow is measured in positive numbers. This requirement does not hold for the source (s) or the sink (t), but together their flow sums up to 0. (Can you explain why ?)

We can see in Figure 30 that all the capacity constraints and conservation constraints are satisfied. The shown flow is therefore a *feasible flow* on G.

Definition 8.2. *A* cut *in a graph $G = (V, E)$ is a partition of the set of vertices of the graph, V, into two sets A and $V - A$, where $A \subseteq V$.*

We can also think of a cut as a set of edges that go from A to $V - A$, i.e. all edges uv such that $u \in A, v \in V - A$. If we remove (cut) these edges from the graph, no vertex in A will be connected to a vertex in $V - A$. The capacity of a cut is defined as follows:

$$cap(A, V - A) = \sum_{u \in A, v \in V - A, uv \in E} cap(uv).$$

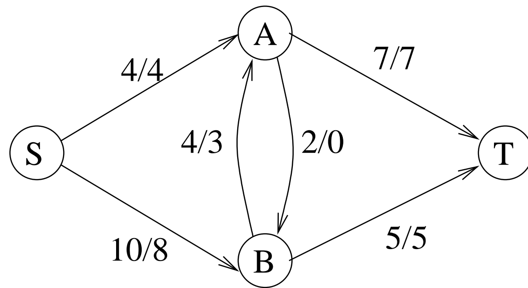


Figure 30: A feasible flow in a graph. Each edge is labeled with capacity/flow. Mirror edges are not shown.

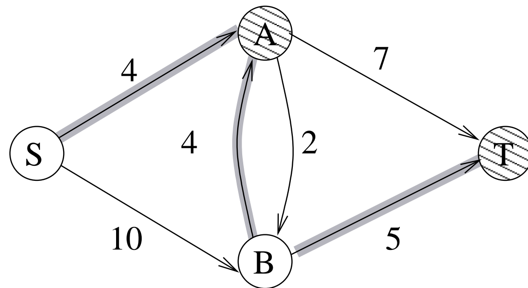


Figure 31: An example of a cut in a graph. Let the set A consist of the unshaded nodes and $V - A$ consist of the shaded nodes. Edges between the partitions of the cut $(A, V - A)$ are highlighted. The capacity of the cut is $cap(sa) + cap(ba) + cap(bt) = 13$. This is not a minimum cut.

The direction of edges is important in the definition of a cut. Capacities of edges that go from $V - A$ to A are not counted towards the capacity of the cut (see Figure 31). This is because we want the capacity of a cut to limit the flow going through that cut in one direction.

Definition 8.3. *The* flow across a cut $(A, V - A)$ *on a graph* $G = (V, E)$ *is given by:*

$$f(A, V - A) = \sum_{u \in A, v \in V - A} f(uv).$$

Note the difference between this definition and the definition of the capacity of a cut. To calculate a net flow through a cut we must take into account flows going in both directions. Hence, the condition $uv \in E$ is removed allowing us to consider mirror edges from A to $V - A$.

Definition 8.4. *An s-t cut on a graph* $G = (V, E)$ *is a cut* $(A, V - A)$ *on* G *such that* $s \in A$ *and* $t \in V - A$.

Maximizing Flow. In order to maximize flow, it is necessary to maximize $f(\{s\}, V - \{s\})$, which is the sum of the flows on all edges going out of the source. Equivalently, we can maximize $f(V - \{t\}, \{t\})$, which is the sum of the flows on all edges going into the sink. It can be proven that these values are equal. In fact, it can be proven that the flows across all *s*-*t* cuts are the same (see homework). Denote this value by $|f|$.

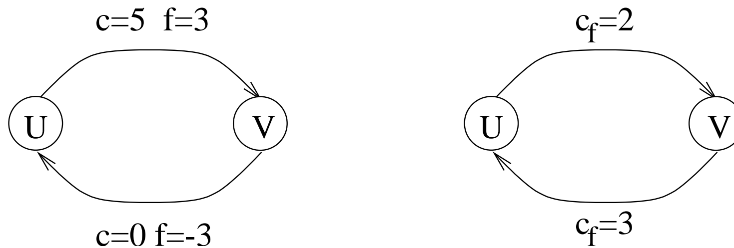


Figure 32: Residual capacities of edges. We can see that if a flow of 3 has been pushed across an edge with capacity 5, we could push 2 more units of flow or 3 fewer units of flow across that edge. This gives us our two residual edges of 2 and 3 respectively.

8.2 Residual Graphs

Currently we have to keep track of two values for each edge of G, capacity and flow. *Residual graphs* will allow us to only keep track of capacity for each edge, updating this capacity as the flow changes. Residual graphs represent how much additional flow can be "pushed" along a given edge. We can always construct the current flow in G from the current residual graph and original capacities of G.

To create a residual graph, consider for each edge uv the remaining capacity (residual capacity) of that edge after some flow $f(uv)$ was pushed across it. Denote this value as $cap_f(uv)$, then

$$cap_f(uv) = cap(uv) - f(uv).$$

Figure 32 shows residual capacities of an edge uv of G and its "mirror" edge vu after a flow of 3 units was pushed across uv.

Definition 8.5. *A residual graph of $G = (V, E)$ given a flow f is defined as $G_f = (V, E_f)$, where*

$$E_f = \{uv \in VxV \mid cap_f(uv) > 0\}.$$

If G_f has a directed path from the source to the sink, then it is possible to increase the flow along that path (since by the above definition each edge in E_f has positive capacity). Increasing the flow on such a path is called *augmentation*.

Consider the following simple algorithm for finding the *max-flow* in a graph.

Algorithm for finding *max-flow* in a directed graph

1. Compute initial residual graph (assume flow on each edge is 0).
2. Find a directed path from the source to the sink.
3. Augment the flow on the path by amount equal to the minimum residual capacity along this path.
4. Update the residual graph.
5. Repeat from step 2 until there is no augmenting path

We need to show that the algorithm will eventually terminate. Observe that if the capacities are integer, we will always augment the flow by at least one unit. Thus, there is an upper bound on the

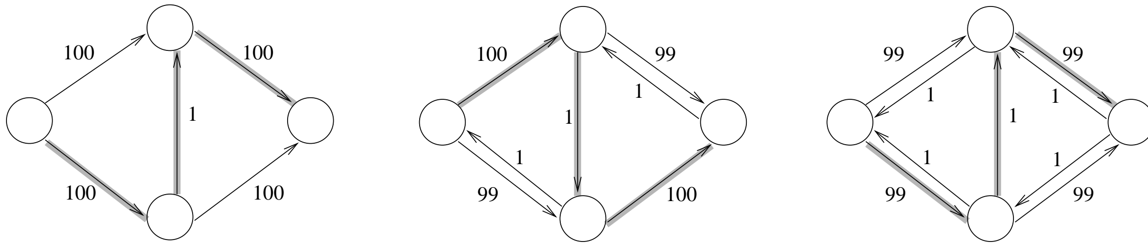


Figure 33: This figure shows a sequence of augmentations. Although we can see that the value of the *max-flow* is 200, the algorithm may make unlucky depth-first choices and perform 200 iterations, the first two of which are shown here.

number of iterations of the algorithm. Indeed, if the maximum capacity of an edge is U and number of nodes is n, then the algorithm will perform at most Un iterations. This is because the upper bound on the flow in the graph is $cap(\{s\}, V - \{s\})$, which is the maximum amount of flow that can go out of s.

The algorithm used at step 2 runs in time $O(m)$ (think about depth-first graph traversal), and the total running time of the algorithm is therefore $O(nmU)$. This is not a polynomial algorithm, since the running time is a function of U. However, if U is small, the algorithm might be efficient. In particular, for constant U we have $O(nm)$ running time.

The following example illustrates that the above algorithm is, in fact, non-polynomial. In other words, it shows that the algorithm is indeed slow and that our analysys is not too pessimistic. Figure 33 shows an example of execution that terminates in $\Omega(Un)$ iterations.

So now we know that, in general, our algorithm is slow. But note that we have not even shown that the algorithm produces an optimum solution ! We will prove this fact in the next section. Meanwhile we will note several easy to see properties:

1. The algorithm builds a flow which cannot be augmented, i.e. is maximal. (We have not shown yet that it is *maximum.*)
2. If the *max-flow* problem has integer capacities, then the algorithm builds an integer flow.

8.3 Equivalence of min cut and max flow

In this section we will use the following terms:

max-flow—the maximum flow that we can pump from the source s to the sink t (the flow which maximizes $|f|$, where $|f|$ is the flow across any s-t cut),

min-cut—the s-t cut of the smallest capacity.

Theorem 8.6. *Given a capacitated graph $G = (V, E)$, the following statements are equivalent:*

a. f is a max flow.

b. There is an s-t cut such that the capacity of the cut is equal to the value of f.

c. There is no augmenting path in E_f (the residual graph with respect to the flow f).

Proof:

b ⇒ a Notice that for every s-t cut $(A, V-A)$, the flow $f(A, V-A)$ from A to $V-A$ is less than or equal to $cap(A, V-A)$, the capacity of the cut:

$$f(A, V-A) = \sum_{u \in A, v \notin A} f(uv) \leq \sum_{u \in A, v \notin A} cap(uv) = cap(A, V-A).$$

Therefore, the flow across any cut cannot exceed its capacity.

Also notice that $f(A, V-A)$ is the same for all s-t cuts $(A, V-A)$. (This is established in homework.) Consequently, the amount of any flow is less than or equal to the capacity of the minimum s-t cut. So if $|f|$ equals the capacity of the min cut, it cannot be increased any further, and therefore, f is a max flow.

a ⇒ c Assume to the contrary that f is a max flow and that there exist at least one augmenting path. But then we can use the augmenting path to augment f, increasing its value and contradicting the assumption that f is max flow. Thus, if f is max flow then no augmenting path can exist.

c ⇒ b This is probably the most "interesting" direction. First, define the set $A = \{v \in V \mid v \text{ is reachable from } s \text{ in } E_f\}$. Recall the definition of E_f. If a vertex v is in A, it means that there is a path of edges with positive residual capacity from s to v.

The sink t is not in A since there is no augmenting path. The vertex s is in A trivially. Now, consider an s-t cut $(A, V-A)$. For every $uv \in E$, where u in A and v in $V-A$, we have $f(uv) = cap(uv)$, for if this were not the case, v would be reachable from s. Similarly, for any $vu \in E$ where u in A and v in $V-A$, we have $f(uv) = 0$. Summing over the edges, we obtain $f(A, V-A) = cap(A, V-A)$.

∎

Observe that our *max-flow* algorithm from the previous section creates a flow such that there are no remaining augmenting paths, i.e. exhibits the properties of part c of our theorem. Thus, our algorithm finds a maximum flow. Observe that if the capacities are integer, we will always augment by an integer amount (easy proof by induction on the number of augmentations). Thus, we have the following useful corollary:

Corollary 8.7. *If all capacities are integers, then there is an all-integer max flow. This flow can be built by iteratively augmenting paths in the residual graph.*

This corollary does not mean that there are no fractional solutions. They usually do exist, but there is at least one integral solution.

8.4 Polynomial max-flow algorithms

8.4.1 Fat-path algorithm and flow decomposition

In this section we present a polynomial time algorithm for solving the max flow problem. Similarly to the Ford-Fulkerson algorithm presented earlier, this algorithm is based on the idea of successive augmentations. However, the previous algorithm had a running time exponential in the length of the representation of the input. The main difference here is that, at each step, we try to find a "fat" augmenting path, i.e. we try to augment by as much as possible. The running time of the algorithm is $O((m + n\log n)m\log(nU))$, where n is the number of nodes, m is the number of edges and U is the maximum edge capacity. If we represent U in our input in binary, this algorithm has a polynomial running time, though not strongly polynomial.

The algorithm The algorithm itself is very simple and relies on our ability to always find an augmenting path of maximum residual capacity, i.e. a path from s to t such that its bottleneck residual capacity is greatest. This can be done efficiently (in $m + n\ln n$ time) by a modified Dijkstra's algorithm.

The algorithm can be defined as follows:

1. Initially f is 0. Find an augmenting path of maximum residual capacity.
2. Augment the flow along this path by the path's bottleneck capacity.
3. Recompute G_f. Repeat steps **1** and **2** until there are no augmenting paths remaining.

First, notice that this algorithm is in fact correct since it stops only when it can no longer find an augmenting path, which by max-flow/min-cut theorem implies that we have a max flow.

Analysis The proof of the running time of this algorithm will rely on two claims. The first is the Decomposition Theorem, and the other is that the optimal flow minus current (feasible) flow is a feasible flow in the current residual graph.

The main idea behind the Decomposition Theorem is that it is possible to separate every flow into a flow along a limited number of cycles, C_i, and along paths from source to sink, P_i. In order to simplify notation, we will use C_i to denote a vector where each coordinate corresponds to an edge and where it is equal to 1 if and only if the corresponding edge belongs to cycle C_i. We will use similar notation for paths. Formally, the *Decomposition Theorem* can be stated as follows:

Theorem 8.8. *The flow in a graph G can be subdivided into flows along cycles C_i and paths P_i in such a way that:*

$$f = \sum_i f(C_i) \cdot C_i + \sum_i f(P_i) \cdot P_i$$

where f is the flow vector and $f(C_i), f(P_i)$ are scalars, representing flows assigned to paths and cycles. Moreover, the total number of paths and cycles is bounded by m, the number of edges in G.

Proof: We can prove the Decomposition Theorem by construction. Let G^f denote the flow graph, i.e. it has an edge uv if $f(uv) > 0$. We will use the following procedure for decomposing the flow. The idea is to iteratively find paths and cycles, assign flow to them, and update the flow graph G^f by removing this flow from the appropriate edges. At each step, we maintain that the current flow in G^f together

with the flows in already constructed paths and cycles sums up to exactly the original flow. At the end, we have G^f flow equal to zero.

1. Start at the source s and follow edges in G^f until either we reach the sink or we close a cycle. Conservation constraints imply that if, during our walk, we arrived to some node $v \notin \{s, t\}$ over edge uv with $f(uv) > 0$, there has to be another edge vw with $f(vw) > 0$. In other words, we will not "get stuck". There are 2 cases to consider:
   (a) We reached t over a simple path. In this case, denote this path by P, compute the minimum of all the flow values on edges of this path ($\min_{vw \in P} f(vw)$) and set $f(P)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(P) \cdot P$$
   (b) We visit a node for the second time. This means that our path includes a cycle. Denote this cycle by C. Compute the minimum of all the flow values on edges of this cycle ($\min_{vw \in C} f(vw)$) and set $f(C)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(C) \cdot C$$
2. If there is at least one edge from s with non-zero flow on it, go back to (1).
3. If there are no outgoing edges from s in current G^f, repeat the same with t.
4. When we reach this point, s and t do not have outgoing edges in G^f. At this point we have a flow f that *satisfies conservation constraints at all nodes*, including s and t. (Try to formally prove this !)
5. Now repeat the following, until there are no more edges in G^f:
   (a) Pick an edge uv in G^f, i.e. $f(uv) > 0$. By conservation constraints, there has to be an edge vw with $f(vw) > 0$. Move to vw and continue. This walk can stop only if we close a cycle. Denote this cycle by C.
   (b) Compute the minimum of all the flow values on edges of C ($\min_{vw \in C} f(vw)$) and set $f(C)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(C) \cdot C$$

Observe that each time we find a path or a cycle, we update flow f in a way that guarantees that at least one of the edges leaves G^f. Thus, the final decomposition will include at most m cycles and paths. ∎

We will need the following lemma for the analysis of the running time:n

Lemma 8.9. *If f^* is the maximum flow on a graph G, then for any feasible flow f, $f^* - f$ is also a feasible flow on the residual graph G_f.*

Define $cap(uv)$ to be the capacity of the edge (u, v) in G, and let $cap_f(uv)$ be the capacity of an edge in the residual graph G_f.

For any edge (u, v) of G_f, we have that if $f(uv) \leq f^*(uv)$, then

$$f^*(uv) \leq cap(uv) \Rightarrow f^*(uv) - f(uv) \leq cap(uv) - f(uv) = cap_f(uv).$$

Thus, the flow of the edge (u, v) in the flow $f^* - f$ does not exceed the capacity of the residual graph G_f. Similar proof shows that the claim is true for the case where $f(uv) > f^*(uv)$. ∎

Now we are ready to prove the bound on the running time of the algorithm. This proof will show basically that at each augmentation cycle, the algorithm will augment by at least $1/m$ of the remaining difference between the value of maximum flow and the value of the current flow. After m iterations, the current (before the first iteration) flow will be reduced by a factor of approximately $1/e$. Using this, one can show that the actual number of augmentation cycles that can occur is bounded from above by $O(m \log(nU))$ where U is the maximum capacity of any edge in the graph.

Theorem 8.10. *The fattest-augmentation path algorithm terminates in $O(m \log(nU))$ iterations.*

Consider the flow f which we have after some number of augmentation steps. If f^* is the optimal flow, then from our lemma we know that $f^* - f$ is a feasible flow. Then, from the Decomposition Theorem, we can break $f^* - f$ into at most m paths and cycles.

Observe that $|f^* - f| = |f^*| - |f|$. (Can you formally prove this ?) Moreover, since cycles do not contribute to the flow value, this means that the flow along the paths in the decomposition sums to exactly $|f^*| - |f|$. Thus, there exists at least one path (denote it by P) in the decomposition of flow $f^* - f$ which has a flow of at least $(1/m) * (|f^*| - |f|)$.

Observe that, since $f^* - f$ is a feasible flow in the current G_f, we have $\forall uv \in P, c_f(uv) \geq f(P) \geq (1/m) * (|f^*| - |f|)$. Thus, our algorithm will find a path (not necessarily P) whose bottleneck capacity is not less than $(1/m) * (|f^*| - |f|)$.

Let F^i denote the total flow from source to sink at iteration i of the algorithm, we must have

$$F^i \geq F^{i-1} + \frac{F^* - F^{i-1}}{m}$$

Allowing δ_i to denote $F^* - F^i$ (where, since there is no flow initially, $\delta_0 = F^*$), we have the following inequality:

$$\delta_i = F^* - F^i \leq F^* - (F^{i-1} + \frac{F^* - F^{i-1}}{m}) = \delta_{i-1} - \frac{\delta_{i-1}}{m}$$

Consequently,

$$\delta_i \ \leq \delta_0 \ (1 - \frac{1}{m})^i = F^*(1 - \frac{1}{m})^i$$

Since we are dealing with integer capacities, all flows will also be integers, so if we are within 1 of the solution, we are done. Formally, if $\delta_k \leq F^*(1 - \frac{1}{m})^k < 1$, then $F^k = F^*$. Taking a natural logarithm of both sides of the inequality, we obtain:

$$0 > \ln F^* + k \ln (1 - \frac{1}{m})$$

Using the Taylor expansion $\ln(1 - x) \approx -x - x^2/2 + O(x^3) < -x$, we find that any $k > m \ln F^*$ satisfies this equation, so at most $m \ln F^*$ iterations are needed to get to a max flow state.

Because at most n edges can emanate from s, we conclude that if U denotes the maximum capacity of any edge in the graph, F^* is bounded from above by nU since this is the most that can possibly flow out of the source. ∎

If we remember that finding a maximum augmenting path using Dijkstra's algorithm takes $O(m + n \log n)$ time, we obtain the final running time of $O((m + n \log n) m \ln nU)$. Observe that this is bounded from above by $O(m^2 \log nU \log n)$.

8.4.2 Polynomial algorithm for Max-flow using scaling

We first explore means to extend max-flow solutions of graphs with approximate integer capacities, to find the max-flow in the original graph. Let us consider the case when the approximate capacities are represented by the first i significant bits of the bit-representation of the capacities, denoted by $\text{cap}_i(\text{e})$ for an edge e.

The algorithm proceeds in phases, where phase i computes max-flow f_i for graph with capacities cap_i. Max-flow for $i = 0$ is trivially 0.

