COMBINATORIAL OPTIMIZATION
LECTURE NOTES FOR CS363/OR349

Andrew V. Goldberg

Supported in part by NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T, DEC, and 3M, ONR Young Investigator Award N00014-91-J-1855

Contents

Preface

This is a set of lecture notes for *CS363/OR349: Combinatorial Optimization* that I taught in Winter 1993. The notes were scribed by students in the class and reviewed me. These notes are not ment to be in the polished form, and probably contain some errors and ommisions. In particular, some references to the literature are missing. The reader is directed to the relevant books and research papers for a formal treatment of the material.

These notes are intended as an aid in teaching and for studying advanced work in the area. Much of the material covered in these notes is very recent and is available in recearch papers only. I plan to update the notes each time the course is tought. I would like to know of any errors and ommisions so that they can be corrected in the future.

I would like to thank the students who took the course and scribed. I would also like to thank my teaching assistants, Sherry Listgarten and Robert Kennedy, for their help with the class in general and with these notes in particular.

LECTURE I

The Stable Marriage and Stable Roommates Problems

Scribe: Jeffery D. Oldham

The stable marriage and network stability problems introduce combinatorial optimization, which involves discrete, as opposed to continuous, optimization.

1. The Stable Marriage Problem

1.1. Definition of the Stable Marriage Problem. The *stable marriage problem* consists of matching members of two different sets according to the members' preferences for the other set's members. For example, medical school students serve as residents at hospitals after graduation. Under the National Intern Matching Program which matches residents (interns) and hospitals, the students submit rankings of preferred hospitals, while hospitals submit rankings of preferred students. Using the rankings, each student is assigned to one hospital [22].

The input to the stable marriage problem consists of:

- a set M of n males,
- a set F of n females,
- for each male and female, a list of all the members of the opposite gender in order of preference.

Note that the size of the input is $\Theta(n^2)$, since each of the $2n$ preference lists has length n.

A *marriage* is a one-to-one mapping between males and females. We want to find a *stable marriage*, *i.e.*, one in which there is no pair mf such that f prefers m to her current partner and m prefers f over his current partner. Two *partners* are *stable* if they are matched in some stable marriage.

EXAMPLE 1. *Consider the following preference lists for the sets* $M = \{a, b, c\}$ *and* $F = \{A, B, C\}$.

a: ABC	A: bac
b: ABC	B: acb
c: BCA	C: abc

Male a prefers female A over females B and C, and female A prefers male b. The marriage $\{aC, bB, cA\}$ *is not stable because a prefers B over his current partner C and B prefers a over her current partner b. The marriage* $\{aB, bA, cC\}$ *is stable.*

1.2. The Gale-Shapley Algorithm. The Gale-Shapley algorithm [13], given preference lists for both males and females, returns a stable marriage by having any unmatched male propose to females in the order of his preference lists until a female accepts. (See Figure I.1.) A female receiving a proposal accepts if she is not matched or she prefers the proposer over her current partner. In the latter case, the partner becomes single.

while there exists an unmarried male m **do**
    m proposes to the first female f on his preference list to which he has not proposed
    if f is unmarried **then** match m and f
    else if f prefers m to her current match m' **then**
        make m' single
        match m and f
    else m remains single

FIGURE I.1. Gale-Shapley Algorithm.

EXAMPLE 2. *Suppose we desire a stable marriage for the preference lists*

a: BAC	A: acb
b: BAC	B: bac
c: ACB	C: cab

The algorithm yields the stable marriage $\{aA, bB, cC\}$ *by iterating these steps.*

(1) *a proposes to B, the first female on his preference list, and she accepts because she is single. The set of matches is* $\{aB\}$.
(2) *b proposes to B, the first female on his preference list, and she accepts, breaking her match with a, because she prefers b over a. The set of matches is* $\{bB\}$.
(3) *Since a is now unmatched, a proposes to A, the next female on his list. With A accepting because she is not matched, the set of matches is* $\{aA, bB\}$.
(4) *Since c is the only unmatched male, c proposes to A, the first female on his preference list. She rejects the proposed match because she prefers her current partner a. The set of matches is* $\{aA, bB\}$.
(5) *c proposes to C, who accepts because she is not matched. Since all males are now matched, the algorithm terminates with matches* $\{aA, bB, cC\}$.

The order of choosing unmatched males does not affect the algorithm's result.

It is easy to see that the algorithm takes $O(n^2)$ time.

1.3. Correctness. We now prove the algorithm terminates, yielding a marriage. The algorithm terminates if all males get married. Because a marriage is a one-to-one function between males and females, all females will then also be married. Suppose a male is not matched. Then a female is not matched. Since the male's preference list contained all the females, he proposed to her during the algorithm's execution. Since the algorithm guarantees matched females remain matched, she is matched. $\Rightarrow\Leftarrow$ All males are matched, and the algorithm terminates.

The resulting marriage M is stable. Suppose the marriage is not stable because male m and female f prefer each other to their current partners $M(m)$ and $M^{-1}(f)$. Since m prefers f over $M(m)$, he proposed to her before proposing to $M(m)$. Female f rejected m at some point because m married $M(m)$, not f. Since a female's partner only becomes more preferable, f does not prefer m over $M^{-1}(f)$. $\Rightarrow\Leftarrow$ The marriage is stable.

1.4. Male Optimality. Now we show that the stable marriage produced by the Gale-Shapley algorithm is *male-optimal*. That is, there is no stable marriage in which any male matches a female he prefers more than the one assigned in this stable marriage.

Let stable marriage M' have a match mf', when the algorithm yields stable marriage M with match mf. Assume m prefers $f' \neq f$ to f, so the result M of the Gale-Shapley algorithm is not male-optimal. The preference list for m has the form $\ldots f' \ldots f \ldots$. Because m prefers f' to f and $mf' \notin M$, f' rejected m for another male $m' \neq m$ during the execution of the Gale-Shapley algorithm. Without loss of generality, assume this rejection is the first rejection of a stable partner. Male m' has no stable partner whom he prefers to f' because f''s rejection of m was the first rejection of a stable partner. Since f' matched m, not m' in marriage M', m''s spouse $M'(m') \neq f'$, and his preference list has the form $\ldots f' \ldots M'(m') \ldots$. $\Rightarrow\Leftarrow$ Marriage M' is not stable because m' and f' prefer each other to their current partners. m' prefers f' over his spouse $M'(m')$, and f' rejected m for m'.

The male-optimal marriage is unique since two marriages must differ and some male must do worse in one of the two. Consequently, the algorithm always produces the same result regardless of the order in which the unmarried males are chosen.

In a *female-pessimal* stable marriage, each female has the worst partner she can have in any stable marriage.

Lemma 1. *The Gale-Shapley algorithm yields female-pessimal stable marriages.*

Proof. Assume marriage M be the male-optimal stable marriage, and let marriage $M' \neq M$ be a female-pessimal stable marriage. Suppose female f matches male m in marriage M and m' in M'. Because M is male-optimal, m prefers f over all other spouses possible in any stable marriage, while, because M' is female-pessimal, the preference list for f has the form $\ldots m \ldots m' \ldots$. Marriage M' is not stable because m prefers f to his spouse $M'(m) \neq f$ and f prefers m to her spouse. ■

2. Introduction to Network Stability

After introducing the stable roommates problem, we reduce it to a network stability problem, using ideas from Stanford graduate Ashok Subramanian [42].

2.1. The Stable Roommates Problem. The *stable roommates problem*, also called the *stable matching problem*, generalizes the stable marriage problem to matches within the same set and partial preference lists. The problem inputs consists of:

- a set S and
- for every member of S, a partial preference list.

Each set member is matched with another member so the matches are stable. Since the *preference lists* are *partial*, one member's preference list need not contain every other set member. A matching may be *unstable* in three ways:

- two unmatched members may be acceptable to each other,
- a matched member may prefer an unmatched member over its current partner, or
- two matched members may prefer each other over their partners.

A matching is stable if it is not unstable. The stable marriage problem may be reduced to the stable roommate problem by defining S as the union of the male and female sets and using the same preference lists.

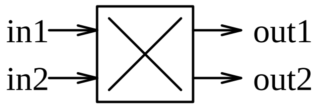
Given a problem instance, a stable matching does not necessarily exist.

EXERCISE 1. *Give an example of a stable matching problem with $|S| = 4$ and complete preference lists that has no solution.*

2.2. Network Definitions. A *network* is a directed graph with vertices representing gates and arcs representing wires. Incoming and outgoing arcs are ordered. Each *gate* has a type and an associated equation using incoming arcs. For example, an **and** gate g_0 may have equation $g_1 \wedge g_2$, where g_1 and g_2 are gates with arcs pointing to g_0. An *input gate* is a node with in-degree zero and out-degree one. An *output gate* is a node with in-degree one and out-degree zero. At most one output of an *adjacency preserving* gate changes if one input changes. For example, **not** gates are adjacency preserving.

A *configuration*, an assignment of truth values to every arc, is *stable* if all the gate equations are satisfied. Note configurations, *i.e.*, assignments, not networks, are stable. Unstable networks exists: consider a **not** gate with its output leading to its input.

An *X-gate* has two inputs and two outputs. It returns $(0,0)$ if both inputs match and its inputs otherwise, *i.e.*, $\text{out1} = \text{in1} \cdot \overline{\text{in2}}$ and $\text{out2} = \overline{\text{in1}} \cdot \text{in2}$. See Figure I.2. X-gates are adjacency preserving. The networks we consider will contain only X, input, and output gates.



in1, in2	0, 0	0, 1	1, 0	1, 1
out1, out2	0, 0	0, 1	1, 0	0, 0

FIGURE I.2. The X-gate.

A *path* in a network containing X-gates is a sequence of distinct gates connected by arcs where, if an X-gate is entered using the gate's first input, the arc from the gate's first output is used. Similarly, if an X-gate is entered using the gate's second input, the arc from the gate's second output is used. A *cycle* is a path where the first and last gates are the same. A *snake* is a maximal path, *i.e.*, a cycle or a path from an input to an output.

THEOREM 1. *An Xgate network configuration with no cyclic snakes and all inputs set to 1 is stable iff the following three conditions hold:*

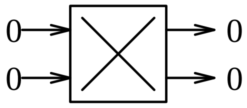
(1) *the values along the snakes form a sequence of 1's followed by a (possibly empty) sequence of 0's,*
(2) *snakes drop (change values from 1 to 0) in pairs, and*
(3) *if two snakes with values of 1 meet, they both drop.*

Proof.

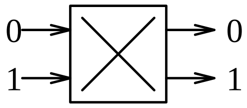
$\Rightarrow$ The three conditions follow directly from the definition of an X-gate.

(1) Because the inputs are set to 1, the snake values start at 1. Inspecting the truth table for X-gates reveals an X-gate's output is 0 whenever its input is 0.
(2) One snake drops only when both inputs are 1. Again using the X-gate truth table, both outputs become 0.
(3) Use the X-gate truth table. $\triangle$

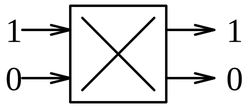
$\Leftarrow$ Using the three conditions, we show the X-gate's outputs match its truth table for any possible input. Since all possible network gate equations are satisfied, the configuration is stable. See Figure I.3. ∎



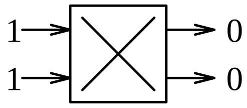
Using the first condition, the outputs must be 0.



Using the first condition, the first output must be 0.
The second output must be 1 using the second condition.



This case is symmetric with the previous case.



The third condition forces both outputs to drop.

FIGURE I.3. The Three Conditions of Theorem 1 Imply Stability.

2.3. The Stable Roommate Problem and Network Stability. We reduce the stable roommate problem to the network stability problem using a network of input, X-, and output gates. Before reducing, we preprocess the (partial) preference lists so one person will accept another as a roommate only if the latter will also accept the former as a roommate. If person a's preference list contains person b, but person b's preference list does not contain person a, b is removed from a's list.

To reduce a stable roommate instance to a network stability instance,

(1) Create an X-gate $\{a, b\}$ for each pair $\{a, b\}$ of mutually acceptable roommates.
(2) Create an input a for each $a \in S$.
(3) Create an output a for each $a \in S$.
(4) For each member a in S, wire from the member's input to the X-gate for a and its most preferred roommate. Using the output from this X-gate, wire to the X-gate for a and its next most preferred roommate, and so on. End the snake at a's output.
(5) Assign all inputs to 1.

EXAMPLE 3. *We construct a network for the stable marriage instance in example* 1.

a: ABC	A: bac
b: ABC	B: acb
c: BCA	C: abc

See Figure I.4.

In the next lecture, we will show stable configurations correspond to stable matchings.

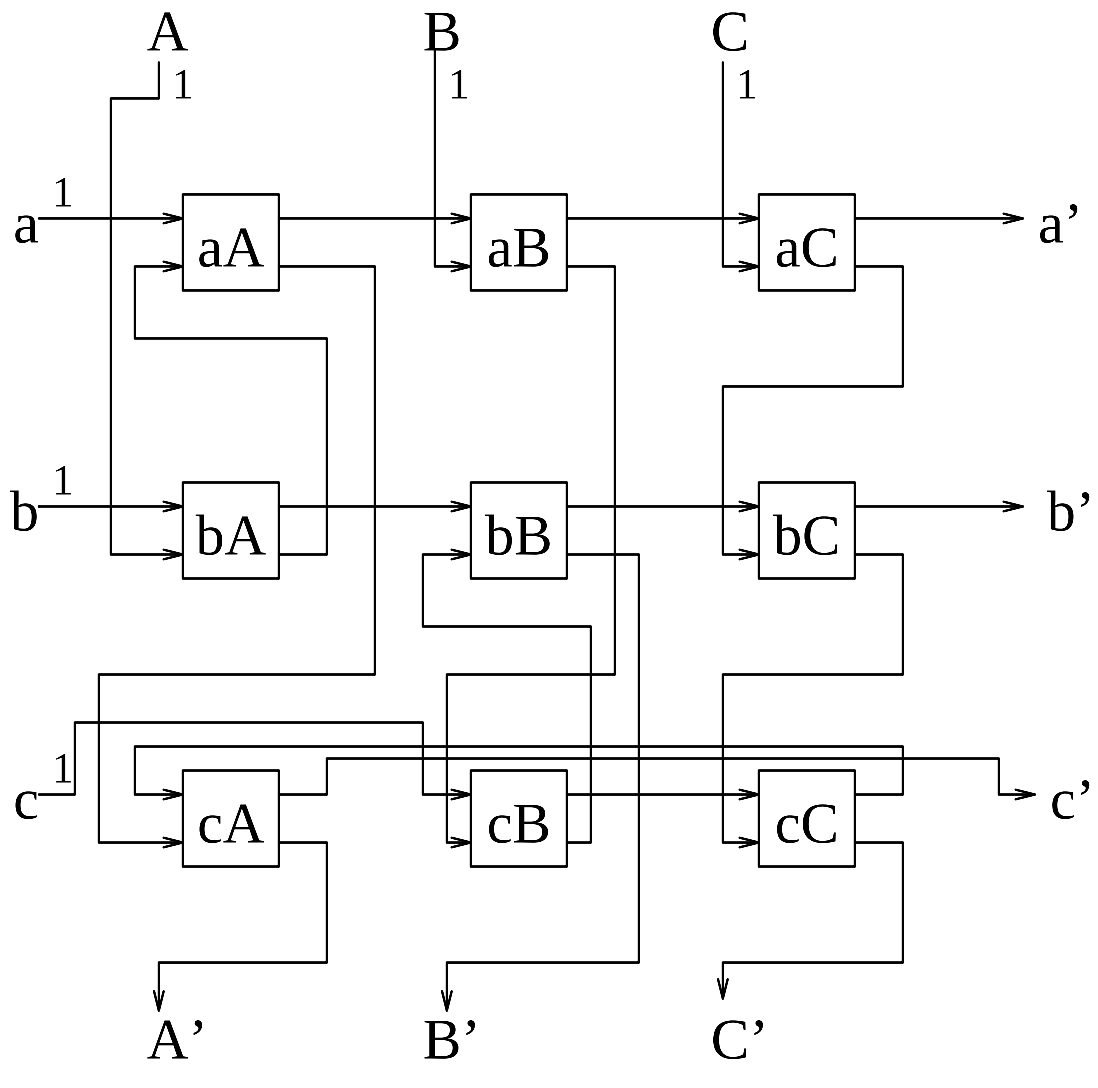


FIGURE I.4. An X-Network for a Stable Marriage Instance.

LECTURE II

The Stable Roommates Problem and Network Stability

Scribe: Jeffrey D. Oldham

Today's lecture shows a one-to-one correspondence between the stable roommates problem and stable configurations of a subset of X-networks. Using the correspondence between the problems and a linear time algorithm testing for stability of these networks, stable matchings are found. Most of the material presented can be found in the technical report [42], while the conference article [32] discusses complexity issues.

1. The Stable Roommates Problem and X-Network Stability

The correspondence between a stable roommates instance and an X-network is defined in the previous handout. After proving the correspondence is one-to-one, we will show how to determine the stability of the X-network and, thus, solve the stable roommates problem.

THEOREM 2. *If X-network N corresponds to stable roommates instance I, there is a one-to-one correspondence between the stable matchings in I and stable configurations in N.*

Proof.
$\Rightarrow$ Using the stable matching M, we construct a configuration of snakes. If a and b are matched, the snakes for a and b drop at X-gate ab. Use the snake theorem to show the configuration is stable. Each snake consists of 1's followed by 0's. Snakes drop only if two members are matched, so they drop in pairs. To show the third condition holds, assume two snakes for members x and y cross but do not drop. By the construction of the network where snakes follow the order of a member's preference list, they prefer each other to their partners (if these exist) so the matching is not stable. $\Rightarrow\Leftarrow$ $\triangle$

$\Leftarrow$ To map the stable configuration into a stable matching, we match a and b if and only if the two corresponding snakes drop at the same gate. To show the resulting matching is stable, we show all unmatched pairs a and b are not stable. First, assume a and b accept each other. Since a and b are not matched and the snake theorem's third condition guarantees both snakes drop if both inputs to the X-gate ab are 1, at least one input must have value 0. Since snakes only drop at matches, a or b must be matched earlier and a and b are unstable. Second, if a and b do not accept each other, their pairing is unstable. ■

2. Adjacency-Preserving Circuits and X-Networks

If one input of an *adjacency-preserving gate* changes, at most one output changes. If one input of an *adjacency-preserving circuit* changes, at most one output changes. If a circuit, e.g., an X-network, contains only adjacency-preserving gates, the circuit is adjacency-preserving. Copy gates are not adjacency-preserving.