The ith significant bit can be either 1 or 0 and hence $\text{cap}_i(\text{e})$ is either $2{*}\text{cap}_{i-1}(\text{e})$ or $2{*}\text{cap}_{i-1}(\text{e}){+}1$. Given f_{i-1}, we need to quickly convert it into f_i. This can be viewed as two steps: first convert it into max flow that satisfies capacities $2{*}\text{cap}_{i-1}$, and then increment some of these capacities by 1 to get cap_i, and update the flow appropriately.

Note that if we have max-flow for certain capacities, then doubling (coordinate-wise) this flow gives us max-flow for doubled capacities. This can be verified by observing that after doubling both flow and capacities, the min-cut is remains saturated and the conservation and capacity constraints are satisfied as well. Thus, $2f_{i-1}$ is max flow for capacities $2{*}\text{cap}_{i-1}$.

Given max-flow for some given set of capacities, consider what will happen if one increments some of the capacities by 1. Observe that the residual capacity of the min-cut grows by at most the number of edges in the min-cut since the residual capacity initially was 0. Thus, incrementing capacities by at most 1 each can increase the value of max-flow by at most m total. This, in turn, implies that, given max-flow before capacity increment, we only need at most m augmentations to get the max-flow for the graph with incremented capacities.

Applying this reasoning to our context implies that to compute f_i, we first double f_{i-1}, compute residual graph with respect to cap_i, and augment at most m times.

There are at most $\log U$ phases, each phase consisting of at most m augmentations, where each augmentation takes at most $O(m)$ time (e.g., using DFS). Total running time is $O(m^2 \log U)$, which is polynomial in the *size of the input*. This type of approach is usually called *scaling*.

8.4.3 Strongly polynomial algorithm for Max-flow

Although our previous algorithm was polynomial, we are not satisfied with the $\log U$ term that appears in our running time, where U is the maximum edge capacity. The reason is that even though our algorithm is polynomial in the *size of the input*, its running time depends on the precision of the capacities. Observe that if all capacities are multiples of some number, we can divide by this number without changing the problem. Moreover, the data representation for the capacities itself may be different and involve more than $\log U$ bits. In this section we will describe an algorithm with running time that depends only on the size of the input graph, i.e. n and m. Such algorithms are called "strongly polynomial". (Strictly speaking, there are several other formal requirements. In particular, we have to make sure that the number of bits needed to represent intermediate results is bounded by a polynomial in n and m times the number of bits in the input.)

The basic idea behind this algorithm is to augment along the shorter paths in the graph first. We begin by constructing a *layered network* for the residual graph G_f, where each layer k consists of nodes that can be reached in k "hops" from the source. We also ensure that nodes in layer k cannot be reached in less than k hops from the source. We do this by using Breadth First Search on the current residual graph G_f. Specifically, we start at the source and place all nodes that can be reached in one hop from

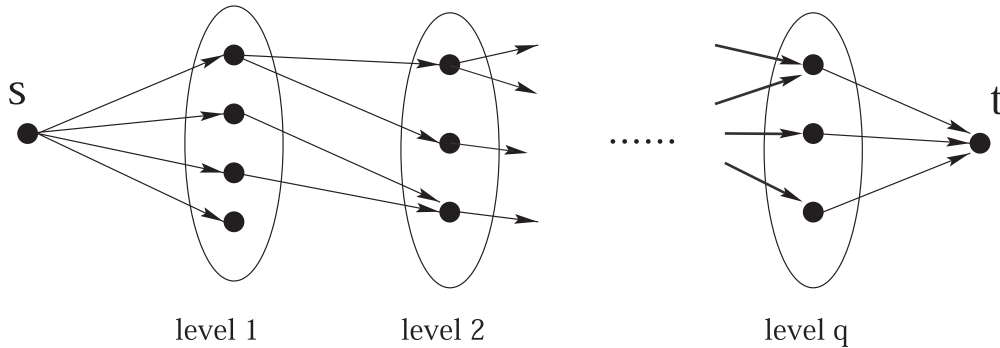


Figure 34: Layered Network

it into layer 1; then we take all untouched nodes that can be reached from layer 1 nodes and place them into layer 2. We continue with the same approach iteratively until we reach the sink or have traversed all edges in G. The running time of this algorithm is the running time of BFS, namely, $O(m)$. Our layered network may look as shown in Figure 34.

We note the following useful facts about this layered network:

1. There can be no edges that skip a layer forward, since if such an edge existed, its endpoint would have to be placed in the level just one greater than its rear endpoint.
2. There can be at most n levels in the network, since no node could be located in 2 different levels, and G contains n nodes.
3. If the sink cannot be reached after n levels, there are no available augmenting paths from source to sink, so we already have max-flow and are done.
4. Nodes may exist beyond the sink, which do not belong to any layer. These nodes will not participate in the current phase since they do not lie on shortest paths.

Our algorithm shall proceed as follows:

1. Construct a layered network from G_f. If it's not possible to reach the sink t in an n-layer network, we have max-flow by the previous remarks and we are done.
2. Find any forward augmenting path in the n-layer network and augment along this path. Only forward residual edges are allowed.
3. Update the residual graph G_f.
4. Repeat step 2 until there is no forward augmenting path left and then restart from step 1.

The above algorithm operates in phases, where at each phase we build a new layered network and perform several augmentations. Observe that since we restrict our search for augmenting paths to those paths that have only forward edges (i.e. edges from layer i to layer $i+1$), every augmentation reduces the number of forward edges by at least 1. This is because the augmentation will saturate the forward edge with the smallest capacity along the augmenting path, essentially removing this edge from G_f. This introduces a “reverse” residual edge, but such edges are not allowed for augmentation until the next phase, i.e. until we rebuild the layered network.

Note that all augmentations during a single phase are along same length paths, where this length is equal to the distance between s and t in the residual graph at the beginning of the phase (i.e., number of layers in the layered network). A phase is terminated when there are no more augmenting paths from using only forward edges. Note that a path using a backwards edge has to be longer than the number of layers in this phase. Thus, if at the beginning of a phase the s to t distance was k, then at the end of the phase it will be at least $k + 1$. In other words, we will have more layers in the next phase.

The running time of the algorithm can be computed as follows:

1. The number of phases is bounded by n because there can be at most n levels in the network and each subsequent phase starts with more layers in the layered network.
2. The number of augmentations per phase is bounded by m because after each augmentation the number of forward edges in the layered network is reduced by at least 1.
3. We can find an augmenting path in $O(m)$ by running DFS in the layered network, disregarding all but the forward edges in G_f.

Therefore, the total running time is $O(m^2 n)$.

The above analysis is not tight. We can significantly improve the bound by reducing the time wasted while searching for augmenting paths. Consider an edge traversed by DFS. If we backtrack along this edge during this DFS, we consider this as a waste. The main observation is that if, during a single phase, we backtrack along an edge, then it is useless to consider this edge again until the next phase. The reasoning is as follows: if we backtrack along the edge uv then there is no forward path from v to t. But augmentations during a phase can only introduce back edges, and hence once we notice that there is no forward augmenting path from v to t, then such path will not appear until the next phase. It is important to note that, initially, there might be a forward path from v to t that is "destroyed" during the phase.

The above discussion implies that each time we backtrack along an edge during DFS, we can mark this edge as "useless" (essentially deleting it) *until the next phase*. Thus, we backtrack over each edge at most once during the phase, which gives us $O(m)$ bound on "wasted" work during the phase.

The "useful" work consists of traversing edges forward, without backtracking. Notice that there can be at most n such edges during a single DFS. (In fact, the number is bounded by the number of layers in the current phase network.)

Combining the above results, we see that the total amount of wasted work during a phase is bounded by $O(m)$ and the total amount of "useful" work is bounded by $O(mn)$ i.e $O(n)$ per augmentation. [Why can't we claim $O(n)$ augmentations?] Since building a layered network takes $O(m)$ time, we get $O(mn^2)$ bound on the running time of the algorithm. Notice that this is a significant improvement over the $O(m^2 n)$ time computed earlier, since m can be as large as $\Theta(n^2)$.

Comparing this bound to the $O(m^2 \log U)$ bound we got in the previous section, we see that our new bound is not always better. The best running time for a strongly polynomial max-flow algorithm is $O(nm \log(n^2/m))$, which we will discuss in subsequent sections.

8.5 The Push/Relabel Algorithm for Max-Flow

8.5.1 Motivation and Overview

In the previous lectures we introduced several max-flow algorithms. There are cases though, where those algorithms do not have good performance, since the only tool we have been using up to now is augmentation. Consider a graph with a many-hop, high-capacity path from the source, s, to another node b, where b is then connected to the sink, t, by many low-capacity paths. Figure 35 illustrates such a scenario. In this topology, the max-flow algorithms introduced earlier must send single units of flow individually over the sequential part of the path (from s to b), since no full path from s to t has a capacity of more than one. This is clearly a fairly time-consuming operation, and we can see that the algorithm would benefit from the ability to push a large amount of flow from s to b in a single step. This idea gives the intuition behind the *push/relabel* algorithm.

More precisely, in the push-relabel algorithm, we will be able to push K units of flow at one time from s to b. If the total capacity from b to t is then less than K, we will push what we can across those lengths, and then send whatever excess remains at b back to s.

For this purpose, we introduce the concept of *preflow.* Preflow has to satisfy capacity constraints. Instead of conservation constraints, we require that for each node v that is neither source nor sink, the amount of flow entering v is at least the amount of flow leaving v. The push-relabel algorithm works with preflow, slowly fixing the conservation constraints, while maintaining that there is no augmentation path from s to t. When the conservation constraints are finally satisfied, preflow becomes the solution to the max flow problem.

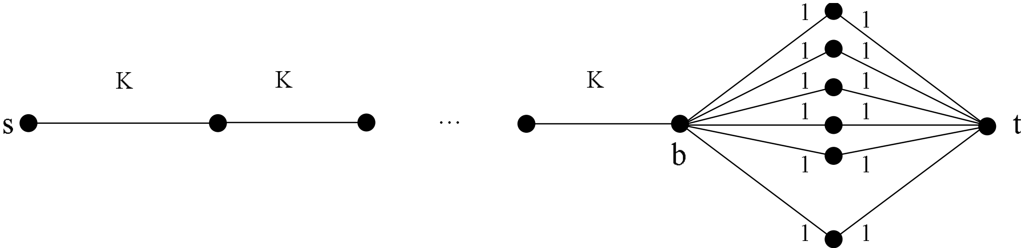


Figure 35: A bad case for algorithms based on augmentation

8.5.2 Description of Framework

The algorithm assigns to each node a value $d(v)$, called the *label.* We fix $d(s)$ at n and $d(t)$ at 0; these values never change. Labels of intermediate nodes are initialized to 0 and updated as described below during the algorithm. At any point in the algorithm, labels must satisfy the *label constraint*:

$$d(u) \leq d(v) + 1$$

for any residual edge (u, v) in the graph. Intuitively one can think of labels as water pressure. According to the laws of physics, water flows from high pressure to the low pressure. This algorithm follows an analogous principle, it pushes flow from a higher label to a lower label.

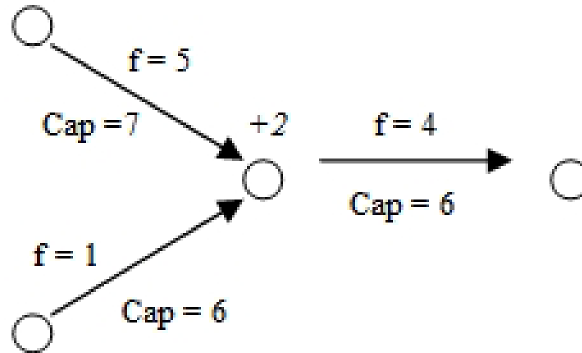


Figure 36: A portion of a graph with excess

The algorithm begins by saturating each of the edges at the source with preflow. This creates excess at the nodes at the other end of these edges, meaning there is more flow entering than leaving them. This also saturates all edges going out of the source, therefore there are no residual edges from source to any node in the residual graph. Hence the label constraint is satisfied. The above-mentioned violation of the conservation constraint allowed by preflow is precisely this excess; violation in the other direction—that is, more flow leaving than entering a node other than the source—will not occur. Nodes with excess are called *active nodes*.

Push-relabel algorithm tries to "fix" active nodes, by trying to push excesses to sink. A push operation can be thought of moving excesses in the graph. For example, consider the portion of a graph in Figure 36, after a one unit push from the center node to the right, the residual graph looks like Figure 37 after a two unit push in the same direction the residual graph is like Figure 37. Note that we cannot push any further because of capacity constraints. The first is an example of a *nonsaturating push* and the second is an example of a *saturating push* since the residual edge is saturated.

From this point, we proceed to push preflow across residual edges in the graph until no active nodes remain, at which point we have reached a final state. In each of these pushes, we move as much flow as possible without exceeding either the capacity of the residual edge or the quantity of excess on the node we are pushing from. Further, we may only push from a node v to another node u if the two nodes satisfy the equation:

$$d(v) = d(u) + 1$$

This restriction maintains the label constraint for the resulting (u, v) residual edge. An edge that satisfies the restriction is called *admissible.*

It is evident that, after the initial pushes from the source described above, no further pushes are immediately possible, since all interior nodes were initially labeled with a value of 0—clearly, we would be unable to push from any node v with excess, since $d(v)$ is 0 and there is no node u with $d(u) = -1$.

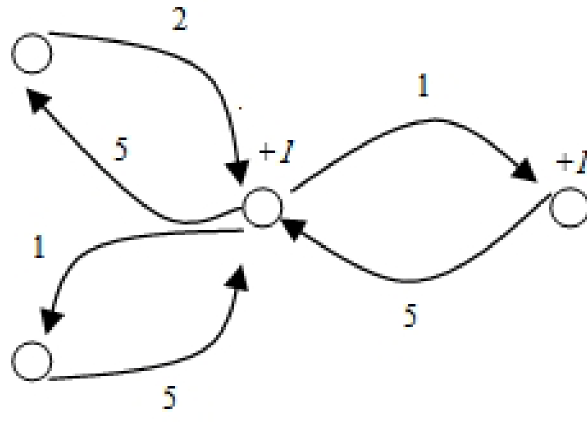


Figure 37: Residual graph after a nonsaturating push

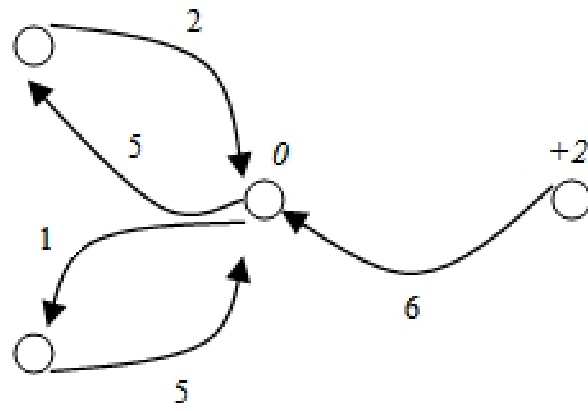


Figure 38: Residual graph after a saturating push

The next step, therefore, is to relabel nodes. When relabeling, we increase the labels of all active nodes as far as possible without violating the label constraint, i.e. we set the label of v to the minimum label of the neighboring nodes plus 1. Formally,

$$d(v) \leftarrow 1 + min(d(u))$$

where (v,u) is a residual edge

Relabeling a single active node v increases its label to precisely the point where it is possible to push something over a residual edge, since after relabeling, we must have a residual edge to some w such that $d(v) = d(w)+1$. Otherwise, either $d(v)$ would be less than $d(w)+1$ for all residual (v, w) edges, in which case we could further increase the label of v, or $d(v)$ would be greater than $d(w) + 1$ for some residual (v, w) edge, and we would be violating the label constraint. Thus, the rules we have given for relabeling are exactly what is needed to allow further pushing.

It is worth observing that at any time after the first set of pushes, a node's label provides a lower bound on the shortest-path distance from that node to the sink in the residual graph. If $d(v) = n$ for some v, there are at least n residual edges between v and t, since $d(t) = 0$ and across each residual edge we have a drop of no more than 1 in the value of the label. We will make use of this fact later in our discussion.

What is described above is the entirety of the algorithm; we can summarize its execution as follows:

1. Set the source label $d(s) = n$, the sink label to $d(t) = 0$, and the labels on the remaining nodes to $d(v) = 0$.
2. Send out as much flow as possible from the source s, saturating its outgoing edges and placing excesses on its neighboring nodes.
3. Calculate the residual edges.
4. Relabel the active nodes, increasing values as much as possible without violating the label constraint (i.e. set the label to the minimum label of the neighboring nodes plus 1)
5. Push as much flow as possible on some admissible edge.
6. Repeat steps 4 and 5 until there are no active nodes left in the graph.

We will show in a later section that the above algorithm eventually terminates with no excesses on any nodes except the source and the sink, and that it has calculated a maximum flow.

8.5.3 An Illustrated Example

In figure 39 we show a graph where initially the labels are set to: $d(s) = n$, $d(t) = 0$ and $d(v) = 0$.

We first saturate all (s, u) edges (ie. all edges originating from the source), update the excesses on the nodes and create the residual edges (figure 40).

Next, we update the labels on the nodes in such a way that will allow us in the next step to push flow from an active node. Thus the question now becomes 'Which arcs can I push on?' To answer this question consider the following cases (all the arcs (v, w) have $d(v) \leq d(w) + 1$):

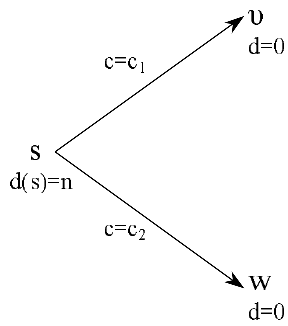


Figure 39:

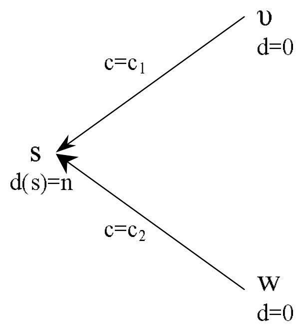


Figure 40:

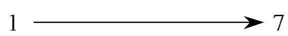


Figure 41:

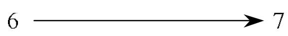


Figure 42:

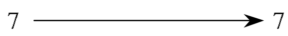


Figure 43:

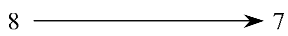


Figure 44:

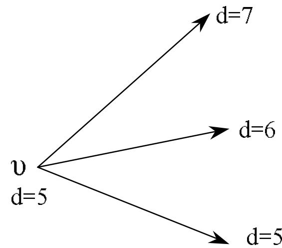


Figure 45:

Figure 41: We cannot push the arc from label 1 to label 7, since it would introduce a residual edge from 7 to 1, violating the labeling constraint.

Figure 42: A push along the arc from label 6 to label 7 will not violate label invariant. However, we do not allow this type of a push (by definition of the algorithm). The intuition is that smaller labels indicate closer distance to the sink and we are trying to push from higher to lower labels since we want to push traffic closer to the sink node.

Figure 43: A push from label 7 to label 7 will not violate labeling constraints, but allowing such a push would let the algorithm push flow back and forth along the edge repeatedly, without making any progress. Such push is disallowed by the definition of the algorithm.

Figure 44: A push along the arc 8 to 7 is allowed.

Figure 45: No push out of node v is allowed without changing its label firs. In this example, relabeling increases the label of v to 6, since it cannot go higher without violating labeling constraints.

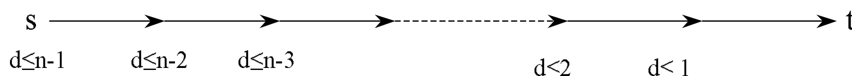
8.5.4 Analysis of the Algorithm

Liveliness We first prove that the algorithm can always perform a push or relabel step as long as there is an active node; i.e., as long as there is some node other than the source and sink with an excess. This is a liveliness property for the algorithm. Note first that an active node v must have at least an outgoing edge in the residual graph; such an edge was introduced when the excess came in. We then see that we must be able either to push from an active node or to relabel it:

- If there exist a node connected to v by an outgoing edge that has label $d(v) - 1$ then we can push.
- If the previous statement is not true, all nodes connected to v must have a label $> d(v) - 1$. In this case we can increase v's label.