LEMMA 2. *The number of inputs to an X-network equals the number of outputs.*

Proof. Define a node's weight to be the number of incoming arcs minus the number of outgoing arcs. Because every X-network arc connects two gates, it contributes zero to the network's weight, which is, thus, zero. X and **not** gates also have zero weight, while input and output gates have weights -1 and $+1$. The claim follows. ■

3. Network Stability and Simplification

Given an X-network with all inputs initially specified, we wish to determine if there exists a stable configuration. Before presenting the linear time algorithm, we present a network simplification process.

Given an input gate preceding another gate, we replace the two gates with simpler equivalent gates. See Figure II.5.

The simplification rules guarantee the following properties:

(1) A simplified network has a stable configuration iff the original network has a stable configuration.
(2) The simplified network has fewer arcs then the original network.
(3) If all inputs of the original network have assigned values, then the same holds for inputs of the simplified network.
(4) If an X-network has an assigned input, then a simplification rule applies.

Given an X-network with all inputs initially specified, network stability is determined by the following algorithm.

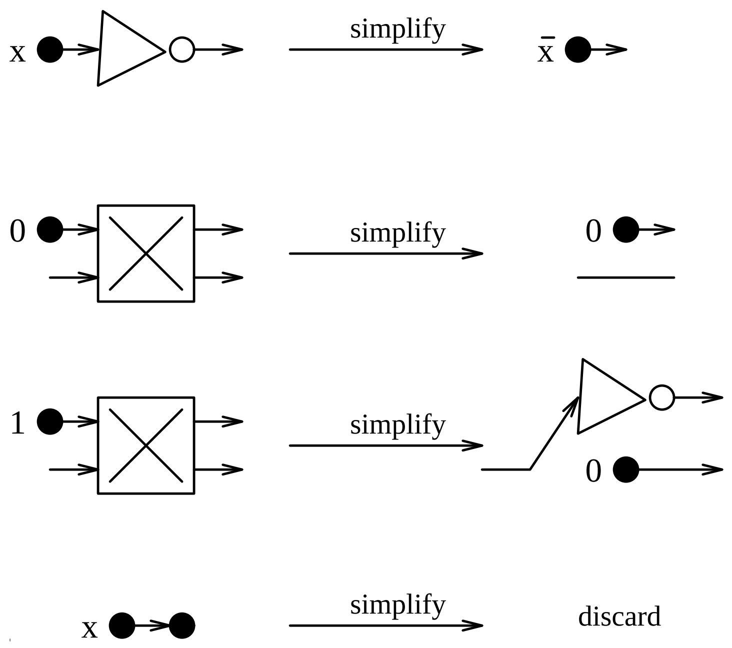


FIGURE II.5. The network simplification rules. The input value x may be either 0 or 1.

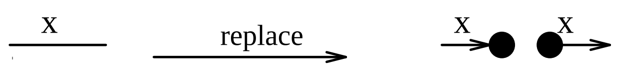


FIGURE II.6. Replacing an arc with input and output gates.

First, all inputs are eliminated by repeatedly applying the simplification rules until the network has no inputs and therefore no outputs. This process takes linear time since each simplification rule decreases the number of arcs.

The simplified network N, having no inputs and no outputs, has only cycles. After forming a new network N' by replacing an arc with input and output gates (see Figure II.6), assign the value 0 to the input gate. If, after further simplifying the network, the output has value 0, we could assign the arc value 0 in the network N. Otherwise, try assigning 1 to the input gate in network N'. If the output has value 1, we could assign the arc value 1 in N. Otherwise, we cannot assign either 0 or 1 to the arc in N and have an unstable configuration. Continue replacing arcs until the the network becomes empty or an unstable configuration is discovered.

To ensure the algorithm executes in linear time, use the standard trick of simplifying the networks with 0 and 1 assigned to the input gate in parallel, stopping when one simplification succeeds. If one of the computations succeeds, the amount of work done by the successful computation is linear in the number of arcs removed by this computation, and the amount of work done by the other computation is at most that of the successful one. If both of the computations fail, the amount of work done by each one is linear in the circuit size and the algorithm terminates.

To determine a stable configuration of the original network, reverse the simplification process, assigning arcs their stable values.

We claim the algorithm is correct. Because the simplification steps preserve stability, we only need to show arc replacement preserves network stability in networks with only X and **not** gates.

LEMMA 3. *Suppose X-network N has no input or output gates but does have a stable configuration. Then assigning the boolean value x to arc e in the algorithm succeeds if and only if N has a stable configuration with arc e assigned x.*

Proof. $\Rightarrow$ We need only consider a stable configuration of network N where arc e is assigned $\bar{x}$. If we cut e and assign the new input and output to $\bar{x}$, the configuration remains stable. Suppose we change the value on the input to x. By assumption, we know that this forces the output value to change to x. Because the X-network gates are adjacency-preserving, the only arc values affected are those on the path from the input to the output. Thus, the resulting configuration is stable. $\triangle$

$\Leftarrow$ Again, use the adjacency-preserving property. ■

COROLLARY 1. *For a stable roommate problem instance, the set of unmatched members is the same for any stable matching.*

Proof. After eliminating all the input gates in the X-network N corresponding to the instance, all the output gates are also eliminated. The output gates' values are completely determined. An unmatched member has output gate value 1, and a matched one has the value of 0. ■

LECTURE III

The Maximum Flow Problem

Scribe: Sanjeev Khanna

1. Network Flows

1.1. Flows and Pseudo-flows. A *network* $G = (V, E)$ is a directed graph with two distinguished nodes, namely a *source* node s and a *sink* node t. A nonnegative capacity $u(v, w)$ is associated with every arc $(v, w) \in E$. If $(v, w) \notin E$, we define $u(v, w) = 0$. We use n and m to denote $|V|$ and $|E|$ respectively.

A *flow* on a network G is a real-valued function $f : V \times V \to \Re$ which satisfies the following three properties :

- Antisymmetry: $f(v, w) = -f(w, v)\ \ \forall v, w \in V$.
  If $f(v, w) > 0$, we say there is a flow from v to w.
- Capacity constraints: $f(v, w) \leq\ u(v, w)\ \ \forall v, w \in V$.
- Flow conservation: $\sum_{w \in V} f(v, w) = 0\ \ \forall v \in V - s, t$.
  If $f(v, w) = u(v, w)$ we say the flow *saturates* (v, w).

A function f that satisfies only the first two properties above is called a *pseudo-flow*. The value $|f|$ of a flow f is the net flow out of the source, $\sum_{v \in V} f(s, v)$ or equivalently, the net flow into the sink, $\sum_{v \in V} f(v, t)$. A classic network optimization problem is the *maximum flow* problem which is the problem of finding a flow of maximum value in a given network.

1.2. Cuts. We define a cut (S, T) to be a partition of the node set V into two parts S and T, such that S contains s and T contains t. The *capacity of a cut* (S, T) is given by $u(S, T) = \sum_{v \in S, w \in T} u(v, w)$. A *minimum cut* is a cut of minimum capacity. The *flow through a cut* (S, T) is defined as $f(S, T) = \sum_{v \in S, w \in T} f(v, w)$.

LEMMA 4. *For any flow f, the flow across any cut (S, T) is equal to the flow value.*

Proof. $f(S,T) = \sum_{v\in S,w\in T} f(v,w) = \sum_{v\in S,w\in V} f(v,w) - \sum_{v\in S,w\in S} f(v,w) = |f| - 0 = |f|$ since $\sum_{v\in S,w\in V} f(v,w) = \sum_{w\in V} f(s,w)$ by flow conservation and $\sum_{v\in S,w\in S} f(v,w) = 0$ by antisymmetry. ■

An immediate application of Lemma 4 is as follows.

LEMMA 5. *For any flow f and cut (S,T), $|f| \le u(S,T)$.*

Proof. $|f| = f(S,T) = \sum_{v\in S,w\in T} f(v,w) \le \sum_{v\in S,w\in T} u(v,w) = u(S,T)$. ■

Thus the value of a maximum flow is no greater than the capacity of the minimum cut. We later prove a theorem which states that in fact these two numbers are always equal.

1.3. Residual Networks. Given a flow f on G, we define the *residual capacity function* $u_f : V \times V \to \Re$ as $u_f(v,w) = u(v,w) - f(v,w)$. The *residual network* $G_f = (V, E_f)$ for a flow f is the graph with node set V, source s, sink t, and an arc (v,w) of a capacity $u_f(v,w)$ for every pair (v,w) such that $u_f(v,w) > 0$. A network flow and the corresponding residual network are depicted in Figure III.7. Only arcs with non-zero capacities are shown in the figure.

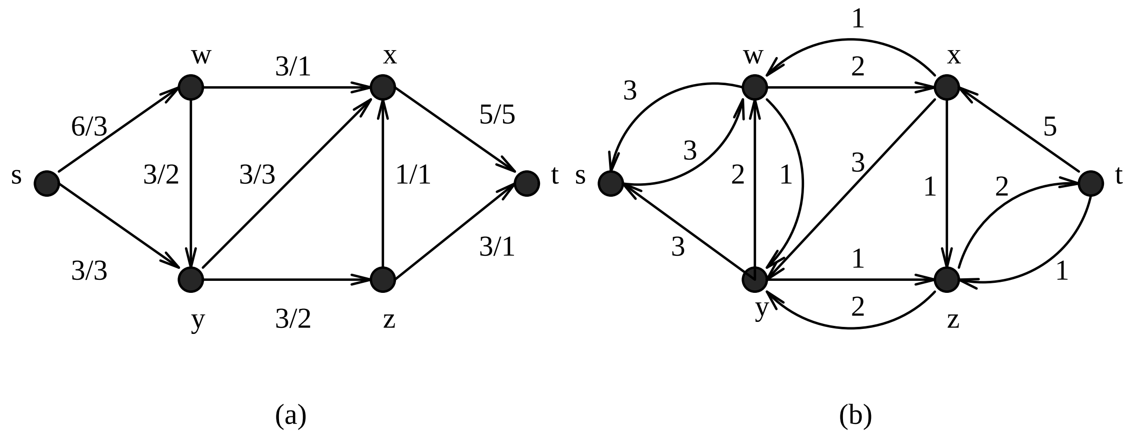


FIGURE III.7. **(a)** The flow f on a network G. **(b)** The residual network G_f.

The following exercise shows how the flow in a residual network relates to the flow in the original network.

EXERCISE 2. *Let f be a flow in G and let G_f be its residual graph. Show the following :*
(a) A function f' is a flow in G_f iff $f + f'$ is a flow in G.
(b) A function f' is a maximum flow in G_f iff $f + f'$ is a maximum flow in G.
(c) The flow sum $f + f'$ in G has value $|f + f'| = |f| + |f'|$.

Thus we know that if f is any flow in G and f^* is a maximum flow in G, then the value of maximum flow in G_f is $|f^*| - |f|$.

1.4. Augmenting Paths. Given a flow f on G, an *augmenting path* is a directed path from s to t in the residual graph G_f. The *residual capacity of an augmenting path* is defined to be the smallest residual capacity of any arc on that path. The path $[s, w, y, z, t]$ in Figure III.7(b) is an augmenting path of residual capacity 1. We are now ready to state and prove a fundamental theorem about network flows.

THEOREM 3. ***(Max-flow min-cut theorem)*** *The following three statements are equivalent:*

(1) *f is a maximum flow,*
(2) *there is no augmenting path for f,*
(3) *$|f| = u(S, T)$ for some cut (S, T) of G.*

Proof.

(1) $\Rightarrow$ (2): If there is an augmenting path P for f, then we can increase the flow value by increasing the flow along P.

(2) $\Rightarrow$ (3): Suppose there is no augmenting path for f. Let S be the set of nodes reachable from s and let $T = V - S$. Since $s \in S$ and $t \in T$, (S, T) is a cut. We further observe that all arcs from S to T must be saturated and thus we have

$$|f| = \sum_{v \in S, w \in T} f(v, w) = \sum_{v \in S, w \in T} u(S, T).$$

(3) $\Rightarrow$ (1): This follows immediately from Lemma 5. ∎

2. The Augmenting Path Algorithm

The Max-flow min-cut theorem forms basis of a simple iterative algorithm, called the augmenting path algorithm. The Max-flow Min-cut theorem and the augmenting path algorithm, are both due to Ford and Fulkerson [11, 12]. Figure III.8 describes the augmenting path algorithm.

```
for each arc (v, w) ∈ E do
     f(v, w) ← 0;
     f(w, v) ← 0;

While ∃ a path P from s to t in the residual network G_f do
     δ ← min_{a∈P}{u_f(a)};
     for each arc (v, w) ∈ P do
          f(v, w) ← f(v, w) + δ;
          f(w, v) ← −f(v, w);
```

FIGURE III.8. The Augmenting Path Algorithm.

THEOREM 4. *The augmenting path algorithm is correct and if all arc capacities are integral, it runs in $O(m^2U)$ time, where m is the number of arcs and U is the maximum capacity of any arc.*

Proof. The algorithm terminates only when there are no more augmenting paths and thus the correctness of the algorithm follows by the Max-flow min-cut theorem. Since the arc capacities are integral, the flow increases by at least one unit at each augmenting step. We also observe that the maximum flow in the network is bounded by mU. These two observations combined with the fact that an augmenting path can be determined in $O(m)$ time (using a graph search procedure like breadth-first or depth-first search), give us the stated result. ■

Thus the total time taken by the augmenting path algorithm is bounded by an exponential function in the input size. The algorithm can take a really large number of iterations on certain problem instances with large flow values. For example, in the network shown in Figure III.9, we can alternate between augmenting paths $[s, x, y, t]$ and $[s, y, x, t]$ and thus take two thousand iterations to produce the maximum flow.

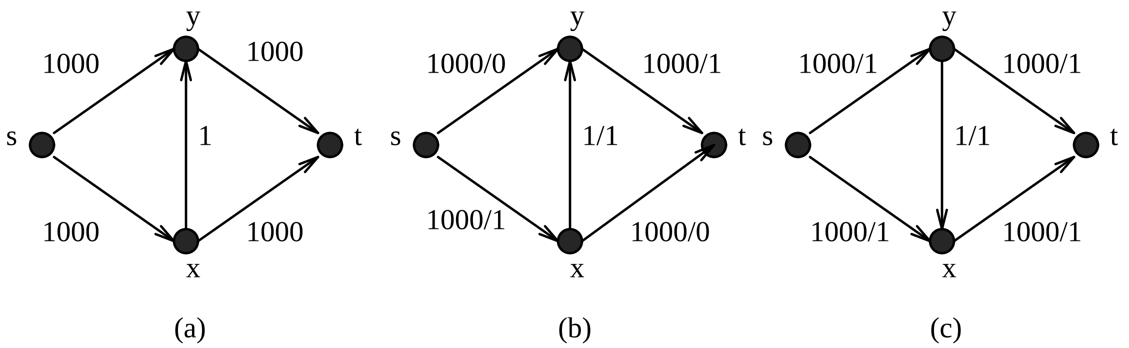


FIGURE III.9. **(a)** The initial network. **(b)** After augmentation along $[s, x, y, t]$. **(c)** After augmentation along $[s, y, x, t]$.

The following exercise from [27] shows that if the arc capacities are irrational, then the process of augmenting along arbitrary paths, may take infinite time to converge to the maximum flow.

EXERCISE 3. *Let r be the positive root of the quadratic equation $x^2 + x - 1$. Consider the network shown in Figure* III.10.

The arc capacity of all the unlabeled arcs is $r + 2$. Clearly, the value of the maximum flow is $1 + r + r^2 = 2$. Show that there exists a sequence of augmentations such that it takes infinite time for the flow value to converge to 2.

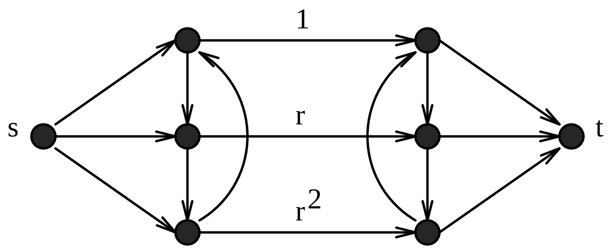


FIGURE III.10. A network with irrational arc capacities on which the augmenting path algorithm may take infinite time.

A useful property of the augmenting path algorithm is that when run on a network with integral arc capacities, it always produces a maximum flow which assigns integer flows to each arc. Such a flow is called an *integral flow.* Hence we have the following theorem:

THEOREM 5. ***(Integrality theorem)*** *There always exists an integral maximum flow on a network with integral arc capacities.*

3. The Decomposition Theorem

The Decomposition theorem states that every flow can be decomposed into a relatively small number of primitive elements of the following kind :

(1) Paths P from s to t, with flow δ: (P, δ).
(2) Cycles Γ, with flow δ: (Γ, δ).

The nodes in the primitive elements satisfy the conservation constraints, except possibly s and t. An example of flow decomposition is shown in Figure III.11 where the flow on a network is decomposed into three primitive elements, two paths and one cycle.

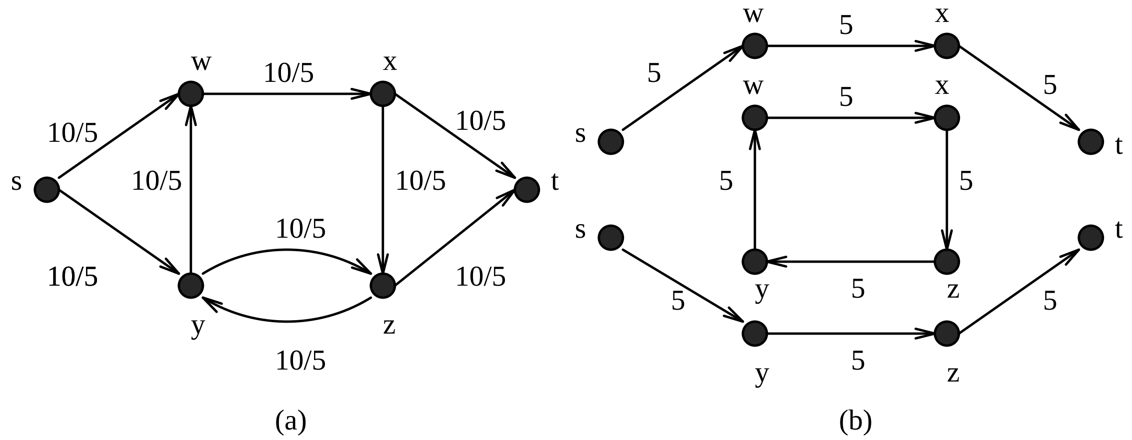


FIGURE III.11. An example of flow decomposition. **(a)** A flow on a network. **(b)** The primitive elements of the flow.