Since some operation can always be performed on an active node, it is clear that the algorithm is live at all points in execution–that is, we cannot possibly reach a point prior to completion where no further action is possible. If we reach a state where we can neither push or relabel anything, there are no active nodes, so we have reached the end of the algorithm. At this time, all excesses must have been pushed either to the sink or back to the source.

Correctness of the Algorithm Why does this algorithm produce the maximum flow? To answer this question, we first prove that throughout the execution of the push-relabel algorithm, the manner in



which labels are maintained ensures that there is no augmenting flow path from the source to the sink.

Consider any path from s to t. Since any such path can be at most of length $n-1$, and we have the labeling constraint, we arrive at the conclusion that $d(s) <= n-1$. But we know that $d(s) = n$ and $d(t) = 0$. This is a contradiction, so we have no augmenting paths (figure 8.5.4).

So, throughout the execution of the algorithm, there is no augmenting path from the source to the sink. Since our algorithm only stops when all excesses are at the source or the sink, when the algorithm stops, there are no excesses in the graph and no augmenting path, i.e. we have a max flow. So to show correctness, we need to prove that the algorithm does indeed stop. We will do this by bounding the number of relabel operations and the number of pushes. To do this, we will need to draw a distinction between saturating pushes—those which fill the capacity of a residual edge—and non-saturating pushes, which do not. The latter occur in the case where there is more capacity on the edge over which we are pushing than there is excess on the node we are pushing from.

We begin by bounding the number of relabel operations.

Bounding Label Sizes We will show that the labels of nodes are bounded in the push-relabel algorithm. We begin by proving the following theorem which guarantees the existence of a residual path from an active node to s. This will be used in bounding label sizes later.

Theorem 8.11. *For any active node v, there is a simple residual path [4] from v to the source s.*

Proof: As a first attempt to prove this, one could try using induction. Assume there is a residual path from some active node w to the source and that there is a residual edge from w to v. Then when we push from w to v, v becomes active and a residual edge is created in the opposite direction; this edge attaches to the beginning of the residual path from w to the source, and so we have a path from v to the source. Unfortunately, this proof will get rather complicated since there are many special cases (e.g. what if later in the algorithm, one of the residual edges on that path from v to the source is saturated by a push?) to be considered. A more elegant proof using contradiction is the following.

Assume that there does not exist such a path from v to the source s. Define A to be the set of nodes reachable from v in E_f (the graph of non-0 residual edges). $s \notin A$. $\bar{A}$ is the rest of the nodes. By definition, since there are no deficits in the graph,

$$\forall(w \neq S), E_f(w) \geq 0)$$

where $E_f(w)$ is the excess on node w, i.e. the flow into w minus the flow out of w. This implies that

$$E_f(A) > 0, \tag{13}$$

since $v \in A$ and $E_f(v) > 0$, since it has an excess.

Consider some node x in A and some node w in $\bar{A}$. There are two possible edges (in the original graph) that go between x and w:

[4] A simple path is one in which no node is used twice; we can easily convert any non-simple path to a simple one by eliminating useless cycles.

- The edge (w, x). We will prove that the flow on this edge is 0. Suppose there is positive flow on this edge. Then there would be a residual edge from x to w. Since $x \in A$, x is reachable from v which implies that w would be reachable from v. But then w would be in A by the definition of A. This gives a contradiction. Hence, the flow from w to x on this edge must be 0: $f[(w,x)] = 0$.
- The edge (x, w). This edge must be saturated with flow, otherwise w would be reachable from x, which is impossible by the argument for the previous case. The flow from w to x on this edge is the negative of this saturation amount: $f[(w,x)] < 0$.

Therefore,

$$\forall w \in \bar{A}, x \in A : f[(w,x)] \leq 0. \tag{14}$$

By the definition of E_f,

$$E_f(A) = \sum_{x \in A} E_f(x) = \sum_{x \in A} \sum_{(w,x)} f[(w,x)].$$

When w and x are both in A, the contribution of the edge between them to the summation will cancel, since in one term it will be positive, and in the other it will be negative. Therefore, we only need to consider the contribution of edges (w, x) where $x \in A$ and $w \in \bar{A}$. Considering only those flows and substituting inequality 14 gives:

$$E_f(A) = \sum_{x \in A, w \in \bar{A}} f[(w,x)] \leq 0.$$

This contradicts inequality 13. Therefore, our assumption was invalid and there must be a residual path from v to S.

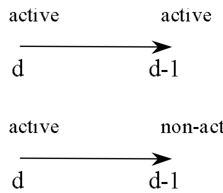
Aside: Does A include the sink? It may include the sink, but it does not matter for this proof, since $\sum E_f \geq 0$ even in this case. Excesses are not really "cancelled" at the sink; instead, they are stockpiled there and a large stockpile at the sink is good. We will now use this theorem to prove that label sizes are bounded.

Theorem 8.12. *The label of a node is at most 2n.*

Proof: Consider an active node v with label d. By theorem 8.11, there is a simple path in the residual graph from v to S. Along a residual path, the label between successive nodes can decrease by at most 1, from the label constraint. If the path from v to S has length k, then $d_S \geq d - k$. Since $d_s = n$ and $k \leq n$, $n \geq d - n$. This implies that $d \leq 2n$. The same limit holds for inactive nodes since they were active when they got relabelled.

This label limit gives a bound on the number of relabel operations that can occur. There are $O(n)$ nodes, each with at most $2n$ relabels, giving a bound of $O(n^2)$. ∎

Bounding the number of saturating pushes Consider an edge in the graph. Imagine a series of saturating pushes back and forth across this edge. The first saturating push sends from label d to label $d-1$. After this, the second push requires a relabel of $d-1$ to $d+1$ to enable the reverse push. Each push after the first requires a relabeling of one of the nodes. Theorem 8.12 implies that the label can reach at most $2n$, so there at most $O(n)$ saturating pushes per edge. This implies that there are at most $O(nm)$ saturating pushes for the algorithm.



Bounding the number of non-saturating pushes Let us first convince ourselves that the above analysis for saturating pushes will not work for bounding the number of non-saturating pushes. The reason is that we can keep sending little non-saturating pushes across an edge without raising the label (if more excess arrives at the from-node). But, it seems that the algorithm is making progress, since intuitively, it is pushing the excess towards the sink (lower label).

In order to prove this formally, we will need the notion of a *potential function.* Potential functions are a useful technique for analyzing algorithms in which progress is being made towards the optimal solution. We will define a potential function Φ which describes how far the current state is from the optimal. Then we will analyze the contribution of each operation in the algorithm to the potential function. There are many potential functions one could choose for this algorithm; we will choose a rather simple one which nevertheless allows us to prove a bound on the number of non-saturating pushes.

Define the potential function to be the sum of all labels of active nodes:

$$\Phi = \sum_{v \ active} d(v)$$

This potential function cannot be negative since all labels are non-negative. We will show that relabels and saturating pushes make a finite total positive contribution to the potential function. Non-saturating pushes will have a negative contribution to Φ associated with each push. Therefore, after some number of non-saturating pushes, Φ will become negative; since it cannot be negative, the algorithm must stop before performing that many non-saturating pushes. Let us analyze $\Delta\Phi$, the change in Φ due to the various operations that the algorithm can perform.

- **Relabels:** For a single relabel step, $\Delta\Phi =$ amount of relabeling. The total amount of relabeling per node is $2n$. Therefore, the maximum contribution to Φ from relabeling all nodes is $\leq n * 2n$, which is
$$\Delta\Phi(all\ relabels) \leq O(n^2).$$
- **Saturating pushes:** A saturating push from v to w may potentially add w to the list of active nodes, if it was previously inactive (if v gets inactive after the push, we will treat it as a non-saturating push for the purposes of our analysis). Therefore, a saturating push may increase the potential function by as much as $2n$ (the max value for a label). There are at most $O(nm)$ saturating pushes, from 8.5.4. Therefore, the total change due to saturating pushes is
$$\Delta\Phi(all\ sat.\ pushes) \leq O(n^2 m).$$
- **Non-saturating pushes:** There are two kinds of non-saturating pushes: active-to-active and active-to-inactive. See figures 8.5.4 and 8.5.4.
  - In the first case, the node with label d becomes inactive (all of its excess was pushed); the other node changes neither activity nor label. Thus $\Delta\Phi = -d$

– In the second case, the node with label d becomes inactive and the node with label $d-1$ changes from inactive to active. Thus $\Delta\Phi = -d + (d-1) = -1$

Therefore, for each non-saturating push, $\Delta\Phi(one\ non-sat.\ push) \leq -1$.

We have found that saturating pushes and relabels contribute $O(n^2m)$ total to the potential function. Each non-saturating push lowers the potential function by at least 1. Therefore, since potential function does not become negative, there can be no more than $O(n^2m)$ total non-saturating pushes.

Running time analysis Since the number of pushes and relabels that the algorithm performs is bounded, the algorithm does indeed stop. From the discussion in 8.5.4, this proves that the algorithm is correct, i.e. it does produce a max flow. Our bounds on the number of operations also allow us to bound the running time of the algorithm. The number of relabels and saturating pushes are dominated by the number of non-saturating pushes, $O(n^2m)$. So, we can consider the number of steps to be $O(n^2m)$. Each step takes $O(n)$ time, since we need to search the entire graph of n nodes for an active node, and then search its neighbors (at most n) for a place to which to push or to determine the value of the new label. The total running time is $O(n^3m)$. We can improve though the running time of the push/relabel algorithm using a better implementation.

8.5.5 A better implementation: Discharge/Relabel

We mentioned earlier that the running time obtained for the naive implementation of push-relabel can be improved by using different rules for ordering the push and relabel operations. We describe one such ordering rule; the resulting algorithm is called discharge/relabel. Recall that the push-relabel algorithm performed $O(n^2m)$ operations, and the time per operation was $O(n)$. The bottleneck in the analysis was the analysis of non-saturating pushes: there were at most $O(n^2m)$ of them, and each took $O(n)$ time.

In fact, we can reduce the time per operation as follows. First, we can easily maintain a list of active nodes, eliminating the search for an available active node. Second, we can relabel and do lots of pushes from a single active node; after those pushes, either there is an excess, in which case relabeling this node is possible, or there is no excess, in which case we just move on to the next active node in our list. Note that the push operation may create newly active nodes. All such nodes are added to the list of active nodes. The two actions performed by the algorithm are relabeling and discharging. A discharge takes a node and keeps pushing flow out of it until no more pushes are possible. At this point, either the node has no excess, or relabeling the node is possible. The discharge/relabel algorithm can be implemented so that the running time is $O(n^2m)$; an improvement of a factor of n over the running time of the naive algorithm. However we have to be a little careful in the analysis in order to claim this bound.

Running Time of Discharge/Relabel We will use the discharge/relabel algorithm as an example to demonstrate the method of analyzing the time complexity of a non-trivial algorithm. For simple iterative algorithms, we are accustomed to calculating complexity by breaking the algorithm down into its iterative steps, phases, and so on, then multiplying the number of steps by the complexity of each. However, algorithms such as discharge/relabel do not have as clear of an iterative structure, so a similar analysis may be impossible or uninformative. Instead, we will break algorithms like this down into the different kinds of "work" that they do, and we will analyze the total running time of each type of work.

The actions of the discharge/relabel algorithm can be broken down into the following types of work:

- Relabel

- Saturating push
- Nonsaturating push
- Choosing the next admissible edge
- Finding the next active node

For each type of work, we will find the per-operation complexity, then find the total complexity for that type of work.

Relabel Consider a relabel operation. When we relabel a node, we must examine all of the edges connected to it to find the new label. Therefore, the relabel step uses time proportional to the degree of the node. The total work is therefore the sum over all nodes of the product of the degree of the node and the number of times it can be relabelled, which is $O(n)$. The degree summed over all nodes is proportional to the total number of edges, or $O(m)$. The total work is thus $O(mn)$:

$$\sum_{v} \text{degree}(v) O(n) = O(mn) \tag{15}$$

Note, however, that this expression takes into account only operations that actually result in a change of some node label, not work done to see if a node label can or must be changed. This will be discussed further below.

Saturating Push Next, consider a saturating push. Such a push can be done in constant time, given that we know which nodes and edge will be involved in the push. This is valid because we've pulled out the work of finding the next admissible edge (to be analyzed below). The total number of saturating pushes will be $O(mn)$ because each of the m edges participates in at most $O(n)$ saturating pushes. The total complexity of saturating pushes is thus $O(mn)$.

Non-saturating Push Now, we can analyze the complexity of non-saturating pushes. Given that we know which nodes and edge participate in the push, we only need constant time. It was shown in 8.5.4 that the number of non-saturating pushes is $O(n^2m)$, so this is the total complexity of non-saturating pushes.

Next Admissible Edge and Next Active Node The issue which has been sidestepped until now is this: how do we keep track of useful edges so that we can actually do the pushes in $O(1)$ time and so that we do not waste work checking to see if nodes need to be relabelled?

The answer is to maintain a linked list of admissible edges (ones we are able to push on) for each node. When we need to do a push, we simply take the next edge on the list, remove it from the list, and check whether it is still admissible (the other end of this edge might have been relabelled since we put this edge into the list). If the edge is still admissible, we push on it. If it is not admissible, we go to the next edge on the list. This takes $O(1)$ time per edge.

What about the time to build the list? It can be lumped into the time for a relabel operation. Observe that if we exhaust the list of useful edges, we can be assured that a relabel operation will be successful (*i.e.*, that it will result in a change of the node label). Thus, when we exhaust a list, we relabel

and build a new list at the same time, since both involve examining the edges from the current node. This also ensures that we do not waste work examining edges to see if a node needs to be relabelled without actually relabeling it.

Going back to our complexity analysis of different work types, we can see that finding the next admissible edge takes $O(1)$ time if we just move down the linked list. Finding the next active node can be done in $O(1)$ time if we maintain a linked list of active nodes. Since we execute either a relabel or a push when we find an active node, the work of finding the node can be lumped together with these operations.

Total Complexity Table 1 summarizes the per-operation and total complexity of each type of work. Finally, we see that the complexity of the whole algorithm is $O(n^2m)$. This example is instructive because non-trivial algorithms will generally require the use of this method for determining time complexity. We have also seen that going through this analysis suggests how to set up data structures in order to obtain the claimed running time for the algorithm.

Table 1:

`Type of Work`	`Per Operation`	`Total`
`Relabel`	$O(degree)$	$O(mn)$
`Saturating Push`	$O(1)$	$O(mn)$
`Non-saturating Push`	$O(1)$	$O(n^2m)$
`Next Admissible Edge`	$O(1)$	
`Next Active Node`	$O(1)$	

Can We Do Better? With data structures called dynamic trees (which are beyond the scope of this class), the complexity can be reduced to $O(nm\log(n^2/m))$ (see the original Goldberg-Tarjan paper [6]). Also, in the special case that the graph has unit capacities everywhere, all pushes are saturating pushes and discharge/relabel runs in $O(mn)$ time. Note that Ford-Fulkerson also runs in $O(mn)$ time on such a graph because the flow can be at most m and there can be at most n augmenting paths.

8.6 Flow with Lower Bounds

8.6.1 Motivating example - "Job Assignment" problem

The fact that given integer capacities one can always find integral max-flow can be used to apply max-flow formulation to a variety of optimization problems. For example, consider the problem of assigning jobs to people. Each person can perform some subset of the jobs and cannot be assigned more than a certain maximum number of jobs. The objective is to find a valid assignment of jobs to people that maximizes the number of assigned jobs. Note that the statement of the problem clearly implies that we would like to get an integral solution.

We can pose this as a maximum flow problem. Suppose the jobs are $\{J_1, J_2, \cdots, J_m\}$, and the available people are $\{P_1, P_2, \cdots, P_n\}$. Corresponding to each person P_j is the value x_j, the maximum number of jobs that the person can handle.

We construct a bipartite graph with a vertex corresponding to each job J_i on the left side, and a vertex corresponding to each person P_i on the right side. If the job J_i can be assigned to person P_j, we add an edge J_iP_j with capacity 1.

Now we add two more vertices to our bipartite graph. First, we add a source s and for each $i \in \{1 \cdots m\}$ connect it to J_i with edge capacity 1. Then we add a sink t and for each $j \in \{1 \cdots m\}$ connect P_j to t with edge capacity x_j. (See Figure 46).

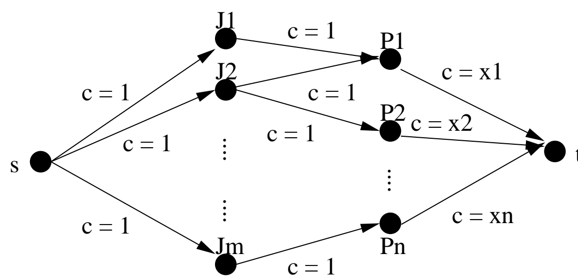


Figure 46: Transforming job assignment into flow.

Observe that an integral flow in this graph corresponds to a valid assignment of jobs to people. Each edge J_iP_j with flow of 1 corresponds to the assignment of job J_i to person P_j. The value of the flow corresponds to the number of assigned jobs. Since all edge capacities are integral, the maximum flow is integral as well.

Note that if each person can handle at most 1 job, then this problem is equivalent to the maximum matching problem in the subgraph induced by $V - \{s, t\}$ (the original bipartite graph of jobs and people).

Let us slightly generalize the problem. Suppose that in addition to the specification of the simple job assignment problem, we add the following conditions:

1. certain jobs must be assigned;
2. certain people must be assigned some minimum number of jobs.

These conditions lead to a new constraint in the corresponding graph formulation; now we also have a **lower bound constraint** on certain edges, i.e. the flow on those edges must be at least a certain minimum amount.

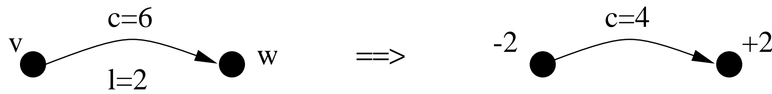


Figure 47: Lower Bounds To Excesses And Deficits.

For each job J_i that must be assigned, we add the constraint that the flow on sJ_i be at least 1. For each person P_j that must be assigned a minimum number of jobs we add the constraint that the flow on edge P_jt must be at least the minimum number of jobs that person must be assigned.

8.6.2 Flows With Lower Bound Constraints

The job assignment example leads us to consider the more general problem of flows with both upper and lower bounds on edges.

Suppose we are given a graph G with capacity $c(ij)$ and lower bound flow $l(ij)$ for each edge ij. Now the flow $f(ij)$ through the edge ij must satisfy $l(ij) \le f(ij) \le c(ij)$ for all edges ij.

We will show that this problem can be reduced to max flow.

Up until now we worked with flows that satisfied conservation constraints. In order to solve flow problems with lower bounds on the capacities we will have to work with flows that violate conservation constraints (while satisfying capacity constraints).

Given a max flow problem with lower bound constraints, we solve it in three steps. We will discuss each one of these steps in detail in the subsequent sections.

1. Translate the lower bound constraints into excesses and deficits: for each edge ij, we set $f(ij) = l(ij)$ and update the capacity by setting $c(ij) = c(ij) - l(ij)$. Do not add the corresponding residual edge.
2. Add auxiliary s' and t' nodes. For every v with excess add an edge $s'v$ with capacity equal to the excess. For each v with a deficit add vt' with capacity equal to that deficit.

   Find the max flow from s' tot' in the resulting graph.
3. Compute max flow between the original s and t in the residual graph obtained at step 2 after deleting s', t', and all edges adjacent to them.

Translating Lower Bound Constraints Into Excesses And Deficits: In the first step of the algorithm we set the flow on each edge uv to be equal to the lower bound on this edge, $l(uv)$. This adds a (positive) excess to v and (negative) deficit to u. We also update the capacity of uv, reducing it by $l(uv)$. Notice that we do not add a backwards residual edge because no flow can be pushed back without violating the lower bound. An example shows the transformation.

Removing Excesses And Deficits: Denote the flow vector that we got in Step 1 by f_1. The goal of the second step is to compute flow f_2 such that $f_1 + f_2$ satisfies both capacity (including lower bounds) and conservation constraints. Observe that, for any flow vector f that satisfies the residual capacities computed in Step 1, $f_1 + f$ will satisfy the capacity constraints of the original problem. The challenge is to find f_2 such that $f_1 + f_2$ will satisfy the conservation constraints as well.

We add an auxiliary source s' and sink t'. Connect s' to all nodes with excess by edges with capacity equal to the excess, and connect t' to all nodes with deficit with capacity equal to the deficit. We also connect s to t and t to s with infinite capacity edges. (The importance of these edges will be seen later). Now we compute max flow from s' to t' in the resulting graph.

Let $\tilde{f}$ be the result of the above max flow calculation. We claim that:

- $\tilde{f}$ saturates all the edges outgoing from s' and all the edges incoming to t' if and only if there is a solution to the original problem.
- In this case, f_2 can be obtained by considering the coordinates of $\tilde{f}$ that correspond to the original graph edges.

First, assume that we found $\tilde{f}$ that saturates all the edges adjacent to s' and t'. Consider f_2 that is obtained by restricting $\tilde{f}$ to the edges in the original graph. If node v has excess $Ex(v)$ then there is an edge $s'v$ with $\tilde{f}(s'v) = Ex(v)$. Since v has an excess there is no vt' edge. Consider the conservation constraint at v in f_2. From the point of view of v, the only difference between $\tilde{f}$ and f_2 is that the edge $s'v$ (together with its flow) disappeared. Hence, the incoming minus the outgoing is equal to $-Ex(v)$. But the same difference for f_1 was $Ex(v)$, which means that $f_1 + f_2$ will satisfy conservation constraints. A similar argument works for nodes with deficit in f_1.

Now we need to show that existence of a feasible solution to the original problem implies that $\tilde{f}$ will saturate all the edges adjacent to s' and t'. Let f^* be a feasible solution to the original problem. Observe that $f^* - l$ is feasible with respect to residual capacity constraints computed in Step 1. Consider the difference between incoming and outgoing flow in $f^* - l$ for some node v:

$$\begin{aligned}\sum_{uv\in E}(f^*(uv) - l(uv)) - \sum_{vu\in E}(f^*(vu) - l(vu)) &= \sum_{uv\in E} f^*(uv) - \sum_{vu\in E} f^*(vu) - \sum_{uv\in E} l(uv) + \sum_{vu\in E} l(vu)\\ &= -\sum_{uv\in E} l(uv) + \sum_{vu\in E} l(vu)\end{aligned}$$

But the last expression is exactly equal to $-Ex_{f_1}(v) = -c(s'v)$. In other words, if v has an excess, then $f^* - l$ has a deficit of exactly the same value at v. Now extend $f^* - l$ by saturating all the edges adjacent to s' and t'. This process cancels all the excesses and deficits giving a legal flow in G'. This flow is maximum, since the cut around s' (and around t') is saturated. Hence, we proved that there exists a flow that saturates these cuts, which means that the flow $\tilde{f}$ that we will find will saturate these cuts as well.

Why did we add the infinite capacity edges connecting original source and sink (s and t) ? Figure 48 shows an example where there is no $\tilde{f}$ that saturates all the edges adjacent to t' and s'. At the same time, it is easy to see that a feasible flow that satisfies the lower bound exists. The reason is that we did not add the infinite capacity edges in this example. Adding these edges solves the problem, as we can see from Figure 49. It is an interesting exercise to go over the above proof and try to find the place where we have used existence of these edges.

Max Flow On Residual Graph: The result of the first two steps is a flow f_1+f_2 that satisfies capacity and conservation constraints. Let f^* be some $s-t$ max flow that satisfies capacity and conservation constraints. It is easy to see that $f^* - f_1 - f_2$ is a feasible flow in the residual graph with capacities equal to $c(uv) - f_1(uv) - f_2(uv)$ for $uv \in E$ and $f_1(uv) + f_2(uv) - l(uv)$ for $vu \in E$. Thus, as the last step of the algorithm, we compute these residual capacities and compute flow f_3, which is max $s-t$ flow in this residual graph. The answer to the original problem is $f_1 + f_2 + f_3$.

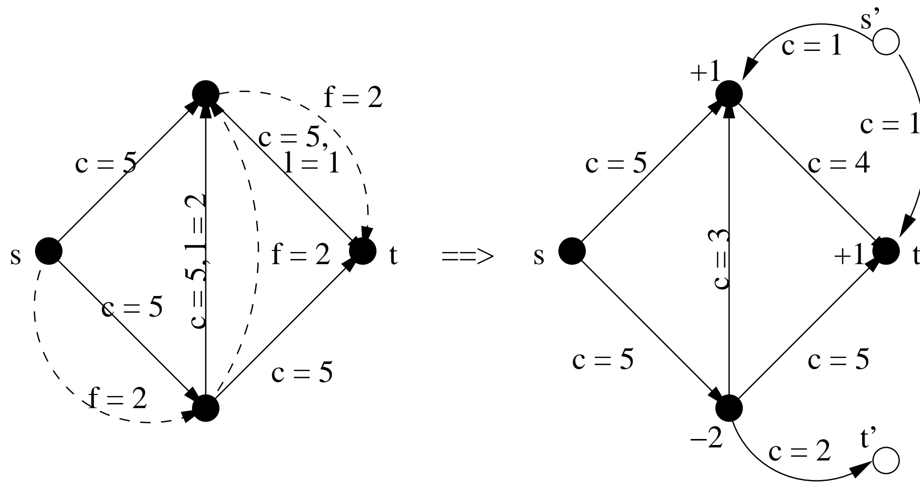


Figure 48: The original graph with two edges that have lower bound constraints has a solution (a legal flow) as shown by dashed lines. However the transformed graph cannot remove all excesses and deficits by finding a flow from s' to t'.

Effects of Lower Bound Constraints Introducing lower bounds can change the properties of the max flow algorithm in somewhat unexpected ways. Previously, there always existed a feasible zero flow solution (all edges carry no flow). With lower bound constraints, the zero flow may not be feasible. In fact, in certain instances, the flow may have a negative maximum value (that is, the net flow is from sink to source rather than vice versa), a situation which would not occur with the original max flow algorithm, under the assumption that no capacity can be negative.

Consider the following example (see Figure 51). We have a graph where the forward capacities from s to t are small relative to the backward capacities from t to s, and one of the backward edges has a large lower bound constraint. When we remove excesses and deficits, we introduce a large flow from t back to s. Thus, when we begin calculating the maximum flow with the residual graph, we are starting with a large negative (but feasible) flow that no forward flow could possibly cancel. The value of the maximum $s - t$ flow in this example is -23, so there is a net flow backwards from t to s.

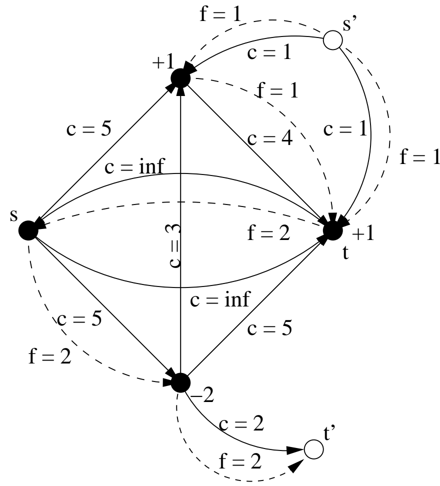


Figure 49: Adding edges from original source to sink and back allows us to find a legal flow which removes the excesses and deficits. The solution is shown by dashed lines.

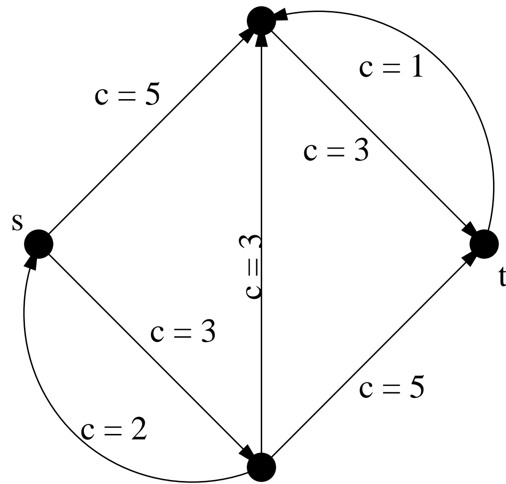


Figure 50: In the residual graph, we do not add residual flows against the lower bound flow, as pushing flow back over such edges would violate the lower bound constraints in the original graph.

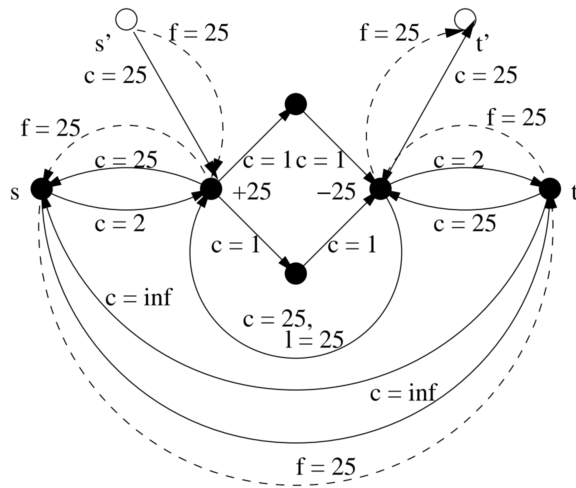


Figure 51: After removing the excesses and deficits from this graph, we are left with a large negative flow from the sink to the source, an impossible outcome with the original max flow algorithm. The flow is shown with dashed lines.

9 Matchings, Flows, Cuts and Covers in Bipartite Graphs

This section will explore the relationships between many graph-related problems such as Matching, Max flow, Min cut, and Min node cover in a special type of graph - bipartite graphs. In particular we see that computing the maximum matching in an undirected bipartite graph reduces to computing the maximum flow on another directed graph that is constructed from the original graph. We also introduce two variants of max flow algorithms on such graphs that take advantage of the peculiar structure of these graphs and are an order improvement over the general flow algorithms on such graphs.

9.1 Matching and Bipartite Graphs

Bipartite Graph

A bipartite graph is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between two sets V_1 and V_2 Figure 52 shows an example of a bipartite graph. A bipartite graph can also be defined as one which can be colored with two colors. In fact, the following three properties are equivalent for any undirected graph:

1. G is bipartite
2. G is 2-colorable
3. G has no cycles of odd length

It is easy to see that the first and the second properties are in fact restatements of each other.

There is simple method to decide whether a graph is bipartite. We do a BFS of the graph starting at an arbitrary node. We include nodes on odd levels in one vertex partition and nodes on the even levels in the other. If at any stage in the traversal, we find that for a previously visited node, a different set than the original assignment is specified by the traversal, then the graph is not bipartite. (See Figure 53.) That is, the graph is bipartite if and only if there are no edges between two nodes at odd levels or between two nodes at even levels.

It is interesting to note that any tree, is bipartite, as it has no cycles.(See Figure 54.)

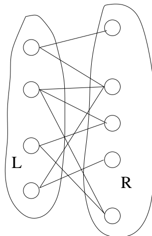


Figure 52: An example of bipartite graph, with the bipartition clearly shown

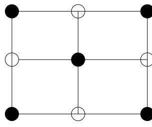


Figure 53: Another graph that is bipartite(But not obviously so!)

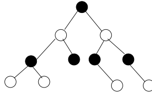


Figure 54: A tree is a bipartite graph

Maximum Matching

$G = (V, E)$ is an undirected graph. $M \subseteq E$ is a matching if for all vertices $v \in V$, there exist at most one edge $(x, y) \in M$ such that $x = v$ or $y = v$. If such an edge exists then we say that the vertex v is matched by the matching M; otherwise, v is unmatched. Informally, a matching is a set of edges such that no two edges share a common node. Figure 55 shows a matching on a bipartite graph (edges included in the matching are the thick lines).

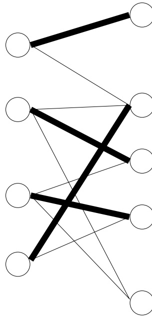


Figure 55: An example of matching in a bipartite graph

A maximum matching is a matching of maximum cardinality, that is, a matching M^* such that for any matching M, we have $|M^*| \geq |M|$. A perfect matching is one in which all nodes in the graph are matched. Clearly, a perfect matching is not always possible (*e.g.*, a graph with an odd number of nodes).If we had a black box to compute a maximum matching then we could use it to check whether a perfect matching exists. All we need to check is if the maximum matching has $|V|/2$ edges.

For future reference, we define a minimum cost matching for graphs in which edges are associated with costs. The minimum cost matching is the matching that minimizes the sum of the costs on chosen edges. The costs may be negative. So, in the special case where all costs are -1, the problem is equivalent to finding a maximum matching.

9.2 Finding Maximum Matching in a Bipartite Graph

In this section, we shall focus on finding a maximum matching in a bipartite graph. In general, one can find maximum matching in a general graph in polynomial time, but the algorithm is quite complicated

and is out of scope. We will show that for the restricted case of a bipartite graph, finding maximum matching can be reduced to a flow problem. Moreover, we will show that due to the special structure of the resulting network, the flow algorithms discussed earlier can be adjusted to run much faster than it takes to compute general maximum flow.

Many practical problems reduce to the problem of finding a maximum matching in a bipartite graph. As an example, one might consider assigning a list of available jobs to a list of people, a list of tasks to a list of machines, or a list of opening positions to a list of applicants. A maximum matching provides a maximum number of assigned jobs, tasks, or opening positions. In general, any one-one assignment from a set of one type to a set of another type often leads to a matching problem in a bipartite graph.

To solve the matching problem, we first try to relate this problem to a maximum-flow problem. The approach is to construct a flow network in which flows correspond to matchings as shown in Figure 56. Given a bipartite graph of the original problem $G = (V, E)$ with two partitions L and R, $V = L \cup R$. we can construct a flow network $G' = (V', E')$ as follows.

1. Add two new vertices s and t to V, i.e. set $V' = V \cup \{s, t\}$.
2. Directed edges of G' are given by $E' = \{(s, u) : u \in L\} \cup \{(u, v) \in E : u \in L, v \in R\} \cup \{(v, t) : v \in R\}$
3. Assign unit capacity to all edges incident on s and t and infinite capacity to the remaining edges.

Thus we have constructed a valid instance of a flow problem. We claim the following property:

Lemma 9.1. *The value of the maximum $s - t$ flow in G' is the value of the maximum cardinality matching in G.*

Proof: In order to prove the claim, we need to show that the following two properties are true:

1. If there exists a maximum matching M^* in G, then the maximum $s - t$ flow $|f^*| \geq |M^*|$
2. If there exists a maximum $s - t$ flow f^* in G', then the maximum matching M^* in G is such that $|M^*| \geq |f^*|$

Property (1) can be shown by assigning a unit flow to every edge in M^* and zero flow to the remaining edges: $\forall (u, v) \in E'$ if $(u, v) \in M^*$ then $f(s, u) = f(u, v) = f(v, t) = 1$ otherwise $f(u, v) = 0$. By construction, vector f is a legal flow and the net flow across the cut around source $(s, V' - s)$ is equal to $|M^*|$. Therefore, the the maximum $s - t$ flow f^* in G' satisfies $|f^*| \geq |f| = |M^*|$.

To prove property (2), we make use of the fact that the existence of a general maximum flow f^* implies the existence of an integer-valued flow, say $f^{*'}$ in G' such that $|f^{*'}| = |f^*|$. (Given a fractional flow, how hard is it to transform this flow into an integer flow in this context?) Observe that there are no "flow cycles" since all edges are directed "left to right" in our construction. Any node in L has incoming capacity of only 1 unit and hence the value of our integral flow $f^{*'}$ on any edge between L and R can be either 1 or 0. Thus, the decomposition of $f^{*'}$ into flow paths and flow cycles consists only of paths where each path has 3 edges and brings exactly 1 unit of flow from s to t. The number of paths in the decomposition is equal to the value of our flow $|f^{*'}|$.

For each path (s, u, v, t) in the decomposition, put the edge uv into matching M. By the discussion above, this is a legal matching. Moreover, the size of this matching satisfies $|M| = |f^{*'}|$. Thus, the size of the maximum cardinality matching satisfies $|M^*| \geq |M| = |f^{*'}|$. ▮

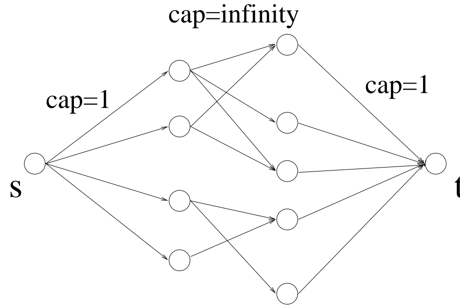


Figure 56: An instance of a flow network arising from the matching problem G'

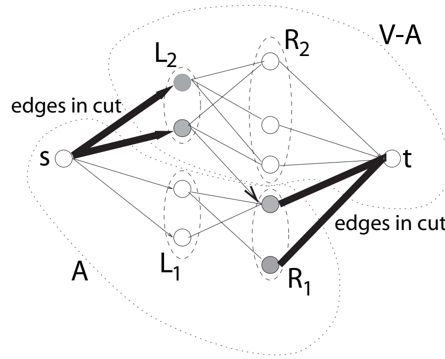


Figure 57: The min-cut described as a node partition. Thick edges are edges crossing the cut in the "right" direction.

9.3 Equivalence Between Min Cut and Minimum Vertex Cover

We already know that the value of the minimum cut is equal to the value of the maximum flow. Also we have seen that the value of the matching in an undirected graph is equal to the value of the maximum flow in a directed graph derived from the original graph. The goal of this section is to prove the following theorem:

Theorem 9.2. *Cardinality of minimum vertex cover in a bipartite graph is equal to the cardinality of maximum matching in this graph.*

Proof: Consider a minimum s-t cut $(A, V - A)$ in the flow graph formed from the bipartite graph (see Figure 57), where $s \in A$ and $t \in V - A$. The capacity of the cut around s is bounded by n. In fact, it is bounded by the size of the left side of the bipartite graph, which is smaller than n, but it is not important for this discussion. Thus, the capacity of the minimum $s - t$ cut $(A, V - A)$ is at most n, which means that no infinite capacity edges can cross the cut from A to $V - A$.

Denote $L_1 = A \cap L$, $R_1 = A \cap R$, $L_2 = L - L_1$, and $R_2 = R - R_1$. Consider the edges that cross the cut $(A, V - A) = (\{s\} \cup L_1 \cup R_1, \{t\} \cup L_2 \cup R_2)$. There are several cases to consider.

- Edges from L_1 to L_2 and from R_1 to R_2: no such edges because the original graph is bipartite.
- Edges from R_1 or R_2 to L_1 or L_2: no such edges in the original graph, since all edges were directed from L to R.

- Edges from L_1 or L_2 to R_1 or R_2: no such edges in the cut since they have infinite capacity and we have already proved that the capacity of the cut is limited by n.
- Edges from s to R_2: no such edges in our construction at all.
- Edge from s to t: no such edge in our construction at all.
- Edges from L_1 to t: no such edges in our construction at all.

Thus, the only remaining possible edges are from s to L_2, and from R_1 to t. Define set S as follows: for every edge su where $u \in L_2$, add u to set S. Also, for every edge vt where $v \in R_1$, add v to S. Observe that cardinality of S is equal to the capacity of the min cut.

We claim that S is a node cover in the original graph. Assume that this is not true. This means that there exists an edge uv, $u \in L, v \in R$ and neither u nor v is in S. By construction, this means that edges su and vt are not crossing the cut. Since edge uv can not cross the cut (it has infinite capacity in our construction), this implies that we have a path from s to t is the graph where none of the edges of this path are crossing the cut. This is clearly a contradiction, i.e. the set S is indeed a node cover.