We can state the following theorem for the decomposition of the flow in any network into primitive elements.

THEOREM 6. ***(Decomposition theorem)*** *Any flow can be decomposed into at most m primitive elements.*

Proof. Let f be a flow function on a network $G = (V, E)$ and let $A = \{a \in E : f(a) > 0\}$. We denote by G^* the subgraph of G induced by the arcs in A. Repeatedly perform the following step until there is no path from s to t in G^* :

> Find a path P from s to t in G^* and set $\delta = \min_{a \in P}\{f(a)\}$. Reduce the flow on each $a \in P$ by the amount δ. The flow on at least one of the arcs is reduced to zero. Remove all arcs with zero flow and add (P, δ) to the set of the primitive elements.

At the end of this process, the resulting graph, say G^{**}, has no path from s to t and thus the net flow from s to t is zero in G^{**}. Note that at this point flow conservation holds at every node, including s and t.

Next we repeat the following step to find cycles in G^{**} until there are no more arcs left in G^{**}:

> Select (v, w) in G^{**}. Due to flow conservation, there must be an outgoing arc from w in G^{**}. Successively keep picking arcs with positive flow till we find a cycle, say Γ. Let $\delta = \min_{a \in \Gamma}\{f(a)\}$. Reduce flow on each arc along the cycle by the amount δ and now remove all arcs on which the flow has become zero. Add (Γ, δ) to the set of primitive elements.

This works because flow conservation holds at every node, so if there is an incoming arc (u, v) with positive flow, there must be an outgoing arc (v, w) with positive flow.

It is clear that each time we add a primitive element, we reduce the number of arcs by at least one. Thus we construct a set of at most m primitive elements which together define the given flow f on G. ∎

We have already seen that even on networks with integral arc capacities, augmenting along arbitrarily chosen paths may take a very long time to converge to maximum flow. Edmonds and Karp [8] suggested to always augment along the path of the maximum residual capacity. We now show how the Decomposition theorem can be used to analyze this heuristic (follows the presentation of [27]).

THEOREM 7. *When the arc capacities are integral, augmenting along paths of the maximum residual capacity produces a maximum flow, say f^*, in at most $O(m \log |f^*|)$ augmenting steps.*

Proof. By the Decomposition theorem, there must be a path from s to t of value at least $|f^*|/m$. Thus augmenting along a path of maximum residual capacity must increase the value of the flow by this amount. So the value of the maximum flow in the residual graph is at most $|f^*| - |f^*|/m = |f^*|(\frac{m-1}{m})$.

After k augmenting steps, therefore, the value of the maximum flow in the residual graph is no more than $|f^*|(\frac{m-1}{m})^k$. Since the arc capacities are integral, the total number of augmenting steps required is bounded by the least number k such that $|f^*|(\frac{m-1}{m})^k < 1$. Using the estimate that $\log m - \log(m-1) = \Theta(\frac{1}{m})$, we get the stated result. ■

The following exercise derives an alternative bound on the number of augmenting steps needed by the Edmonds and Karp heuristic.

EXERCISE 4. *Use the Decomposition theorem to show that when the arc capacities are integral, then augmenting along paths of the maximum residual capacity produces the maximum flow in at most $O(m \log |U|)$ augmenting steps where U denotes the largest capacity of any arc in the network.*

A path of maximum residual capacity can be found by modifying Dijkstra's algorithm.

EXERCISE 5. *Design and analyze an efficient algorithm to find an augmenting path of the maximum residual capacity in a residual network.*

4. The Capacity Scaling Algorithm

We now present a somewhat different approach to finding a maximum flow in a network. This approach, which is known as *capacity scaling*, essentially involves iteratively obtaining solutions to successively better approximations to the original problem instance. This technique is also due to Edmonds and Karp [8] and independently due to Dinitz [46]. We need to develop some notation before we describe the algorithm.

As before, we let U denote the maximum capacity of any arc in the network. We interpret the capacity of each arc as a $\lceil \log(U+1) \rceil$-bit binary number. Let $u_i(a)$ denote the i most significant bits of the binary representation of the capacity of arc a. We denote by P_i the network G with the capacity function u replaced by u_i. Thus $P_{\lceil \log(U+1) \rceil}$ denotes the original problem instance while P_0 denotes the network with all arc capacities being zero.

The capacity scaling algorithm, shown in Figure III.12, starts at each iteration with the maximum flow on the network P_i and obtains the maximum flow on the network P_{i+1} using the augmenting paths method. The algorithm starts with a zero flow on the network P_0.

The correctness of the capacity scaling algorithm is immediate. The following theorem analyzes the time complexity of this algorithm.

THEOREM 8. *On a network with integral arc capacities, the capacity scaling algorithm runs in $O(m^2 \log U)$.*

```
for i ← 0 to ⌈log(U + 1)⌉ − 1 do

    Update the arc capacities in the network P_i by multiplying
    the current arc capacities by 2 and then adding 1 to the
    capacity of an arc a if u_{i+1}(a) = 2u_i(a) + 1;

    Update the current flow by multiplying it by 2 ;

    Apply the augmenting path algorithm by starting with the
    current flow on the new network P_{i+1}.
```

FIGURE III.12. The Capacity Scaling Algorithm.

Proof. Let f_i denote the maximum flow in P_i. We claim that $|f_{i+1}| \leq 2|f_i| + m$ where $0 \leq i < \lceil \log(U+1) \rceil$. To see this, consider a minimum cut (S,T) saturated by f_i in P_i. We know by the Max-flow min-cut theorem that $f_i(S,T) = u_i(S,T)$ and by Lemma 5, $f_{i+1}(S,T) \leq u_{i+1}(S,T)$. Now $u_{i+1}(a) \leq 2u_i(a) + 1$ and the total number of arcs across the cut (S,T) can be no more than m. Thus using Lemma 4, we get the following :

$$|f_{i+1}| = f_{i+1}(S,T) \leq u_{i+1}(S,T) \leq 2u_i(S,T) + m = 2f_i(S,T) + m = 2|f_i| + m.$$

Since the augmenting path algorithm starts with a flow of value $2|f_i|$ on P_{i+1} and all arc capacities are integral, no more than m augmentations are needed in any iteration of the capacity scaling algorithm. The theorem follows immediately. ■

Algorithms like the capacity scaling algorithm discussed above are often referred to as *weakly polynomial* algorithms due to the dependence of their run-time on the size of the input numbers. In the next few lectures, we will see polynomial time algorithms for the maximum flow problem whose run-time is solely a function of the variables m and n. This later class of algorithms is referred to as *strongly polynomial* algorithms.

LECTURE IV

The Push-Relabel Algorithm

Scribe: Shane Henderson

Today's lecture describes the *Push-Relabel* method [18] for finding a maximal flow in a network, and then determines bounds for basic operations used by the algorithm. Most of the material presented here can be also found in [4] which has an excellent introductory discussion of the algorithm.

1. Preliminaries

Consider a directed network $G(V, E)$ with a source s and a sink t where the capacities of the arcs are given by the non-negative function u. We seek a maximal flow from the source to the sink.

The *excess function* $e_f(v) = \sum_{(v,w)\in E} =$ (incoming flow) $-$ (outgoing flow) at node v. Notice that we are not requiring flow conservation (otherwise e_f is a particularly boring function).

A node v is *active* if $(v \neq s, t)$ and $(e_f(v) > 0)$. A node is active if it is receiving more flow than it is passing on.

A *distance labeling* is a function d from the nodes to the nonnegative integers, such that $d(t) = 0, d(s) = n$, and $d(v) \leq d(w) + 1$ for all residual arcs (v, w). We may refer to a *valid* labeling. This is exactly the same thing as a distance labeling. We call it a valid labeling to emphasize the fact that it satisfies the constraints a distance labeling must satisfy.

An arc (v, w) is *admissible* if $((v, w) \in E_f)$ and $(d(v) = d(w) + 1)$.

Recall that a *pseudoflow* f is a function on the arcs of a network that satisfies the capacity and antisymmetry constraints. A pseudoflow does not necessarily have to satisfy conservation of flow. A *preflow* is a pseudoflow f where the excess function is nonnegative for all nodes other than s and t, *i.e.*, a preflow is what you have when the inflow $\geq$ the outflow for all nodes except the source and sink.

2. The Algorithm

2.1. A Rough Description of the Algorithm. The push-relabel algorithm works by loading up the source node s with more than enough flow, pushing that flow to the sink, and then pushing the excess flow back to the source. The algorithm uses two operations *Push* and *Relabel* to do this, hence the name.

It may be helpful to view the distance function as the height of a node, the arcs as pipes, and flows as water in the pipes. In this context we always push water downhill. The push operation corresponds to allowing excess water at a node to flow down to another node, and the relabeling operation corresponds to lifting a node up to allow water to flow. An arc is admissible if the arc has some spare capacity and the tail of the arc is a bit higher than the head, so we can "pour" more water down the arc. For more details see [4].