By Lemma 9.1, capacity of the min cut in our graph is equal to the cardinality of maximum matching in the original bipartite graph. We showed how to construct a node cover S where the size of S is equal to the capacity of the cut. Thus, $|S| = |M^*|$. It is easy to see that cardinality of any node cover is bounded from below by the cardinality of any matching. (Note that, for any vertex cover, at least one of the end vertices of each edge in the matching has to be in the cover.) In particular, if we denote the minimum node cover by S^*, this means that $|M^*| \leq |S^*|$. Putting this all together, we have

$$|M^*| \leq |S^*| \leq |S| = |M^*|.$$

This means that the node cover S that we have constructed is indeed a minimum cardinality node cover. ∎

Although not needed for the proof of the theorem, it is interesting to note that any node cover can be translated into a cut in our constructed graph. In particular, minimum node cover S^* can be translated into a minimum cut which, in turn, can be translated into matching. The cardinality of this matching will be maximum by the above theorem.

Define $L_2 = S^* \cap L$ and $L_1 = L - L_2$. Similarly, define $R_1 = S^* \cap R$ and $R_2 = R - R_1$. Consider the cut $(\{s\} \cup L_1 \cup R_1, \{t\} \cup L_2 \cup R_2)$. In order to compute the capacity of this cut, we consider all types of edges that can participate in it:

- Edges s to L_2: such edges can cross the cut. Each such edge contributes 1 to the capacity of the cut.
- Edges R_1 to t: such edges can cross the cut. Each such edge contributes 1 to the capacity of the cut.
- Edges s to R_2: do not exist in the graph.
- Edge s to t: does not exist in the graph.
- Edges L_1 to L_2: do not exist in the graph.
- Edges R_1 to R_2: do not exist in the graph.
- Edges R_1 to L_2: do not exist, all edges directed from L to R.

- Edges L_1 to R_2: such edge *will not be covered by nodes in* S^* and thus, assuming S^* is a node cover, these can not exist.

We see that each node in L_2 and each node in R_1 contributes one unit to the capacity of the cut. By construction, $L_2 \cup R_1 = S$, which proves the claim. This is definitely the min-cut, if it were not then S^* cannot be the minimum node cover.

9.4 A faster algorithm for Bipartite Matching

Above, we talked about how to find a maximum matching in a bipartite graph by transforming it into a max-flow problem. Although general max-flow algorithms can then be used, we will exploit some of the properties of graphs arising from the construction to develop a faster algorithm for the problem.

The properties of such graphs are:

1. Max flow $\leq n$ ($\leq n$ unit-capacity edges leave the source).
2. There are no cycles in these graphs.
3. Length of all flow paths is equal to 3.
4. Unit-capacity edges (those from the source and to the sink).

Using regular Ford-Fulkerson, we can solve the problem in $O(nm)$ time using just property 1. Similarly, using the layered network algorithm, we can also achieve $O(nm)$, from property 1. Now, we will combine these algorithms to improve the running time to $O(m\sqrt{n})$, which is the best worst-case complexity known for the problem.

We use the combination of ideas that we studied earlier, like layered networks and push-relabel algorithms increase the efficiency of the max-flow finding procedure. We now demonstrate two algorithms, one that is a combination of Ford-Fulkerson and the layered network algorithm, and the second, a practically faster algorithm than the first, that is combination of the Ford-Fulkerson and the push-relabel algorithm.

Here is an intuitive basis for the first algorithm: from the above properties, we know that we will augment along at most n paths, so we wish to minimize the cost of finding each path. Notice that the layered network algorithm provides the most benefit when we can augment by a large amount without rebuilding the layered network, essentially reducing the overhead. When we get to a point where we do not succeed in finding many augmenting paths per each network rebuild, we are wasting time. From here on, Ford-Fulkerson will do very well if we can bound the amount by which the flow can increase. The first phase does just that. The key property that we will use in these proofs, is the fact that flow paths in such graphs are node-disjoint.

Algorithm 1

1. Repeatedly build the layered network and augment until the network has at least k layers. Observe that at this point the length of any augmenting path is at least k.
2. Run basic Ford-Fulkerson to find the remaining augmenting paths.

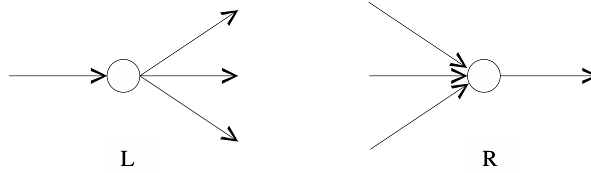


Figure 58: Nodes before any augmentation.

We will analyze the amount of work in each part of the above algorithm separately. Since we rebuild the layered network at most k times, the work spent on rebuilds is bounded by $O(km)$. Each augmentation can be done in $O(m)$ and there are at most $O(m)$ augmentations per each layered network. This gives $O(m^2 + km)$ for the first part.

The above bound is not tight. Recall that when we first covered the layered network algorithm, we made the following optimization: when we are doing a depth-first search back through the layers to find an augmenting path, if we ever get stuck and have to backtrack, we mark that edge as bad and don't look at it again until we next rebuild the network. If we apply the same optimization here, we can separate out the running time by the different computations we do: $O(km)$ building the network, $O(km)$ looking at bad edges, and $O(kn)$ looking at edges that we use (at most n paths each of length at most k), giving a total running time of $O(km)$ for this stage of the algorithm.

Now, we have a flow f such that the residual graph has no paths shorter than k from the source to the sink. We wish to find an upper bound on the running time for the regular Ford-Fulkerson algorithm to continue from where the layered algorithm left off, and find the final max flow.

Consider the flow difference between the max flow f^* and the current flow f. Their difference $f^* - f$ is a feasible flow in the residual graph of f. Decompose this difference into flow paths and cycles. Since each flow path has a value of at least 1, the number of flow paths can be at most $|f^* - f|$.

Notice that each node in the residual graph (except the source and the sink) has only one incoming or outgoing edge. This is true at the beginning when we connect the source to each node in the set L, and each node in R to the sink (Figure 58). This property is maintained in augmentation, since augmenting through a node in the unit-capacity case flips one incoming edge and one outgoing edge, leaving the number of incoming and outgoing edges unchanged (Figure 59).

We are going to show that any two flow paths in $f^* - f$ cannot share a node. First, since capacity remains unchanged throughout augmentation and we are working in a unit-capacity case, two flow paths cannot share an edge. Otherwise, the capacity constraint will be violated. Now, notice that each flow path through a node takes up one incoming edge and one outgoing edge of the node. Because flow paths do not share edges, a node needs to have at least p incoming edges and p outgoing edges in order to participate in p flow paths. But we know that each node in our residual graph has only one incoming or outgoing edge. Therefore, each node can only be on one flow path.

Thus, since each flow path contains at least k nodes (after we've run the layered network algorithm), there will be at most $\frac{n}{k}$ flow paths left because of the node disjointness of paths. In other words, we can bound the value of the remaining flow after the layered network-based phase by $\frac{n}{k}$. The time required to find each augmenting path by the Ford-Fulkerson algorithm is $O(m)$ (DFS), so the FF stage of the algorithm takes $O(\frac{n}{k}m)$.

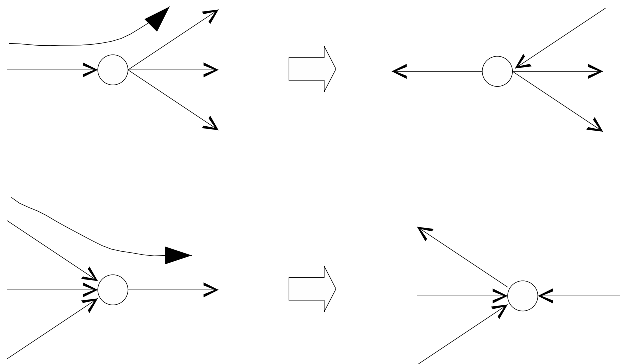


Figure 59: In the unit-capacity case, augmentation maintains the number of incoming/outgoing edges.

Combining this with the running time of the layered network algorithm in the first part of the algorithm, we get an overall running time of $O(km + \frac{n}{k}m)$, and we are free to choose the value of k. This is minimized by setting $k = \sqrt{n}$, giving a running time of $O(m\sqrt{n})$.

9.5 Another $O(m\sqrt{n})$ algorithm that runs faster in practice

We now present an algorithm of the same structure as the previous one – we divide the problem into two parts: first we find a non-maximum flow such that all remaining paths from source to sink in the residual graph or of length at least $\sqrt{n}$, and then we continue with regular Ford-Fulkerson to find the max flow. This time, instead of using the layered network approach for the first part, we will use the push/relabel approach. Recall that in the push/relabel approach, the label at a node provides a lower bound of its distance to the sink. This gives rise to the following algorithm:

Algorithm 2

1. Run push/relabel until all active nodes have labels $\geq \sqrt{n}$
2. Run basic Ford-Fulkerson to find the remaining augmenting paths (at most $\sqrt{n}$ of them).

There is a technical problem with proving the running time of this algorithm. The node disjointness property depends on the fact that there is either at most one incoming edge or at most one outgoing edge at each node. This is true in these graphs if the conservation property holds. If we allow excesses to accumulate, this property may not hold (Figure 60).

To fix this problem, we need to impose two restrictions on the push operation:

1. Maintain unit excesses only.
2. All excesses can only reside on nodes in L. (i.e. No nodes in R can have any excesses).

Node disjointness will then hold, since each node in L has only one incoming edge at the beginning. Maintaining unit excesses ensures each node in L to have ≤ 1 incoming edge throughout. At the same time, each node in R will have only one outgoing edge if no excesses reside on it.

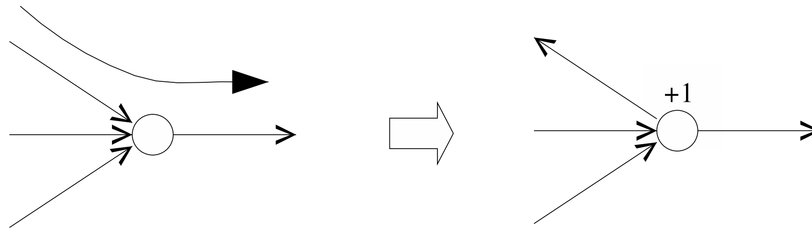


Figure 60: Number of outgoing edges increases when we allow excesses.

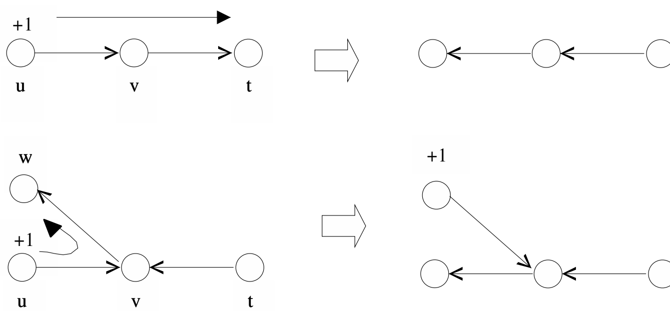


Figure 61: Pushing excess from a node u to a node v.

In this case, if we can make all flow paths to be at least $\sqrt{n}$ long, each path will use at least $\sqrt{n}/2$ nodes in R because we are working on a bipartite graph. Since no two paths can share a node in R, there can be at most $2\sqrt{n}$ such paths.

Now, what remains to be shown is that we can always maintain the above properties in push operations.

When the push/relabel algorithm starts, we saturate all the links from s to L, thus making all nodes in L carry one unit of excess. Afterwards, whenever we push an excess from a node $u \in L$ to a node $v \in R$, there are two cases to consider (Figure 61).

1. If v has an outgoing edge to the sink, push the excess immediately to the sink.This forms a double push operation.
2. If v does not have an outgoing edge to the sink, it must have an outgoing edge to a node $w \in L$ because v did not carry any excess before the push. So we can push the excess from u to v and then to w. What we need to make sure is that w did not have an excess before the push. Otherwise, we will violate the constraint that only unit excesses exist. But this is always the case because w has an incoming edge from v. If w had an excess before, it would not have any incoming edges.

In both cases, no excess resides in R and unit excesses exist only in L i.e. the required properties can be maintained by push operations.

When all active nodes have labels $\geq \sqrt{n}$, the remaining excesses can be returned to the source. We can then ignore the labels and run basic Ford-Fulkerson to augment any remaining paths. The following is a summary of the corrected algorithm, which also runs in $O(m\sqrt{n})$ time:

Algorithm 2 – corrected

1. Push/relabel until all active nodes have labels $\geq \sqrt{n}$. Do not introduce any excesses in R during the process.
2. Return the remaining excesses to the source.
3. Run basic Ford-Fulkerson to find the remaining augmenting paths (at most $O(\sqrt{n})$ of them).

Though both algorithms take the same time asymptotically, the second algorithm can be implemented to run faster that the first. Also the method of excesses allows some 'implementation tricks' that improve the constants in practice. In general push-relabel algorithms are more parallelizable as compared to the layered network algorithms.

9.6 Tricks in the implementation of flow algorithms.

Some Points regarding heuristics and tricks employed in implementation of flow algorithms:

1. If space is not a consideration(or if the graph is dense) then use an adjacency matrix as the representation. It results in good cache behavior when the row corresponding to the outgoing edges from a particular vertex is scanned.
2. In the case of bipartite graphs, sometimes, it helps augmenting path algorithms to start off by finding a greedy matching (This can be converted into a flow, as we have seen earlier) .This matching can be computed by considering the vertices in input order and seeing if they can be matched with some un-matched adjacent node.
3. A periodic application of Global Relabeling helps in push-relabel algorithms. As we know the labels represent a lower bound on the distances from any vertex to the sink. These labels help the algorithm to push flow towards the sink. If we periodically calculate the values of the shortest path from each node to the sink, this helps the push 'to know' the direction to the sink better. We should not do this too often as it would dominate the running time. We should also not do this too rarely, as this would make the algorithm proper-less effective. So it is done every Θ(m) relabels.
4. There are also implementation strategies regarding the order in which the active edges are selected for doing the pushes.These are called highest label (HL) , lowest label (LL) and FIFO. HL corresponds to finishing with one vertex completely, as the push and relabel procedures cause monotonic increases in the label. LL corresponds to a more layered approach. FIFO round robins the selection procedure. Usually we require buckets corresponding to each label to implement the HL and the LL heuristics. The FIFO and the LL implementations are usually among the best algorithms on most test cases.
5. The gap heuristic says that if there are no vertices with a label d then all the nodes with labels>d need not be considered as there is no path to the sink from them. A useful side-effect of the HL and the LL heuristics is that they allow us to implement the gap heuristic.

6. There is also the Augment-Relabel class of algorithms that is a cross between the push relabel and the augmenting path algorithms.These algorithms use the labels to guide the augmenting path search procedure.

7. In general, augmenting methods are better than push methods if the magnitude of the maximum flow is small, otherwise the push-relabel methods show superior performance.

There have been a number of experimental studies of the several heuristics that have been used, e.g. see [3].

10 Partial Orders and Dilworth's Theorem

In this lecture we shall introduce *partial orders* and prove *Dilworth's Theorem*. These have applications in finding concurrency and resource allocation as well. The proof of Dilworth's Theorem will also demonstrate some useful proof techniques.

10.1 Partial orders

In this section we will introduce some definitions.

Definition 10.1. *A* ***(strict) partial order*** *over a set V is a binary relation, $\prec$, over V that is*

1. *irreflexive: for all $x, y \in V, x \neq y$, $x \prec y$ implies $y \not\prec x$ (i.e. if the relation holds for (x, y) it does not hold for (y, x));*
2. *transitive: for all $x, y, z \in V$, $x \prec y$ and $y \prec z$ implies $x \prec z$.*

A structure $(V, \prec)$ such that $\prec$ is a partial order over V is called a **partially ordered set**. When no confusion can arise, we use "partial order" instead of "partially ordered set" to shorten the notation. We say that two distinct elements $x, y \in V$ are **comparable** if $x \prec y$ or $y \prec x$. They are **incomparable** otherwise. Also, the relation $x \prec x$ does not have any meaning and is not used.

We are going to deal only with **finite** partially ordered sets. A convenient representation is a directed graph $G = (V, \prec)$, $\prec$ being the set of edges. The canonical graphical representation for a partial order is that of a graph whose edges, all directed from lower to higher vertices (which is possible since partial orderings are acyclic), represent the ordering relation.

An alternative representation is a directed acyclic graph $G = (V, E)$, where $x \prec y$ if and only if there is a directed path from x to y in E. In other words, the "$\prec$" relationship can be computed by taking a transitive closure of this graph. For simplicity, we will use a general directed acyclic graph in our examples and will not explicitly draw the transitive edges. Figure 62 depicts a partial order that will be used for all subsequent examples. There are two implicit (transitive) edges, $x_1 \prec x_3$ and $x_1 \prec x_5$. Since in the subsequent discussions we will never use the graph without the transitive edges, we will use G to denote the transitive closure of the depicted graph (a transitive closure is an extension or a superset of a binary relation, e.g. $\prec$, such that whenever (a,b) and (b,c) are in the extension, (a,c) is also in the extension).

An example application of partial orderings is the analysis of parallelism of computer code. Vertices represent atomic tasks and the ordering constraints represent the precedence dependencies between them. For example, we can interpret Figure 62 as follows: x_2 and x_4 (and x_1) must terminate before x_3 can be scheduled for execution, but x_5 can run in parallel with x_3. Let us now mathematically define this notion of a sequence of tasks being executed in serial or in parallel.

Definition 10.2. *A* ***chain*** *in a partially ordered set $(V, \prec)$ is a subset of V such that every pair of elements is comparable, i.e.*

$$C \subseteq V \text{ is a chain if and only if } \forall x, y \in C, x \neq y, \ (x \prec y \vee y \prec x).$$

In other words, a chain is a subset of V which is totally ordered by $\prec$. By our definition, all subsets of V of size one are chains. A **chain partition** is a partition of V (i.e., a family of pairwise disjoint, nonempty subsets of V whose union is all of V) each set of which is a chain.

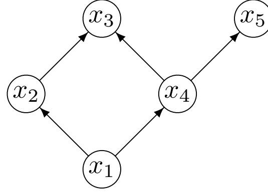


Figure 62: The example partial order.

Definition 10.3. *An* ***antichain*** *is a set of pairwise incomparable elements of* V*, i.e.*

$$U \subseteq V \text{ is an antichain if and only if } \forall x, y \in U \ (x \not\prec y \wedge y \not\prec x).$$

The elements of a chain are usually listed in order, so that in chain $C = \{x_1, x_2, \ldots, x_k\}$ we have $x_1 \prec x_2 \prec \cdots \prec x_k$. It is easy to see that any chain can be represented in this way. In figure 62, $\{x_1, x_2, x_3\}$ and $\{x_1, x_5\}$ are chains, $\{x_2, x_5\}$ is an antichain, $\{x_1, x_2, x_4\}$ is neither a chain nor an antichain, and $\{x_i\}$ is both a chain and an antichain for every i (this is general: the only sets that are both chains and antichains are the singleton sets). Some chain partitions of the example graph are $\{\{x_1, x_2, x_3\}, \{x_4, x_5\}\}$, $\{\{x_1, x_4\}, \{x_2, x_3\}, \{x_5\}\}$, and $\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\}$.

A chain essentially represents a sequence of tasks that must be executed one after another, while an antichain represents tasks that can be executed in parallel. Cardinality of max antichain is the maximum number of tasks one can execute concurrently. But does that mean from the execution perspective, it is greatly parallelizable? Not necessarily, as an example in Figure 63 the antichain formed by $x_1, x_2, \ldots, x_n$ could be large but the whole task in general is not highly parallelizable. Minimum chain partition corresponds to assignment of tasks to minimum number of threads without introducing any unnecessary serialization. Intuitively, it is clear that the number of threads should be at least as large as the maximum number of tasks that can be executed concurrently. Otherwise we "lose" some of the parallelism by serializing tasks that do not have to be serialized.

10.2 Dilworth Theorem

The following theorem formalizes this intuition. In fact, it proves a stronger claim: in the context of parallel task execution, it claims that the minimum number of threads is *equal* to the maximum number of concurrently executable tasks.