At all times a preflow f and a distance labeling d are maintained and push and relabel operations are applied until there are no active nodes.

LEMMA 6. *If f is a preflow and d is a distance labeling, t is not reachable from s in the residual graph G_f.*

Proof. Suppose there is an augmenting path $s = v_0, v_1, ..., v_l = t$. We may assume the path is simple (otherwise remove any loops in the path) so that $l < n$. Since d is a distance labeling we know that for any arc (v_i, v_{i+1}) on the path, $d(v_i) \leq d(v_{i+1}) + 1$. Iterating we get

$$d(v_0) \leq d(v_1) + 1 \leq d(v_2) + 2 \leq \cdots \leq d(v_l) + l.$$

We know that $d(v_0) = d(s) = n$ and $d(v_l) = d(t) = 0$. This implies that $n \leq l$. But $l < n$. Therefore, no augmenting path exists. ■

2.2. Initializing the Algorithm. The push-relabel algorithm requires a preflow and a distance labeling before it can get started. To generate a preflow fill all the arcs leaving s to capacity and set the flows in all the other arcs to zero. See Figure IV.13. An initial distance labeling is given by $d(s) = n$ and $d(v) = 0$ for all other nodes v. Notice that any arc from s to a node v is at its capacity, so it is not a residual arc, and therefore we do not require $d(s) \leq d(v) + 1$. Also, for any residual arc (v, w), $d(v) = 0 \leq 0 + 1 = d(w) + 1$ so that d is indeed a distance labeling.

Once we have a preflow and a distance labeling we begin applying push and relabel operations, so now might be a good time to describe those operations!

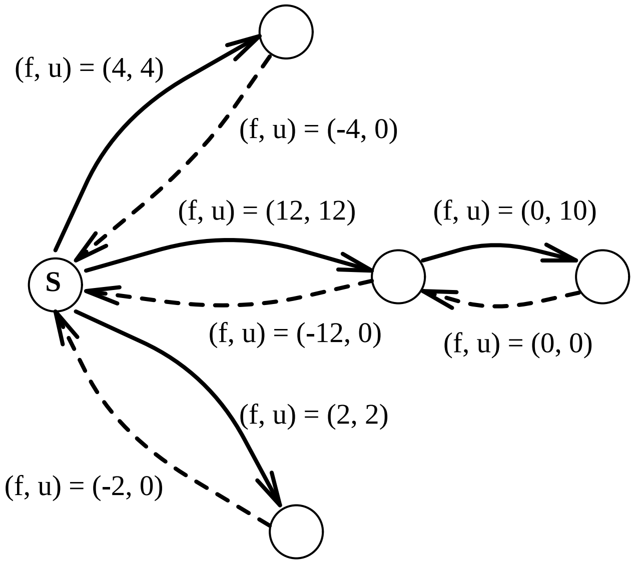


FIGURE IV.13. The initial preflow.

2.3. The Push Operation. The idea behind the *push* operation is to move excess flow downhill. In order to do this we need an active node (one with excess flow) and an admissible arc (one with some spare capacity to handle the increased flow). We need an arc $(v, w) \in G_f$ where $e_f(v) > 0$ and $d(v) = d(w) + 1$. It is tempting to say we could push the flow further downhill if $d(v) > d(w) + 1$. However, in this case the arc does not satisfy $d(v) \leq d(w) + 1$ and so cannot be a residual arc.

A *push* from v to w increases $f(v, w)$ by as much as possible. How much can we increase $f(v, w)$ by? We need to maintain the preflow which means keeping $f(v, w) \leq u(v, w)$ and $e_f(v) \geq 0$. Hence we may increase $f(v, w)$ by $\delta = \min\{u_f(v, w), e_f(v)\}$. This increases $e_f(w)$ by δ and decreases $f(w, v)$ and $e_f(v)$ by δ. If $u_f(v, w) = 0$ after the push (the arc gets filled to capacity) we have a *saturating* push, otherwise we have a *non-saturating* push. Figure IV.14 gives examples of both types of push operation.

When pushes are no longer possible, relabel operations are performed.

2.4. The Relabel Operation. We can think of a relabel operation as lifting a node v (increasing $d(v)$) so that flow can move "down" to a neighboring node. It is only worthwhile lifting v if v has some excess flow and all arcs leaving v with excess capacity are level or uphill. Hence the relabel operation is performed on an active node v when all arcs leaving v are **not** admissible. The relabel operation replaces $d(v)$ by $\min_{(v,w)\in E_f}(d(w) + 1)$. Figure IV.15 provides an example.

The following lemmas describe important properties of push and relabel operations.

LEMMA 7. *The push and relabel operations maintain preflow and distance labeling valid-*

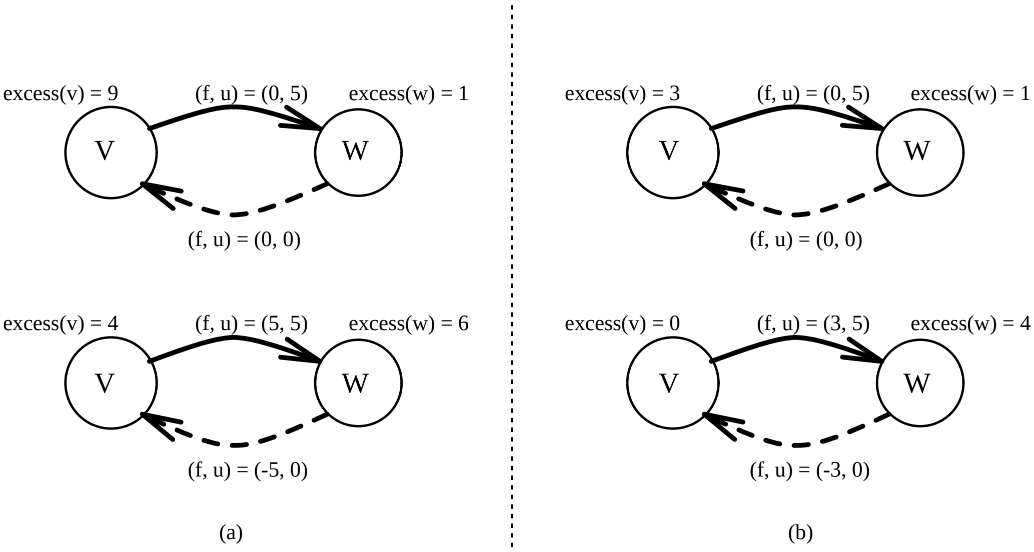


FIGURE IV.14. (a): a saturating push; (b) a non-saturating push.

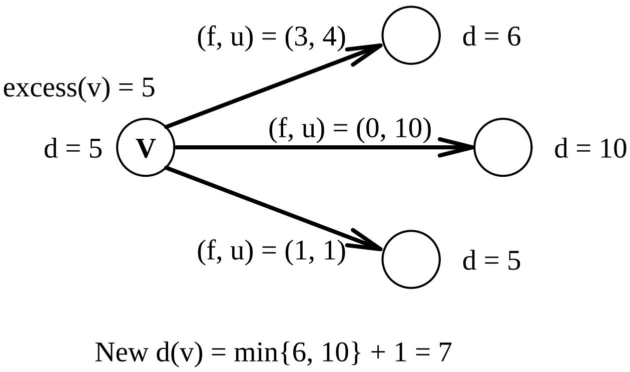


FIGURE IV.15. A *relabel* operation.

ity.

Proof. The relabel operation does not change the flow and is constrained so that it does not violate the distance labeling. Hence it is harmless. The push operation is constrained so that the properties of the preflow are not violated. It can however create new residual arcs by increasing the flow along an arc (v, w) that previously contained no flow. This creates spare capacity in (w, v), making (w, v) a residual arc; see Figure IV.16. We therefore require that after the push $d(w) \leq d(v) + 1$, but a push operation is only applied when $d(v) = d(w) + 1$ so that $d(w) = d(v) - 1 \leq d(v) + 1$. Hence the distance labeling is still valid after a push operation. ■

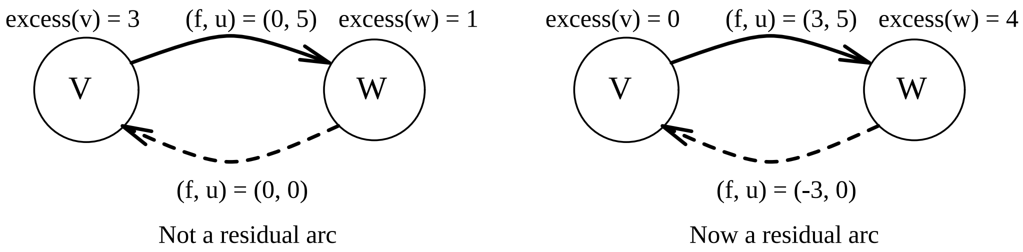


FIGURE IV.16. The *push* operation can create a new residual arc.

LEMMA 8. *If a node v is active, then either push(v,w) applies for some w or relabel(v) applies.*

Proof. Suppose v is active. If there is an admissible arc (v, w), then push can be applied to (v, w). If there are no admissible arcs from v, then relabel can be applied to v. ■

2.5. The Algorithm. This is very simply stated.

while a *push* or *relabel* is possible
    perform the operation

If the algorithm terminates then we must have a flow because no nodes will be active. The flow is maximum because Lemma 6 shows that no augmenting path exists. In the next section we bound the number of operations, showing that the algorithm terminates.