Theorem 10.4 (Dilworth's Theorem)**.** *Let* $G = (V, \prec)$ *be a partially ordered set,* U^* *an antichain of maximum cardinality,* ρ^* *a chain partition of minimum cardinality. Then* $|U^*| = |\rho^*|$*.*

We divide the proof of the theorem into two parts. First, we prove that $|U^*| \leq |\rho^*|$ and then we prove that $|U^*| \geq |\rho^*|$.

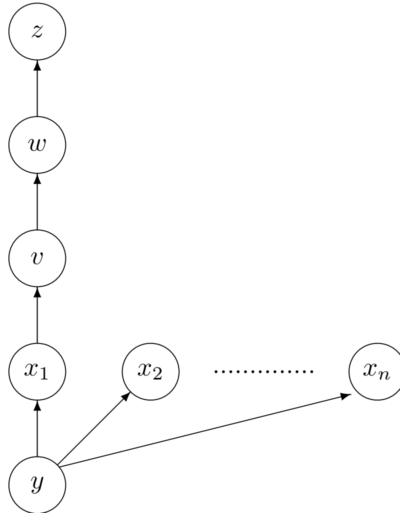


Figure 63: A task sequence that is not highly parallelizable.

Lemma 10.5. $|U^*| \leq |\rho^*|$

Proof: Assume this is not true and consider any antichain U and chain decomposition ρ, where $|U| > |\rho|$. By pigeonhole principle, at least two of the elements of U are in the same chain in the decomposition. But this is a contradiction, since all elements in a chain are comparable. ▮

Lemma 10.6. $|\rho^*| \leq |U^*|$

Proof: Let $V = \{x_1, \ldots, x_n\}$. The main trick is the to construct a bipartite graph $G' = (V', E')$ from the original graph G as follows:

$$V' = \{a_i, b_i \mid x_i \in V\},$$
$$E' = \{(a_i, b_j) \mid x_i \prec x_j \ in \ G\}$$

that is, each vertex x_i in G is split into two vertices a_i, b_i in G', and every edge in G corresponds to an edge in G' from an a vertex to a b vertex. G' is a bipartite graph, the two components being the a vertices on one side, and the b vertices on the other. Figure 64 depicts the new graph corresponding to our previous example. Note that it is important to consider all the relations in the original graph, not just the ones that were drawn (It is a good exercise to see where the proof breaks down if this were not true). The thick edges in the figure give a maximum matching for the graph (to be used later).

The main part of the proof is split into the following two lemmas.

Lemma 10.7. *For any matching M' in G', there exists a chain partition ρ in G such that*

$$|M'| + |\rho| = n$$

where n is the number of vertices in G.

Proof: Starting from a matching M', we can construct a chain partition $\rho = \{C_1, \ldots, C_k\}$ of G as follows. Let G'' be the subgraph of G induced by the set of edges in G corresponding to the edges in M',

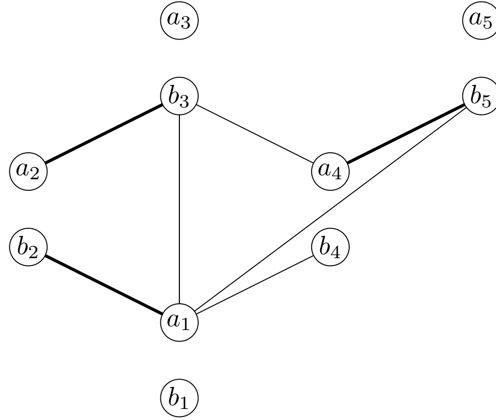


Figure 64: The bipartite graph for the example, with a maximum matching.

i.e. G'' is the graph induced by $\{x_i \prec x_j \mid (a_i, b_j) \in M'\}$. We claim that each connected component of G'' is a simple path (and therefore a chain). Together, the connected components form a chain partition (each isolated vertex becomes a separate chain).

To prove that the connected components are indeed simple paths, notice that, since M' is a matching, for all a_i and b_i in G' there can be at most one edge incident to a_i and at most one edge incident to each b_i. Thus, each vertex in G can have at most one incoming and at most one outgoing edge in G'', and, since G'' is acyclic (because G is), each connected component must be a simple path.

The result of applying this procedure to our example is depicted in Figure 65, where the thick edges represent the paths. The resulting chain partition is $\{\{x_1, x_2, x_3\}, \{x_4, x_5\}\}$.

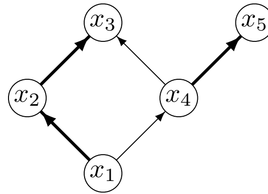


Figure 65: The chain partition resulting from the matching.

Now it is easy to conclude the proof of the lemma:

$$\begin{aligned} n = & \sum_{i=1}^{k} |C_i| \\ = & \; k + \sum_{i=1}^{k} (|C_i| - 1) \\ = & \; |\rho| + |M'| \end{aligned}$$

since the number of edges in M' used to create each chain C_i equals $|C_i| - 1$. ▮

We must think of the above Lemma as: Given a huge matching in G', we can find a small chain ρ in G.

Lemma 10.8. *For any vertex cover, S', in G', there exists an antichain, U, in G such that*

$$|S'| + |U| \geq n.$$

Proof:

Project S' on the original graph G to form S, where $S = \{x_i \mid a_i \in S' \text{ or } b_i \in S'\}$. Then we have $|S| \leq |S'|$, since in general more than one vertex in S' can be mapped to the same vertex in S. Let $U = V \setminus S$. We claim that U is an antichain.

To see why this is true, assume, by contradiction, that there are two comparable vertices $x_i, x_j \in U$, that is, $x_i \prec x_j$ in G. Then the edge (a_i, b_j) is in G'. Since S' is a vertex cover of G', either a_i or b_j must be in S'. Therefore either x_i or x_j will be in S, but this contradicts the fact that both x_i and x_j are in U, by the very definition of U.

Thus, we constructed antichain U such that

$$|S'| + |U| \geq |S| + |U| = n.$$

∎

In our example, a minimum vertex cover of the graph is given by $\{a_1, a_2, a_4\}$, and the corresponding antichain is $\{x_3, x_5\}$.

Now we can go back the proof of Lemma 10.6. If we apply Lemma 10.7 to a maximum matching M^* and Lemma 10.8 to a minimum vertex cover S^* of G', we get that there exist a chain partition ρ and an antichain U such that

$$|M^*| + |\rho| = n \text{ and } |S^*| + |U| \geq n$$

hence

$$|M^*| + |\rho| \leq |S^*| + |U|.$$

Since G' is a bipartite graph, we know that

$$|M^*| = |S^*|,$$

and therefore

$$|\rho| \leq |U|.$$

The claim follows since $|\rho^*| \leq |\rho|$ and $|U| \leq |U^*|$. ∎

Dilworth Theorem follows immediately by combining Lemmas 10.5 and 10.6.

As a final remark, notice that every step in the above proof is constructive, so it yields a procedure for finding a maximum antichain and a minimum chain partition.

11 Farkas Lemma and Proof of Duality

11.1 The Farkas Lemma

Consider the equation:

$$\left.\begin{aligned} Ax &= b \\ x &\geq 0 \end{aligned}\right\}$$

To prove that it has a feasible solution only needs a short certificate, i.e. to give a feasible solution x. But how to prove that it does not have feasible solutions? It turns out that for this problem, we can prove non-existence also using a short certificate. The tool needed is Farkas Lemma:

Lemma 11.1 (Farkas Lemma)**.**

$$(\exists x \geq 0, Ax = b) \Leftrightarrow (\forall y, \ (\quad y^T A \geq 0 \Rightarrow y^T b \geq 0)).$$

Proof: We only prove the implication "$\Rightarrow$". The proof of converse is omitted.(See [14], section 7.1.)

Assume there exists y_0 that satisfies $y_0^T A \geq 0$, but $y_0^T b < 0$. Now assume that there also exists x_0 that satisfies $x_0 \geq 0, Ax_0 = b$. Then we have:

$$Ax_0 = b \tag{16}$$

$$y_0^T(Ax_0) = y_0^T(b) \tag{17}$$

$$(y_0^T A)x_0 = y_0^T b \tag{18}$$

Since $y_0^T A \geq 0$ and $x_0 \geq 0$, we have $(y_0^T A)x_0 \geq 0$, contracting with the assumption $y_0^T b < 0$. Therefore, no such x_0 exists. ∎

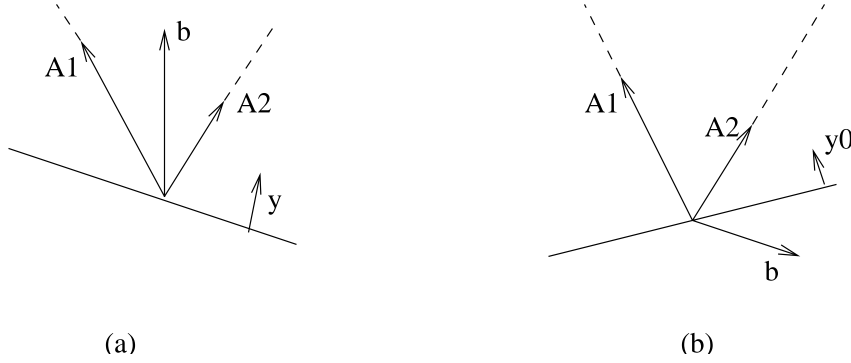
Although we are not going to give a formal proof of "$\Leftarrow$" direction, here is an intuitive explanation why it is correct. Think of the $n*m$ matrix A as a vector of some column vectors: $A = [A_1, A_2, \ldots, A_m]$, where each A_i is an n-dimensional column vector. Then $b = Ax = [A_1, A_2, \ldots, A_m] * x$, where $x \geq 0$ is a non-negative combination of A_is, therefore in an n-dimensional hyperspace, vector b lies inside the cone confined by those A_i vectors. In this hyperspace, y defines a hyperplane, and the sign of $y^T v$, where v is some vector, tells to which side of the hyperplane v lies. Since $y^T A \geq 0$, all A_is are to the same side of the hyperplane. "$\Rightarrow$" says in this case, any vector inside the cone is also to the same side of hyperplane y. For "$\Leftarrow$", if $\nexists x \geq 0, Ax = b$, then b is outside the cone, and we can always find one hyperplane y_0 such that the cone is to one side of y_0, while b is to the other side. Figure (a) and (b) illustrate the two cases in a 2-dimensional space.

11.2 Alternative Forms of Farkas Lemma

In the last section we stated Farkas Lemma in the following form:

$$(\exists x \geq 0, Ax = b) \Leftrightarrow (\forall y, \quad y^T A \geq 0 \Rightarrow y^T b \geq 0).$$

Farkas Lemma comes in several alternative forms. Rather than prove the various forms from scratch, we can convert them to the basic form and then apply the above version of the Farkas Lemma to get the desired result.



(a) (b)

Consider the following alternative form:

Lemma 11.2.

$$(\exists x, Ax \leq b) \Leftrightarrow (\forall y \geq 0, \quad y^T A = 0 \Rightarrow y^T b \geq 0).$$

Proof: Firstly, convert $\exists x$ into $\exists x \geq 0$. Since any x can be written as difference of two non-negative numbers, we replace x with $x_1 - x_2$ and convert the left side to:

$$x_1 \geq 0,\ x_2 \geq 0, [A; -A] * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq b$$

Next, convert inequality $Ax \leq b$ to an equality using a slack variable. Since $A(x_1 - x_2) \leq b$, we can add a positive slack variable, x_s, to $x_1 - x_2$ such that $A(x_1 - x_2 + x_s) = b$. Now the left side matches the original form:

$$x_1 \geq 0,\ x_2 \geq 0,\ x_s \geq 0; [A; -A; I] * \begin{bmatrix} x_1 \\ x_2 \\ x_s \end{bmatrix} = b$$

Let $A' = [A; -A; I]$. Thus the original right side reads:

$$\forall y, \quad y^T A' \geq 0 \Rightarrow y^T b \geq 0$$

Because multiplying y by A' just multiplies each column of A' by y, the above form is equivalent to:

$$\forall y, \quad \big(y^T A \geq 0, -y^T A \geq 0, y^T I \geq 0\big) \Rightarrow y^T b \geq 0$$

$y^T I \geq 0$ is equivalent to $y \geq 0$. Both $y^T A \geq 0$ and $-y^T A \geq 0$ imply $y^T A = 0$. Thus we have syntactically converted the alternative of Farkas Lemma into the original, and therefore have proved the alternative form correct:

$$\begin{matrix} x_1 \geq 0 \\ x_2 \geq 0 \\ x_s \geq 0 \end{matrix}\ ,\ [A; -A; I] * \begin{bmatrix} x_1 \\ x_2 \\ x_s \end{bmatrix} = b \ \Leftrightarrow\ \forall y \geq 0, y^T A = 0 \Rightarrow y^T b \geq 0$$

∎

11.3 Duality of Linear Programming

Consider a linear program (the primal) and its related dual. The optimal solution of the primal is equal to the optimal solution of the dual

Primal		Dual
$max\{cx \mid Ax \leq b\}$	$=$	$min\{y^T b \mid y \geq 0, y^T A = c\}$

This is called the Duality of Linear Programming.

Using Farkas Lemma, we will prove that if both the primal and the dual are feasible then LP Duality is valid. Results exist for the situation when there is no solution as also the one in which the solution is unbounded. These will not be covered in this class.

- Claim 1: $\forall x, y$ feasible, $cx \leq y^T b$.
- Claim 2: $\exists x^*, y^*$ feasible, $cx^* \geq y^{*T} b$.

If both of these claims are true (assuming a feasible solution exists for both the primal and the dual), then we have $cx^* = y^{*T} b$ since cx^* has to be both $\leq$ and $\geq y^{*T} b$.

Proof: (Claim 1) Assume x,y is a feasible solution. We want to show that $cx \leq y^T b$:

$$cx = (y^T A)x = y^T(Ax) \leq y^T b$$

The first equation holds because $y^T A = c$ (according to dual); the last inequality holds because $Ax \leq b$ and $y \geq 0$. ∎

Claim 1, referred to as *weak duality*, establishes that the value of a feasible solution to the dual is an upper bound on feasible solutions to the primal.

Proof: (Claim 2) We would like to show that there exist feasible solution x^*, y^* such that $cx^* \geq y^{*T} b$. In fact, these x^*, y^* are optimum by Claim 1 above.

Begin the proof by creating a matrix of inequalities containing all the constraints of the primal, the dual as well as the the additional inequality $cx^* \geq y^{*T} b$, which we are trying to believe :

$$\begin{bmatrix} A & 0 \\ -c & b^T \\ 0 & A^T \\ 0 & -A^T \\ 0 & -I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} b \\ 0 \\ c^T \\ -c^T \\ 0 \end{bmatrix}$$

First notice the above equation represents all related inequalities:

1. $Ax \leq b$, (1 is the only inequality from Primal),
2. $-cx + b^T y \leq 0 \equiv y^T b \leq cx$, (2 is what we are trying to prove),
3. $A^T y \leq c^T$, and

4. $-A^T y \leq -c^T$, and (3&4 imply $y^T A = c$ which is the first inequality from the Dual) ,
5. $-Iy \leq 0 \equiv y \geq 0$ (5 is the other inequality from Dual)

Notice that if we call the first matrix from the equation A', the next vector x', and the last vector b', then proving claim 2 is equivalent to proving the existence of vector x' satisfying all these inequalities. By Farkas Lemma, it is equivalent to prove:

$$\forall \gamma \geq 0, \gamma^T A' = 0 \Rightarrow \gamma^T b' \geq 0.$$

γ is a vector whose size equals the number of rows in A'. We break γ into five segments $\gamma^T = [u, \lambda, v, w, q]$ where:

- u is the segment of γ which multiplies $[\ A \quad 0\ ]$ in A',
- λ is the segment of γ which multiplies $[\ -c \quad b^T\ ]$ in A',
- v is the segment of γ which multiplies $[\ 0 \quad A^T\ ]$ in A',
- w is the segment of γ which multiplies $[\ 0 \quad -A^T\ ]$ in A', and
- q is the segment of γ which multiplies $[\ 0 \quad -I\ ]$ in A'.

We are to prove:

$\forall \gamma^T = [u, \lambda, v, w, q] \geq 0, \gamma^T A' = 0 \Rightarrow \gamma^T b' \geq 0.$

Multiplying out $\gamma^T A'$ and $\gamma^T b'$:

$\forall u, \lambda, v, w, q \geq 0, (uA - \lambda c = 0, \lambda b^T + (v - w)A^T - qI = 0) \Rightarrow ub + (v - w)c^T \geq 0$

Since $qI \geq 0$ we can drop it and replace $=$ with $\geq$:

$\forall u, \lambda, v, w, q \geq 0, (uA - \lambda c = 0, \lambda b^T + (v - w)A^T \geq 0) \Rightarrow ub + (v - w)c^T \geq 0.$

We prove it by considering 2 cases, $\lambda > 0$ and $\lambda = 0$:

1. Case $\lambda > 0$:

$$
\begin{aligned}
ub &= b^T u^T \\
&= \frac{1}{\lambda}(\lambda b^T u^T) \\
&= \frac{1}{\lambda}(\lambda b^T) u^T \\
&\geq \frac{1}{\lambda}(w-v)A^T u^T, && \text{since} \begin{cases} \lambda b^T + (v-w)A^T \geq 0 \\ \frac{1}{\lambda} > 0 \\ u^T \geq 0 \end{cases} \\
&\geq \frac{1}{\lambda}(w-v)(uA)^T \\
&\geq \frac{1}{\lambda}(w-v)(\lambda c)T && \text{since } uA - \lambda c = 0 \\
&\geq (w-v)c^T
\end{aligned}
$$

and therefore $ub + (v - w)c^T \geq 0$.

Consequently $\forall u, v, w, q \geq 0, \forall \lambda > 0, \gamma^T A' = 0 \Rightarrow \gamma^T b' \geq 0$ is true, which means that there exists a x' which solves the inequality $A'x' \leq b'$, thus $\exists x^*, y^*$ which satisfies $y^T b \leq cx$.

2. Case $\lambda = 0$.

   Intuitively, when $\lambda = 0$ the inequality $y^T b \leq cx$ (inequality #2) is ignored, leaving only inequalities 1, 3, 4, and 5, which are simply the primal and the dual. Because we started out assuming that both the primal and dual had feasible solutions, there must exist a feasible solution x' which satisfies $A'x' \leq b'$ when $\lambda = 0$. To formalize the intuition:

   Given: $uA = 0$, $(v - w)A^T - qI = 0$, and feasible solution: $Ax \leq b, y^T A = c^T, y \geq 0$. Show: $ub + (v - w)c^T \geq 0$.

   With $\lambda = 0$ we are left with following inequality matrix:

$$
\begin{bmatrix} A & 0 \\ 0 & A^T \\ 0 & -A^T \\ 0 & -I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} b \\ c^T \\ -c^T \\ 0 \end{bmatrix}
$$

   Because we assumed a feasible solution we can invoke Farkas Lemma to claim that:

   $\forall u, v, w, q \geq 0, uA = 0, (v - w)A^T - qI = 0 \Rightarrow ub + (v - w)c^T \geq 0$

   which is exactly what we are trying to prove.

∎

12 Examples of primal/dual relationships

We will consider several examples of duality in linear programming; these examples show that more often than not, from solving the dual problem one may get insight into the solution to the primal problem.

12.1 Complementary Slackness

For some of the examples below, it will be useful to use the notion of "complementary slackness". Consider an LP

$$\max\{cx|Ax \leq b, x \geq 0\}$$

and its dual

$$\min\{y^t b|y^t A \geq c, y \geq 0\}$$

Strong duality implies that for any optimum pair of primal and dual solutions x, y, we have:

$$cx = y^t Ax = y^t b$$

Since x and y are both non-negative, the above equalities imply that:

- for any $x_i > 0$, we have corresponding dual inequality tight, i.e. $(y^t A)_i = c_i$
- for any $y_i > 0$, we have corresponding primal inequality tight, i.e. $(Ax)_i = b_i$.

12.2 Maximum bipartite matching

Consider the problem of finding a maximum matching on a bipartite graph G. Assume that $V = L \cup R$ and all the edges are between nodes in L and nodes in R. Let $x(uv) = 1$ denote that the edge uv is in the matching. Relaxing the resulting IP, we get the following LP:

$$\begin{aligned} \text{maximize} \quad & \sum_{uv} x(uv) \\ \forall uv \in E: \quad x(uv) & \geq 0 \\ \forall v \in R: \quad \sum_u x(uv) & \leq 1 \\ \forall u \in L: \quad \sum_v x(uv) & \leq 1 \end{aligned}$$

The dual has one variable per node in the graph. Call these variables $y(u)$ and $z(v)$.

$$\begin{aligned} \text{minimize} \quad & \sum_u y(u) + \sum_v z(v) \\ \forall u \in L: \quad y(u) & \geq 0 \\ \forall v \in R: \quad z(v) & \geq 0 \\ \forall uv \in E: \quad y(u) + z(v) & \geq 1 \end{aligned}$$

Note that the resulting dual is exactly the problem of finding a (fractional) minimum node cover of G (which we earlier saw was related to the problem of maximum matching).

12.3 Shortest path from source to sink

Consider the problem of finding the distance from s to t. For vertex v in the graph, we have a variable $d(v)$. One can think about the shortest path problem in the following way:

- pick up t from the graph and start pulling it out (assuming that the graph is "attached" to the ground by s);
- at some point, you will stop because the *shortest* path from s to t is completely stretched.

Translating the above intuition into LP form, we get:

$$\begin{cases} \forall\ (uv) \in E \quad d(v) \leq d(u) + cost(uv) \\ maximize \ \ d(t) - d(s) \end{cases}$$

- The inequalities here are upper bounds on distances; for convenience, they may be rewritten as $d(v) - d(u) \leq cost(uv)$.
- Though we want to find the shortest path we are maximizing. How can this be? We approach the solution from the bottom, that is we start with small $d(v)$'s and increase it as long as possible, that is as long as the inequality holds. The shortest path then is the path along which the inequalities become equalities. All the other paths must be at least as long otherwise the corresponding constraints would have become tight earlier.
- For convenience, one may assume beforehand that $d(s) = 0$.
- An all-zeros solution ($\forall u \ \ d(u) = 0$) is a feasible solution of this problem.

Variables in the primal problem correspond to inequalities in the dual. Since $d(u)$ are unrestricted in the primal problem, we get strict equalities in the dual. Thus the dual will look like $y^t A = c$. Looking at the primal, we see that each column of the matrix A consists only of 0,1,-1 entries. Columns correspond to nodes and rows correspond to edges. Consider a column that corresponds to node v. It has zeros in all rows that correspond to edges that do not have v as their endpoint. It has +1 entry for each edge where the edge is of the form uv and -1 for each row that corresponds to an edge of the form vw. Thus, we get:

$$\begin{array}{rcl} \text{minimize} & & \sum_{uv \in E} \ell(uv) cost(uv) \\ \ell(uv) & \geq & 0 \quad \forall (uv) \in E \\ \sum_{u:uv \in E} \ell(uv) - \sum_{u:vu \in E} \ell(vu) & = & \begin{cases} 0 & \forall v \neq t \\ 1 & \text{for} v = t \\ -1 & \text{for} v = s \end{cases} \end{array}$$

From the duality theorem, we have:

$$dist(s \to t) = min \sum_{uv \in E} \ell(uv) cost(uv)$$

Observe that our dual problem looks exactly like a flow problem with flow variables $\ell(uv)$! In fact it is a *min cost flow problem* where the goal is to bring exactly 1 unit of flow from source to sink while

minimizing the cost. Observe that we do not have capacity constraints. The formulation of the dual problem unveils the connection between the uncapacitated min-cost flow problem and the shortest path problem.

Observe that our result "makes sense": given an uncapacitated network, any $s-t$ path can be used to bring the required unit of flow. The cost will be equal to the length of this path. Hence, as long as we restrict our attention to single paths (do not allow to split flow among several $s-t$ paths), the best solution is the shortest $s-t$ path. Observe that splitting flow among several $s-t$ paths does not help. (Why?)

12.4 Max flow and min-cut

Consider the max flow problem on a graph G. Let us try to formulate LP for the max-flow on this graph. The first problem that we encounter in formulating the LP is what should be chosen as variables. There are two approaches here:

1. Choose a variable for every edge and write equations for conservation and capacity constraints.
2. Choose a variable for every possible $s-t$ path.

The advantage of the former formulation is that the number of variables is polynomial in size where as in the latter we have a large number of variables. But the second approach is closer to how we think and see the problem. So let us try to use the path-based approach to formulate LP for the max-flow problem.

Let p be an $s-t$ path in G and $f(p)$ the flow along that path. Then we have:

$$\begin{array}{rrcl} & \text{maximize} & & \sum_p f(p) \\ \forall\, e: & \sum_{e \in p} f(p) & \leq & cap(e) \\ \forall\, p: & f(p) & \geq & 0 \end{array}$$

The dual has one variable per capacity constraint. Call these variables $\ell(uv)$. Denote a_{ij} as the element of A in the ith row and j^{th} column. Note that we can write:

$$a_{ij} = \begin{cases} 1 & \text{if path } p_i \text{ contains } j^{th} \text{ edge} \\ 0 & \text{otherwise} \end{cases}$$

There may be some confusion as to why the elements of matrix A should contain just 1's and 0's. The matrix A contains coefficients and not values. e.g., if there is a capacity constraint $f(p_5)+f(p_8) \leq cap(e_1)$ we have $A[e_1][p_5] = 1$, $A[e_1][p_8] = 1$ and $A[e_1][allotherpaths] = 0$. Also note that the order in which edges appear depends on how we numbered the edges which is arbitrary.

Following the guidelines for constructing a dual problem, we get the following LP:

$$\begin{aligned} \text{minimize} \qquad & \sum_{uv \in E} \ell(uv) cap(uv) \\ \forall p: \sum_{uv \epsilon p} \ell(uv) & \geq 1 \\ \forall uv \in E: \ \ell(uv) & \geq 0 \end{aligned}$$

We can interpret this LP in the following manner: $\ell(uv)$ denotes the length of the edge uv; $cap(uv)$ denotes the volume per unit length of the edge uv. We are trying to minimize the total volume of edges (which might be thought of as rubber pipes) used while making sure that each path from s to t is of length at least 1. Note that $\ell(uv)$ is not a real length but should be considered as a "virtual" length.

It is easy to see the weak duality relationship in this case, i.e. that any solution to the primal is less than any solution to the dual. Indeed, suppose we have a feasible dual ℓ and a feasible primal (flow) f. Then the flow f "fits" within our system of pipes with lengths of pipe uv defined to be $\ell(uv)$. Clearly, the total volume of the available pipes is not less than the total volume of the flow, which gives

$$\sum_p f(p) \leq \sum_{uv} \ell(uv) c(uv)$$

Given an $s-t$ cut $(A, V-A)$ in G, with $s \in A$, define ℓ_{cut} to be equal 1 on edges that cross from A to $V-A$ and 0 otherwise. Observe that this is a valid dual solution, since any path has to cross the cut at least once and thus $\forall p: \sum_{uv \in p} \ell(uv) \geq 1$ is satisfied. (Note that, in general, an $s-t$ path might cross the cut several times, "going back and forth" between A and $V-A$.).

The value of the dual associated with such ℓ_{cut} is $\sum_{uv} \ell_{cut} cap(uv) =$ capacity of the cut. Weak duality immediately implies that the value of max flow is limited by the capacity of any one of these cuts, i.e. it is limited by the capacity of minimum cut. Of course, this is not news to us, but it is still interesting that one can use duality to prove this claim.

In fact, using duality we can compute min cut directly from the optimum dual solution. Let ℓ be an optimal solution to the dual. Then define A to be the set of nodes reachable from s using 0 length (that is $\ell(uv) = 0$) edges. Define the set B to be all other nodes. Note that since ℓ is feasible, we have $s \in A$ and $t \in B$. Consider the $s-t$ cut (A, B) and define ℓ' be the feasible dual corresponding to this cut, i.e. it is equal to 1 on edges that cross the cut from A to B and equal to zero otherwise.

Given a pair of optimum primal and dual solutions, complementary slackness implies that if $f(p) > 0$ (that is, flow flows along the path p), then we have $\sum_{uv \in p} \ell(uv) = 1$. In other words, p is a shortest path from s to t (since all paths have length at least 1) i.e., optimum flow flows along the shortest path. Moreover, if $\ell(uv) > 0$, then $\sum_{p: uv \in p} f(p) = cap(uv)$, that is, the edge uv is saturated by the flow f.

This immediately implies that the cut (A, B) that we have just constructed has to be saturated by any max flow. In order to conclude that the value of max flow is indeed equal to the capacity of this cut, we have to show that each path p that has non-zero flow in the optimum solution crosses the cut only once. Consider a path that crosses the cut more than once. Since there is flow on this path, complementary slackness implies that the length of this path p with respect to ℓ is exactly 1. Say p crosses from A to B on u_1u_2, then back on u_3u_4, and forward again on u_5u_6. Observe that, by construction, $\ell(u_1u_2) > 0$. Moreover, $u_5 \in A$ and hence there is a path from s to u_5 that uses only edges with $\ell(e) = 0$. Thus there exists a path p' that goes from s through u_5 and then follows p to t, such that the length of p' with respect to ℓ is strictly less than 1, which is a contradiction.

12.5 Multicommodity Flow

Recall the wire routing problem we considered before. The objective there was to find a path for each wire. As the first step in our approximation algorithm we assumed that we (fractionally) solve LP which tells us which path is taken (fractionally) by each wire.

In general, assume that we are given a graph G with capacities *cap* on the edges and "commodities" $1, 2, \ldots, k$. Each commodity i has an associated source s_i, sink t_i, and demand d_i. The goal is to find a feasible (fractional) flow that satisfies all of the demands. In other words, the flow associated with commodity i should bring demand d_i from s_i to t_i. In our wire routing example, commodity i corresponds to a request to route a wire between two points s_i and t_i.

Instead of looking for a feasible solution, we will write an LP that tries to find the maximum demand multiplier z such that the problem is feasible even if we multiply all of the demands by z.

For simplicity, we will use the "path formulation". Let p_i^j denote the jth path connecting s_i and t_i. Let $f(p_i^j)$ to denote a flow of commodity i along this path.

$$
\begin{array}{rrcl}
 & \text{maximize} & & z \\
\forall i: & \sum_j f(p_i^j) & = & z d_i \\
\forall e: & \sum_{e \in p_i^j} f(p_i^j) & \leq & cap(e) \\
 & z & \geq & 0 \\
 & f(p_i^j) & \geq & 0
\end{array}
$$

Building the dual problem:

- $\sum_j f(p_i^j) - z \cdot d_i = 0$ in the primal corresponds to a variable q_i in the dual.
- $\sum_{e \in p_i^j} f(p_i^j) \leq cap(e)$ corresponds to a variable $\ell(e)$ in the dual.
- The column of A for z consists of k entries of the form $-d_i$ one entry per commodity, and zero entries, one per edge.

Our first attempt at formulating the dual problem:

$$
\begin{array}{rrcl}
 & \text{minimize} & & \sum \ell(e) cap(e) \\
 & -\sum d_i q_i & \geq & 1 \\
\forall p_i^j: & \sum_{e \in p_i^j} \ell(e) + q_i & \geq & 0
\end{array}
$$

- We have inequality in $-\sum d_i \cdot q_i \geq 1$ in the dual because in the primal, we maximize $1 \cdot z$ under the condition $z \geq 0$;

- $f(p_i^j)$ appears only once in $\sum f - z \cdot d_i$, and appears for all edges in p_i^j;
- we have inequality in $\sum_{e \in p_i^j} \ell(e) + q_i \geq 0$ because in the primal, we have $f(p_i^j) \geq 0$.

Let us rewrite the dual so that it is more meaningful:

- $\sum_{e \in p_i^j} \ell(e)$ is the length of the path p_i^j.
- Let $q_i = -y_i$.

After these changes, the dual problem is transformed into:

$$\begin{array}{rcl} \text{minimize} & & \sum \ell(e) cap(e) \\ \sum d_i y_i & \geq & 1 \\ \forall p_i^j : \; length_\ell(p_i^j) & \geq & y_i \end{array}$$

We would like to claim that y is not an "important" part of the dual solution. Indeed, suppose ℓ is given. Observe that y does not affect the value of the objective function. Thus, we can safely try to increase it as much as possible in order to satisfy $\sum d_i y_i \geq 1$. If we succeeded in satisfying this constraint, we have a feasible solution. Otherwise we need to go and find a different ℓ. We can not set y values arbitrary high since we need to satisfy $\forall p_i^j : \; length_\ell(p_i^j) \geq y_i$. The highest value that we can set y_i to is the length, with respect to ℓ, of the shortest s_i to t_i path.

Thus, we get the following formulation:

$$\begin{array}{rcl} \text{minimize} & & \sum \ell(e) cap(e) \\ \sum_i d_i dist_\ell(s_i \to t_i) & \geq & 1 \end{array}$$

Let us introduce the notion of the *volume* of the system:

- $cap(e)$ is a cross-section,
- ℓ is the path length.

Then their product is the path volume.

Each piece of flow is flowing along a path of length at least $dist_\ell(s_i \to t_i)$. Thus, total flow of commodity i uses at least $z d_i dist_\ell(s_i \to t_i)$ volume. This implies:

$$\text{total volume} \geq z \sum_i d_i dist_\ell(s_i \to t_i);$$

Rewriting the inequality:

$$\begin{aligned} z &\leq \frac{\text{total volume}}{\sum_i d_i dist_\ell(s_i \to t_i)} \\ &= \frac{\sum \ell(e) cap(e)}{\sum d_i dist_\ell(s_i \to t_i)} \end{aligned}$$

from $\sum d_i dist_\ell(s_i \to t_i) \geq 1$, we obtain:

$$z \leq \sum \ell(e) cap(e);$$

in other words, the primal is the lower bound on the dual (just as we have expected).

As we know, in the optimum

$$z = \sum \ell(e) \cdot cap(e),$$

i.e., we have succeeded in filling up the volume.

13 Online Algorithms

13.1 Introduction

In the algorithms we have studied before, all the information necessary to solve the problem has been available at the beginning. Simply put: data first, answers later. These algorithms are called off-line. In this lecture, we move to a new paradigm - online algorithms.

With online algorithms one does not have all the information at the beginning. The algorithm gets fed information as it runs, and must make a decision immediately upon receiving the new piece of data. For instance, a web server may receive a request for a page, which must be immediately assigned to a machine. When another page is requested, another decision must be made again. By contrast, the algorithms we have studied so far (off-line algorithms), would receive all the requests for web pages at the beginning, and can decide how to assign all requests to machines, before the requests begin to appear.

Without being able to predict what is going to happen later, an online algorithm must make an irrevocable decision any time a new request appears. These decisions can have permanent consequences; for example, once one assigns a request to a machine, it may not be possible to migrate it to another computer. In another word, no backtracking is allowed in online algorithms. Somehow, online algorithms must make decisions that produce good outcomes in the long run while having little apriori knowledge of what requests may arrive.

In this note, we start to introduce online algorithms with a relatively simple example, involves renting skis. After briefly reviewing terminology used in online algorithms, we move on to the second example, an exploration of the properties of various online/off-line algorithms for caching.

13.2 Ski Problem and Competitive Ratio

You are going skiing several times during the season. You can either:

1. Rent skis for \$10
2. Purchase skis for \$100, which will last a season

Clearly, you wish to minimize the amount of money you spend during this season.

In the off-line case, you know how many times you will go skiing. Obviously, if you are going skiing ten or more times, you buy the skis up front for \$100. Otherwise, you rent at \$10 per time you go skiing.

In the online case, you don't know how many times you will go skiing. So what do you do? There are a number of strategies you could try: rent always, buy immediately (the first time you go skiing), or rent for a while, then buy. We need a way to compare these approaches.

The most common way of measuring performance for online-algorithms is called the competitive ratio, which is defined as follows:

$$\text{Competitive ratio } = \sup_{\sigma} \frac{\text{performance on-line}(\sigma)}{\text{performance off-line}(\sigma)}$$

where σ is a sequence of events to the program.[5]

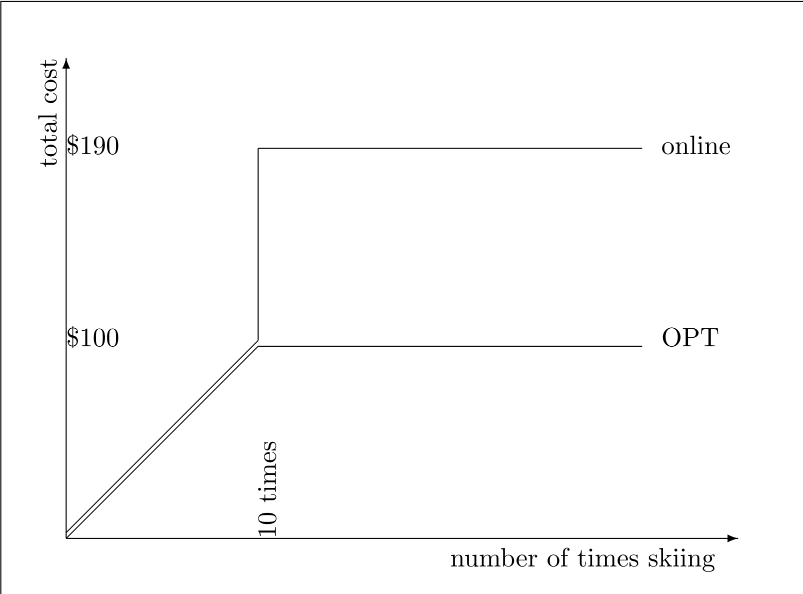
[5] The sup is used instead of max due to the possibly infinite size of the set.

The competitive ratio gives an upper bound on how bad the online algorithm performs as compared to the off-line algorithm. In a sense, it is a measure of worst case performance. When there is no upper bound, we say the algorithm is *not competitive*.

It is important to note that the competitive ratio isn't always the best way to analyze an algorithm. For example, an algorithm could perform extremely poorly (say a factor of 1000) on some pathological inputs and very well (within a factor of 2) on the vast majority of inputs. Its competitive ratio would be 1000. Meanwhile, an algorithm that was consistently at most a factor of 10 off the offline optimum would have a competitive ratio of just 10. In some cases, we might be more satisfied with the performance of the first case, and this isn't reflected in the competitive ratio. In a way, this is the same problem as the one we encounter with worst case analysis of running time.

Returning to the ski problem, the "rent always" strategy is not competitive. By making the number of trips arbitrarily large, it is possible to make the ratio of online vs. offline cost to be unbounded. Buying the first time has a competitive ratio of 10. After all, in the worst case you only go skiing once and pay \$100 with this strategy as opposed to the off-line algorithm's \$10.

The rent for a while, then buy strategy turns out to be better than either of the other strategies, measured by competitive ratio defined above. Consider renting nine times then buying on the tenth. This strategy will make you pay \$190 dollars if you go skiing ten or more times, within a factor of 2 of the \$100 you would pay in the off-line case. If you go skiing less than ten times, you pay the same as in the off-line case.



To generalize the algorithm, assume you can rent a pair of skis for a dollars, and can buy it for Ta dollars, the best online algorithm is to rent skis for *T-1* times and then buy. In the worst case (you ski exactly T times), you are paying *2T-1* dollars, and the cost of the optimal offline strategy is T. Hence this algorithm is *(2-1/T)*-competitive, an it is also the best online algorithm.

13.3 Paging and Caching

A *cache* is fast memory meant to store frequently used items. We will model a cache as an array of k pages, and the rest of the pages are in slower memory. The input sequence σ in this problem are requests to pages, $< \sigma_1, \sigma_2, \sigma_2 ... >$. We will try to analyze the simplest possible variant of the problem, where each page can reside anywhere in the cache.