3. Analysis of the Algorithm

In this section we provide bounds for the number of push and relabel operations.

LEMMA 9. *If v is an active node, there is a (simple) path from v to s in the residual graph G_f.*

Proof. Let S be the set of all nodes reachable from v in G_f and S^c be all nodes outside S. Suppose for contradiction that $s \in S^c$.

Since v is active $e_f(v) > 0$ and because f is a preflow, $e_f(w) \geq 0 \; \forall w \neq s$. Hence $e_f(S) = \sum_{w \in S} e_f(w) > 0$. Keep this in mind. Let (w, x) be an arc where $w \in S^c$ and $x \in S$. If $f(w, x) > 0$ then $f(x, w) < 0 = u(x, w)$ which means that $(x, w) \in E_f$ which in turn implies $w \in S$. This is a contradiction and so $f(w, x) \leq 0 \; \forall w \in S^c, x \in S$. Let $f(W, X)$ be the total flow from a subset of nodes W to a subset of nodes X so that $f(S^c, S) \leq 0$. So now

$$
\begin{aligned}
e_f(S) &= f(V, S) \\
&= f(S, S) + f(S^c, S) \\
&= f(S^c, S) \\
&\leq 0
\end{aligned}
$$

.

But $e_f(S) > 0$ so we obtain the required contradiction and the lemma is proved. ■

Lemma 10. $\forall v, d(v) \leq 2n - 1$.

Proof. It suffices to examine active nodes because $d(v)$ can only increase when v is active. If v is active, s is reachable from v in G_f by the previous lemma, and hence there is a simple path from v to s of length $l \leq n - 1$. Using the same reasoning as in Lemma 6 we know that $d(v) \leq d(s) + l \leq d(s) + n - 1 = 2n - 1$. ■

Lemma 11. *$\forall v$, $d(v)$ never decreases and relabeling v increases $d(v)$. Immediately after relabeling v, v does not have any incoming admissible arcs.*

Proof. The fact that $d(v)$ never decreases follows if we can show that relabeling v increases $d(v)$, since relabeling v is the only way of changing $d(v)$. Before the relabeling, we know that $d(v) \leq d(w)$ for all w reachable from v in the residual graph (otherwise a push is possible from v). After relabeling v, we know that there is some w_0 where $d(v) = d(w_0) + 1$ where w_0 is reachable from v in the residual graph. Therefore, $d(v)$ increased by at least one. Before the increase, we had $d(u) \leq d(v) + 1$ for all residual arcs (u, v). After the increase, $d(u) \leq d(v)$, so the second claim of the lemma follows. ■

Lemma 12. *The total number of relabelings is $O(n^2)$ and the cost of performing the relabelings is $O(nm)$.*

Proof. The cost of relabeling a node v is the out degree of v because we examine each node on the end of an arc from v to find the minimum distance label. Therefore, the cost to relabel every node exactly once is $O(m)$ since $\sum_{v \in V} outdegree(v) = m$. Each node can be relabeled at most $2n - 1$ times by the previous two Lemmas. Hence the total number of relabelings is at most $n(2n-1) = O(n^2)$ and the total cost of relabeling is at most $m(2n-1) = O(mn)$. ∎

LEMMA 13. *The number of saturating pushes is O(nm).*

Proof. Consider a saturating push along an arc (v, w). After the push, node v cannot push to w until w pushes back to v along the same arc. This can only occur if during the first push $d(v) = d(w) + 1$ and in the second $d(w) = d(v) + 1$. So $d(w)$ must increase by at least 2. Hence by lemma 10, a push and a push back can occur at most $n - 1$ times, so that each arc can experience a saturating push at most $n - 1$ times. Since there are only m arcs, the number of saturating pushes is bounded above by $m(n-1) = O(nm)$. ∎

LEMMA 14. *The number of non-saturating pushes is $O(n^2m)$.*

Proof. Define $\Phi = \sum_{v \in X} d(v)$ where X is the set of active nodes, so that $\Phi \geq 0$. Initially, $\Phi = 0$. Each non-saturating push decreases Φ by at least one because a non-saturating push from v to w deactivates v, subtracting $d(v)$ from Φ, while adding at most $d(w)$ to Φ, and $d(w) = d(v) - 1$. Therefore, the number of non-saturating pushes is bounded by the amount Φ can increase during the execution of the algorithm. Φ can only increase because of a saturating push or a relabeling. A saturating push from v to w can increase Φ by at most $2n - 1$, since no distance labels change, and only node w, whose height is at most $2n - 1$, can become active. There are at most $O(mn)$ saturating pushes, so the increase in Φ due to saturating pushes is $O(n^2m)$. Each relabeling increases Φ by the increase in the label of the relabeled node. There are n nodes, and each node can increase its label by at most $2n - 1$. Therefore, the increase in Φ due to relabelings is bounded by $2n^2 - n$ or $O(n^2)$. Clearly, $O(n^2m)$ is the dominant term. ∎

LECTURE V

Implementations of Push-Relabel Algorithms

Scribe: Meredith Goldsmith

We finished Lecture 4 by proving upper bounds on the number of update operations (saturating pushes, nonsaturating pushes, relabels) in any push-relabel algorithm for the maximum flow problem. Today, we discuss specific *implementations* of the push-relabel algorithm. In general, the bottleneck on running time is the number of nonsaturating pushes. A generic algorithm will be shown to have running time $O(n^2 m)$. By carefully choosing the order in which update operations are performed, we reduce the number of nonsaturating pushes and the time to $O(n^3)$. Using a special data structure, the time per nonsaturating push can be reduced for an overall time bound of $O(nm \log n)$. For complete details on these time bounds, see [18] or [16].

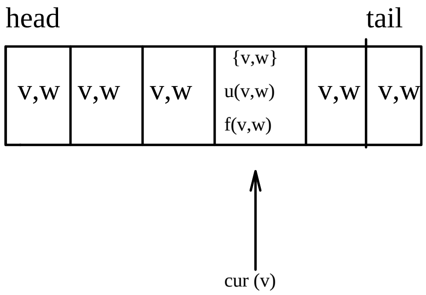


FIGURE V.17. Adjacency edge list data structure.

1. Simple Implementation

We will implement push-relabel by performing a series of operations called *discharge* which combine push and relabel operations in an efficient fashion. *Discharge*(v) takes an active node v and tries to push all excess flow through arcs leading to the sink; if there are no admissible arcs leaving v, *discharge* relabels v. To implement *discharge* we need some data structures to represent the network and the preflow.

1.1. Data Structures. Active nodes can be maintained in a stack or a queue. If we are not interested in a specific ordering of discharges, the particular data structure used is not important as long as we can find an active node in constant time, can add a new active node in constant time, and delete a node that becomes inactive in constant time.

For every node v we create a list $l(v)$ of all undirected edges adjacent to v. An *undirected edge* is defined as an unordered pair $\{v, w\}$ such that $(v, w) \in E$. The list can be in any (fixed) order. Notice that

(1) each edge $\{v, w\}$ appears in two lists, the one for v and the one for w, and
(2) $l(v)$ contains an edge $\{v, w\}$ for each possible residual arc (v, w).

We associate with each edge $\{v, w\}$ the values $u(v, w)$ and $f(v, w)$. For each list $l(v)$, we maintain a pointer $cur(v)$ which indicates the next candidate for a pushing operation from v. Initially, $cur(v) = head(l(v))$.

1.2. Discharge Subroutine. The *discharge* operation applies to an active node v. *Discharge* iteratively attempts to push $e_f(v)$ through (v, w) such that $\{v, w\} = cur(v)$ if a pushing operation is applicable to this arc. If not, the operation replaces $\{v, w\}$ by the next edge on the adjacency list $l(v)$ of v; or, if $\{v, w\}$ is the last edge on this list, it resets $cur(v)$ to $head(l(v))$ and relabels v. *Discharge*(v) stops when $e_f(v)$ is reduced to zero or v is relabeled.

The main loop of the implementation consists of repeating the *discharge* operation described in Figure V.18 until there are no remaining active nodes. When there are no remaining active nodes, neither push nor relabel can be applied, so f is a maximum flow.

```
discharge(v).
Applicability: v is active.
    time-to-relabel ← false;
    repeat
        if cur(v) is admissible then push(cur(v))
        else
            if cur(v) is not the last arc on the arc list of v then
                replace cur(v) by the next arc on the list
            else begin
                cur(v) ← first(v);
                time-to-relabel(v) ← true;
            end;
    until e_f(v) = 0 or time-to-relabel;
    if time-to-relabel then relabel(v);
```

FIGURE V.18. The *Discharge* Operation.

LEMMA 15. *Discharge applies relabeling correctly.*

Proof. *Discharge* only applies if v is active and stops as soon as v becomes inactive, so if it calls *relabel*(v), v is active. It remains to show that when *discharge* relabels v, v has no outgoing admissible arcs.

At one time $cur(v)$ pointed to the head of v's adjacency list. Then $cur(v)$ marched down the list, considering each arc (v, w) and then advancing to the next arc. When we advanced, either we had $d(w) \geq d(v)$ or (v, w) was saturated.