When a requested page is not in cache, a *page fault* happens, and the page is brought into the cache. If the cache happens to be full at that time, we need a strategy to decide which page to evict in order to bring in the new page. We will consider several paging strategis and will investigate their compatitive ratios.

In order to define a competitive ratio, we need the notion of "cost". The simplest approach is to set cost to be equal to the number of page faults. In this case the competitive ratio of our online algorithm $\mathcal{A}$ is defined by:

$$\text{Competitive ratio} = \sup_{\sigma} \frac{\text{Number of page faults of } \mathcal{A}}{\text{Number of page faults of optimal off-line algorithm}}.$$

The choice of cost is very important when we interpret the competitive ratio. For example, instead of using page fault count as our cost, we could have counted cache hits. In this case, the competitive ratio would have been the supremum of the ratio of offline page hits over online page hits. Suppose you achieve a competitive ratio of 2 in terms of hits. Although "2X" sounds good, it is really not. Consider the case where optimal paging finds, say, 90% of pages already in the cache. In this case, we are guaranteed to find at least 45%, which is quite bad. In general, you want any approximation algorithm to claim its ratio on the smaller term, (in this case, page faults instead of cache hits).

13.3.1 Last-in First-out (LIFO)

For some input sequences, LIFO can be the best strategy. For example, given a cache with 2 items, and a sequence $\sigma = (abc)^\infty = abcabcabcabc\ldots$, LIFO is the best strategy. However, LIFO is a poor online (and off-line) caching algorithm. It has no upper bound on its competitive ratio, ie, a cache implementing LIFO as its eviction algorithm can be made to have an infinite number of page faults, while the optimal algorithm will only have a finite number of page faults. Thus LIFO is *not competitive.*

Consider the same cache with 2 items, but now the sequence of accesses $\sigma = a(bc)^\infty = abcbcbcbc\ldots$. The optimal algorithm will first evict a and then there will be no further cache misses. However, LIFO always evicts from the second slot ie, alternately evicting b and c. LIFO will never evict a, so there will be a page fault on each access. Since there is no upper bound on the number of requests in the sequence, the competitive ratio is unbounded.

13.3.2 Longest Forward Distance (LFD)

Before moving on to discuss online paging algorithms, we first consider the optimal off-line caching algorithm - *Longest Forward Distance* algorithm or LFD. Each time LFD needs to evict a page, it looks into the future requests and evicts a page (that is currently in the cache) that is going to be requested furthest into the future.

Theorem 13.1. *LFD is the optimal off-line caching algorithm.*

Intuitive arguments: The argument here is that for any off-line algorithm $\mathcal{A}$ that alleges to be better than LFD, we can "massage" the algorithm to incorporate LFD rules, and the resultant algorithm $\mathcal{A}^*$ performs no worse than $\mathcal{A}$. To construct $\mathcal{A}^*$, assume that $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ is the sequence that $\mathcal{A}$ outperforms LFD. At some point of the sequence, the two algorithms diverge. Let i be that point, and in serving request σ_i, LFD evicts page v, and $\mathcal{A}$ evicts page u. Unlike $\mathcal{A}$, $\mathcal{A}^*$ will kick out v as what LFD does. Since LFD has chosen to evict v instead of u, the next request for v must come after the request for u.

Time:	...	i	...	t_1	...	t_2	...
Request:	...	?	...	u	...	v	...
$\mathcal{A}^*$:	...	*Fault*	...	*NoFault*	...	*Fault*	...
$\mathcal{A}$:	...	*Fault*	...	*Fault*	...	*NoFault*	...

After time i, each time $\mathcal{A}$ faults on a page other than u, $\mathcal{A}^*$ also faults, and we can replace the same page in $\mathcal{A}$ and $\mathcal{A}^*$. If $\mathcal{A}$ faults on u, $\mathcal{A}^*$ does not fault and performs better than $\mathcal{A}$. If on the other hand $\mathcal{A}^*$ fault on v, $\mathcal{A}$ may not fault, but since the first request for u comes before request for v, $\mathcal{A}$ have already faulted on u, and it may also fault on v. Thus intuitively, $\mathcal{A}^*$ faults no more than $\mathcal{A}$.

Proof: Denote t, t_1 and t_2 the first time that $\mathcal{A}$ replaces v, the first time that u is requested after time i, and the first time that v is requested after time i respectively. Clearly, $t_1 \leq t_2$. We also use a notation x/y to represent the fact that cache entries for $\mathcal{A}$ and $\mathcal{A}^*$ differ by only one entry: $\mathcal{A}$ has x and $\mathcal{A}^*$ has y in their caches. We consider three scenarios: $t \leq t_1$, $t_1 < t \leq t_2$ and $t > t_2$.

1. $t \leq t_1$: After time i, $\mathcal{A}$ has cache $A + v$ and $\mathcal{A}^*$ has cache $A + u$. The cache difference is v/u. Before time t, any time $\mathcal{A}$ faults (it can not fault on u!), $\mathcal{A}^*$ also faults. $\mathcal{A}$ and $\mathcal{A}^*$ could replace the same page, as neither u nor v will be evicted. The cache difference between $\mathcal{A}$ and $\mathcal{A}^*$ will always be v/u until time t. At t, $\mathcal{A}$ evicts v and $\mathcal{A}^*$ evicts u so $\mathcal{A}$ and $\mathcal{A}^*$ now have the same cache entries, and behave the same afterward. $\mathcal{A}$ and $\mathcal{A}^*$ thus have the same number of page faults.

Time:	...	i	...	t	...	t_1	...	t_2	...
Request:	...	?	...	?	...	u	...	v	...
$\mathcal{A}^*$ Cache:	$---$	$A+u$	A + u	A	$---$	$-$	$---$	$----$	$---$
$\mathcal{A}$ Cache:	$---$	$A+v$	A + v	A	$---$	$-$	$---$	$----$	$---$

2. $t_1 < t \leq t_2$: Like case 1, before t_1, $\mathcal{A}$ and $\mathcal{A}^*$ differ only by v/u. At time t_1, $\mathcal{A}$ faults but $\mathcal{A}^*$ does not. Since $\mathcal{A}$ does not kick v out (it does that at t), it has to kick another page out. Assume the page is w, then $\mathcal{A}$ and $\mathcal{A}^*$ differ by v/w. Later when w is requested, $\mathcal{A}$ may fault again and kicks x out to accormodate w, yielding a cache difference v/x. This type of faults on $\mathcal{A}$ ($\mathcal{A}^*$ does not always fault on these requests!)could happen a number of times before t. Eventually, right before time t, assume the cache difference is v/z, we can kick out v in $\mathcal{A}$ and z in $\mathcal{A}^*$, and the two algorithms converge afterward. In this scenarior, $\mathcal{A}$ has at least one more fault than $\mathcal{A}^*$ (the fault on page u).

Time:	...	i	...	t_1	...	?	...	$t-1$	t	...	t_2	...
Request:	...	?	...	u	...	w	...	?	?	...	v	...
$\mathcal{A}^*$ Cache:	$---$	$A+u$	A + u	$A+w$	$A+w$	$A+x$	...	$A+z$	A	$----$	$---$	
$\mathcal{A}$ Cache:	$---$	$A+v$	A + v	$A+v$	$A+v$	$A+v$	...	$A+v$	A	$----$	$---$	

3. $t > t_2$: Before t_2, the behaviors of the 2 algorithms are the same as in case 2. We have concluded that $\mathcal{A}^*$ has at least one more fault. Assume that after $t_2 - 1$, the cache difference is v/z, then at

time t_2, v is requested, causing $\mathcal{A}^*$ to fault but not $\mathcal{A}$. At this time, we can replace z by v in $\mathcal{A}^*$. Afterward, there is no cache difference any more, so $\mathcal{A}$ and $\mathcal{A}^*$ behaves the same. Even though $\mathcal{A}^*$ takes 1 more page fault than $\mathcal{A}$ at t_2, the fault is more than compensated, adn $\mathcal{A}$ has already had at least 1 fault before.

Time:	...	i	...	t_1	...	?	...	$t_2 - 1$	t_2	...	t	...
Request:	...	?	...	u	...	w	...	?	v	...	?	...
$\mathcal{A}^*$ Cache:	$---$	A + u	A + u	A+w	$A+w$	$A+x$	...	$A+z$	A	$----$	$---$	
$\mathcal{A}$ Cache:	$---$	A + v	A + v	A+v	$A+v$	$A+v$	...	$A+v$	A	$----$	$---$	

We have shown that in all cases, $\mathcal{A}^*$ performs at least as well as $\mathcal{A}$. Thus LFD is an optimal off-line caching algorithm. ∎

13.3.3 Least Recently Used (LRU)

We now move on to the Least Recently Used (LRU) algorithm, and prove that it has a competitive ratio of k. Instead of proving the theorem directly, we will first look at a group of algorithms called "Marking Algorithms". We will prove that any algorithm in this group is k-competitive.

Lets divide an input sequence into "phases". A phase is the maximum length interval that has $\leq k$ new pages. For example, if there are 2 cache items ($k = 2$), sequence *abbabdeaab* has 3 phases shown below:

$$\text{Phases: } \underbrace{abbab}_{\text{Phase 1}}\ \underbrace{de}_{\text{Phase 2}}\ \underbrace{aab}_{\text{Phase 3}}$$

A marking algorithm works as follows: it "unmarks" all the cache entries at the beginning of a phase, and puts a marked new page into an arbitrary chosen unmarked slot in the cache, when we have a page fault in the phase. It unmarks all the pages when all the cache entries are marked. The corresponding code for this algorithm is given below:

```
Access(Page p)
    if (p is not in the cache) then
        if (all pages marked) then
            unmark all pages
        endif
        evict randomly selected unmarked page
        put p there
    endif
    mark p
end
```

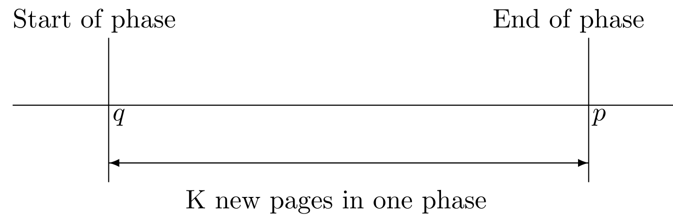
Following is an example of a marking algorithm: At the start of each phase, we clear all the cache. Each page fault then causes a new page to be brought into an empty entry in the cache, until all the cache entries are filled up. We then clear all the cache entries and start a new phase. The corresponding code for this algorithm is given below:

```
ACCESS(Page p)
    if (p is not in the cache) then
        if (cache is full) then
            clear all cache entries
        endif
        put p as next entry in cache
    endif
end
```

Theorem 13.2. *Any marking algorithm has a competitive ratio of k.*

Proof:

Suppose we have an off-line cache with only h entries, with $h \leq k$. We want to see how marking algorithm with k cache entries performs compared to off-line algorithms with limited cache size.



In each phase, there are k new pages, so marking algorithm will fault exactly k times. Suppose that at the starting time of a phase, the first page requested is q, and the first page requested in the next phase is p. When we enter the current phase, the off-line algorithm may or may not have q in the cache. In either case, however, we need a page to hold q during this current phase. So the number of free pages for holding other pages is only $h-1$, since the offline algorithm is assumed to have only h slots. Also notice that, counting p, we have a $k+1$ new pages. So the total number of offline page faults is at least $k-(h-1) = k-h+1$.

Thus the ratio between online marking algorithm with k cache entries and the best off-line algorithm with h cache entries is

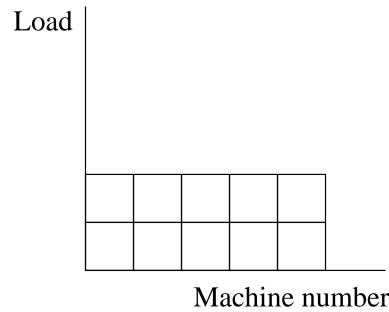
$$R \leq \frac{k}{k-h+1}.$$

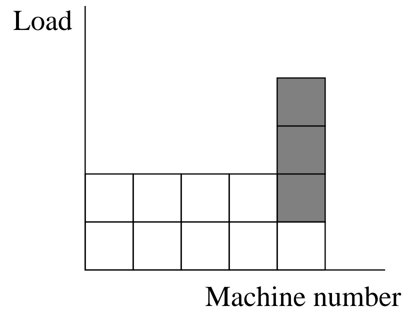
The maximum value of R is k when $k = h$. Thus marking algorithm is k-competitive. ∎

The result we just proved suggests that increasing cache size for online-algorithm could overcome its lack of knowledge about the future requests. For example, by doubling the size of the cache (i.e. using $2k$ instead of k-size cache), we can make online algoritm performs at the most 2 times worse than the off-line algorithm with k cache entries. Making k even larger could potentially make online algorithm performs as well as or even better than off-line algorithm with limited cache.

Theorem 13.2 can be used to show that LRU is k-competitive.

Theorem 13.3. *LRU has a competitive ratio of k.*





Proof: We will show that LRU in fact is a special case of marking algorithms. To see this, define a phase in LRU to start at the time that a cache entry p is brought into cache and end at the time that the entry is evicted. With LRU, p is never evicted before any other $k-1$ pages existing in cache at the start of the phase. So bring p into cache is equivalent to marking p and wait until all other pages are marked up to finally clear the cache. Since marking algorithm is k-competitive, we conclude that LRU is also k-competitive. ▮

13.4 Load Balancing

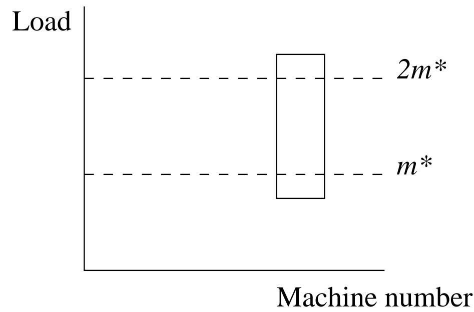
Problem Informally, we have some number of identical machines and a series of arriving jobs, each of which will continuously run and never stop. Each job uses some percentage of CPU and must be scheduled as soon as it comes in. The goal is to schedule the jobs in such a way as to minimize the maximum load at any given time.

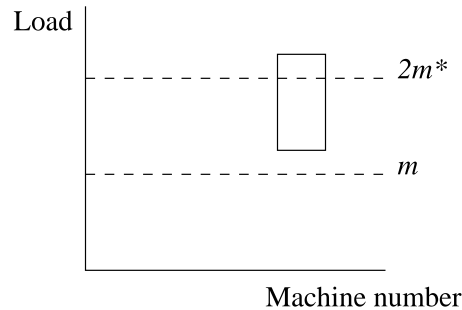
Clearly, if all the jobs are the same size, it is very easy to schedule them optimally.

However, if the jobs are different sizes, the scheduling task becomes more difficult. Suppose several small tasks arrive, followed by a larger task. If we first distribute the small tasks equally among the machines, the machine to which we assign the larger task will be much more loaded than the others.

We may formally state the problem as follows: We have m machines. At time i, we are given job j_i with load l_i, which needs to be scheduled immediately to some machine of our choice, $m(i)$. Let $M(t)$ be the load of the machine with maximum load under our scheduling, at time t. Let $M'(t)$ be the load of the machine with maximum load under an offline, optimal scheduling, at time t. We want to design an algorithm in such a way so as to minimize the maximum value of $M(t)/M'(t)$ for all t.

Solution We claim that it is possible for an online algorithm to achieve a competitive ratio $M(t)/M'(t) \leq 2$.





Algorithm: When job j_i comes in, schedule it to one of the machines with minimal load. If there is more than one machine with minimal load, schedule it to the one with minimal ID (this is to determine the scheduling deterministically).

Proof: (by contradiction)

Let $|m|$ = load of machine m. Let machine m' be the machine with maximum load. Let the (machine with) optimal maximum load (for an offline algorithm) be m^*. Assume for the sake of contradiction that $|m'| > 2 * |m^*|$. Let the last job assigned to machine m' have load l'.

We note the following:

- The situation pictured below is not possible, because no job can be larger than the maximum load of a single machine. Therefore, $l' \leq |m^*|$. Combining this with $|m'| > 2*|m^*|$ gives $|m'| - l' > |m^*|$
- Instead, we must have a situation such as pictured below. Since l' was scheduled to m', all other machines must have load at least $|m'| - l' > |m^*|$

Thus, all machines are scheduled at load greater than $|m^*|$, contradiction with the existence of a scheduling such that no load is greater than $|m^*|$. ∎

13.5 On-line Steiner trees

Recall that a Steiner tree is a spanning tree that only touches a specified subset of the nodes. Steiner trees have practical applications in reducing traffic on a large network by only routing multicast packets along the edges of the tree.

We wish to build a Steiner tree in an on-line fashion. Nodes (which we will call a subscribers) will announce that they wish to join the tree and we will build paths to them.

Algorithm: Attach each new node to the closest previous subscriber by the shortest path between the two.

Analysis:

Let S be the set of points for which we paid $\geq L$ each to attach.

Lemma 13.4. *The distance between any two points is $\geq L$.*

Proof: Suppose for two nodes $u, V \in S$, the distance between them is strictly less than L. Without loss of generality, assume that v came after u. u was an existing subscriber when v arrived. Then since the distance between them is $< L$, we should have paid strictly less than L to attach v to the existing tree. This is a contradiction. ∎

Lemma 13.5. *A Hamiltonian tour through S will take at least $|S| \cdot L$.*

Proof: Suppose the hamiltonian tour visits the vertices in the order $v_1, v_2, \ldots, v_|S|, v_1$. Then the distance along the tour in the segment between v_i and v_{i+1} must be at least the distance between v_i and v_{i+1} which is at least L. Hence the length of the hamiltonian tour is at least $|S| \cdot L$. ∎

Let OPT be the cost of the optimal Steiner tree for the set of requests.

Lemma 13.6. *There is a Hamiltonian tour of cost at most $2OPT$ that visits all the requests.*

Proof: Double all the edges of the optimal Steiner tree and consider an Eulerian walk of the graph obtained. ∎

Associate with each request, the cost incurred to connect it to the existing tree. Let L_i be the ith largest cost in this set.

Lemma 13.7. $L_i \leq 2 \cdot OPT/i$.

Proof: Consider the set S consisting of the i most expensive requests. Since we pay at least L_i for each of them, by Lemma 13.5 any Hamiltonian tour visiting all the requests of S much have length at least $|S|L_i = i \cdot L_i$. But from Lemma 13.6, there is a Hamiltonian tour of cost at most $2 \cdot OPT$ that visits all the requests. Clearly $i \cdot L_i \leq 2 \cdot OPT$. This implies $L_i \leq 2 \cdot OPT/i$. ∎

Theorem 13.8. *The online algorithm pays a cost of at most $O(\log n)OPT$, where n is the number of requests.*

Proof: The total cost paid by the online algorithm is

$$\sum_{i=1}^{n} L_i \leq \sum_{i=1}^{n} 2 \cdot \frac{OPT}{i} = O(\log n)OPT$$

∎

References

[1] N. Alon and J. Spencer. *The Probabilistic Method*. John Wiley, 2000. Second Edition.

[2] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An improved lp-based approximation for steiner tree. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 583–592, New York, NY, USA, 2010. ACM.

[3] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *The ACM Journal of Experimental Algorithmics*, 3(8), 1998. http://www.jea.acm.org/1998/CherkasskyAugment/.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2009. 3rd edition.

[5] M. X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.

[6] A. Goldberg and R. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[7] D. Hochbaum. *Approximation Algorithms for NP-hard problems*. PWS Publishing Company.

[8] L. Kou, G. Markowsky, and L.Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15:141–145, 1981.

[9] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and eds. D.B. Shmoys. *The Traveling Salesman Problem*. John Wiley, 1985.

[10] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge, 2005.

[11] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[12] P. Raghavanand and C.D. Thompson. Randomized rounding. *Combinatorica*, 1987.

[13] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. In *Proc.11th ACM-SIAM Symposium on Discrete Algorithms*, pages 770–779, 2000.

[14] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[15] Vijay Vazirani. *Approximation Algorithms*. Springer, 2003.