Suppose $d(w) \geq d(v)$ at the time $\{v, w\}$ was the current arc. *Discharge*(v) calls *relabel*(v) only when it has reached the end of v's adjacency list, and no other call to *relabel* can occur with v as the node being processed. Therefore, during the march down v's adjacency list, $d(v)$ remains unchanged. Since the distance label of a node is monotonically increasing, so $d(w)$ could only have increased. So $d(w) \geq d(v)$ still holds.

In the second case, if $(v, w) \notin E_f$ when it was the current arc, the residual flow along (v, w) could only have increased if at some later time w pushed flow to v. This would be true only if $d(w) > d(v)$; in fact, $d(v) = d(w) - 1$. Since $d(v)$ remained unchanged and $d(w)$ cannot decrease, (v, w) is still not admissible. ■

LEMMA 16. *A generic version of the push/relabel algorithm based on discharging runs in $O(nm)$ time plus the total time need to do the nonsaturating pushes and to maintain the set of active nodes, for a total of $O(n^2m)$ time.*

Proof. Finding an active node takes constant time, and this cost can be absorbed by a subsequent push or relabel operation. Every step of *discharge*(v) either causes a push, increments the pointer $cur(v)$, or relabels v. We proceed to count the total time needed for these operations.

Assign the cost for running down the arc lists in the discharge procedure to the relabeling procedure. This is legitimate since we run down the arc lists at most one more time than the number of relabeling calls and each relabeling call will run down the arc lists to find the minimum $d(w)$ for each neighbor w. Recall that the the total cost of relabeling was shown to be $O(nm)$. A push takes constant time, and the number of saturating pushes is $O(nm)$, for a total running time of $O(nm)$ plus the time for nonsaturating pushes. With $O(n^2m)$ nonsaturating pushes, the push-relabel max-flow algorithm with the above data structures and the discharge procedure runs in at most $O(n^2m)$ time. ■

2. Maximum Distance Discharge Algorithm

By discharging active nodes in a restricted order, we can reduce the number of nonsaturating pushes from $O(n^2m)$ to $O(n^3)$. One common ordering which can be shown to have running time of $O(n^3)$ is the FIFO ordering. The FIFO algorithm maintains the set of active nodes as a queue, selecting new nodes to discharge from the head and adding newly active nodes to the tail.

Another implementation, called the maximum distance ordering, always discharges the node with the largest distance label. Using a more sophisticated analysis, the maximum distance label algorithm has been shown by Cheriyan and Maheshwari [3] to run in $O(n^2\sqrt{m})$ time, but we will show a bound of $O(n^3)$.

2.1. Implementation of Maximum Distance Discharge Algorithm. The way to implement max distance ordering is straightforward. Put the active nodes in buckets according to their distance label. Recall that $\forall v\ 0 \leq d(v) \leq 2n-1$. Create an array of buckets B_i, $i \in [0, \ldots, n-1]$, such that bucket $B_i = \{\text{active } v : d(v) = i\}$. A pointer p into the array points to the last bucket looked into, whose label is always the largest of any active node. Let B_p be the bucket pointed to by p.

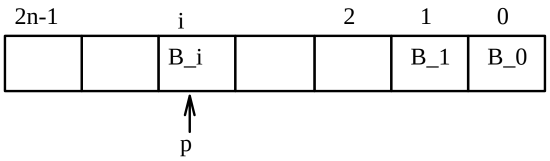


FIGURE V.19. Array of Buckets

At initialization place all active nodes in B_0 and set $p \leftarrow 0$. At each iteration, the max distance algorithm removes a node from B_p for discharging. If bucket B_p is empty, move p to the right; if a node v is relabeled during the discharge, set $p \leftarrow d(v)$ and put v into B_p. If additional nodes w were made active by *discharge*, add these to $B_{d(w)}$. Stop when p moves too far to the right because there are no remaining active nodes. See Figure V.20 for the pseudocode.

```
process-max-dist-node
    repeat
        remove a node v from bucket B_p ;
        last-label ← d(v) ;
        discharge(v) ;
        for each newly active node w, add w to B_d(w) ;
        if d(v) ≠ last-label then do
            p ← d(v) ;
            put v into bucket B_p;
        end
        else if bucket B_p is empty then move p ← p − 1 ;
    until p < 0 ;
```

FIGURE V.20. Maximum Distance Discharge Algorithm.

LEMMA 17. *The time spent updating the pointer p is $O(n^2)$.*

Proof. The pointer p can move to the left only when a node is relabeled and by the amount equal to the increase in the distance label of the node. Thus the number of steps to the left is $O(n^2)$. The number of steps to the right cannot exceed the number of steps to the left by more than the size of the array, which is $2n-1$. ■

The bottleneck will be the number of nonsaturating pushes. We will show a bound of $O(n^3)$ nonsaturating pushes by dividing the computation into phases.

DEFINITION 1. *A phase is the period between two subsequent relabelings (of any node).*

THEOREM 9. *The maximum label algorithm runs in $O(n^3)$ time.*

Proof. During a phase, the pointer p moves only to the right (that is, the maximum label is nonincreasing). The number of phases is clearly the same as the number of relabelings, $O(n^2)$. There is at most one nonsaturating push per node per phase, since if we have a nonsaturating push we do not relabel. Thus, the number of nonsaturating pushes in the maximum label algorithm is $O(n^3)$, which is also the running time by Lemma 16. ■

3. Implementation with Dynamic Trees

There are two ways to reduce cost of nonsaturating pushes:

(1) Reduce the number of nonsaturating pushes with smart ordering of active nodes.
(2) Reduce the cost per nonsaturating push by doing a bunch of such pushes at once.

The idea behind the second one is to save information about arcs used in nonsaturating pushes and use this information.

3.1. Dynamic Trees Data Structure. The general idea is to perform a series of pushes at once along a path in one direction, using the dynamic tree data structure which Sleator and Tarjan [41, 5] designed for this purpose. Here, we treat this data structure as a black box.

This data structure works with a forest of node-disjoint dynamic trees. Tree arcs always point from a node to its parent. Dynamic trees support the operations described in Figure V.21. For any sequence of k operations, if n is the maximum tree size, the amortized cost per dynamic tree operation is $O(\log n)$.

3.2. Send Operation. We represent the residual graph G_f by a *forest* containing every node in some tree. We initialize the forest by calling *make-tree* on every node singleton. Throughout the algorithm, we maintain the invariant that

- Every tree arc is admissible (except possibly in the middle of processing a tree).
- Every active node is a tree root.

make-tree(v): Make new node v into a one-node dynamic tree.
find-root(v): Find and return the root of the tree containing node v.
find-value(v): Find and return the capacity of the arc from v to its parent, or else ∞ if v is the root.
find-min(v): Find the node of minimum capacity along the path from v to the root.
change-value(v, δ): Change the value of the arc on path from v to root by δ.
link(v, w, x): Combine the trees containing nodes v and w by making w the parent of v and x the residual capacity along the connecting tree arc. This operation does nothing if v and w are in the same tree or if v is not a tree root.
cut(v): Reverse of link. Break the tree containing v into two trees by deleting the arc from v to its parent. This operation does nothing if v is a tree root.

FIGURE V.21. Dynamic tree operations.

ALGORITHM 1. Proceed as in discharge-based maximum labeling algorithm, but replace calls to *push*(v, w) by calls to *send*(v, w). Additionally, whenever a node v is relabeled, for all tree arcs (u, v), *cut*(u) is performed because these arcs are no longer admissible.

Pseudocode for *send* appears in figure V.22. The procedure *send*(v, w) takes an active node v (which must be a tree root) and an admissible arc (v, w). Nodes $\{v, w\}$ must be in different trees if the invariant holds. Next v is linked to w by an arc with value $u_f(v, w)$. Then *send* pushes as much excess as possible from v to the root of its new tree, and then cuts all saturated arcs to restore the invariant that all tree arcs be admissible. Note that the preflow is represented in two different ways: explicitly for the arcs not in the dynamic trees and implicitly for the tree arcs. When the algorithm terminates, the explicit flow on the tree arcs can be obtained easily.

```
Send(v, w).
Applicability:  v is active and (v, w) is admissible.
    link (v, w, u_f (v, w));
    parent (v) ← w;
    δ ← min (e_f (v), find-value (find-min (v));
    change-value (v, −δ);
    While v ≠ find-root (v) and find-value(find-min (v)) = 0 do begin
        z ← find-min (v);
        cut (z);
        f (z, parent (z)) ← u (z, parent (z));
        f (parent (z), z) ← u (z, parent (z));
    end.
```

FIGURE V.22. The *Send* operation.

LEMMA 18. *The number of cuts is $O(nm)$; the number of links $O(nm)$; the number of send operations is $O(nm)$.*

Proof. Cuts are performed when an arc gets saturated or when relabel is called. A *relabel*(v) operation causes at most degree(v) cuts. The number of saturations has been shown to be $O(nm)$. Each node can be relabeled at most $O(n)$ times, and the cost of cutting arcs associated with relabeling every node exactly once is $O(m)$, so the total cost of cuts associated with relabeling is $O(nm)$. The number of links can be no more than n plus the number of cuts, so this is also $O(nm)$. *Send* always calls *link*, so the number of sends must be $O(nm)$. Other work done by the algorithm can be charged to links and cuts, so that each link and cut is charged a constant amount of work and a constant number of dynamic tree operations. ∎

THEOREM 10. *Any discharge-based implementation of push relabel which uses dynamic trees runs in $O(nm \log n)$ time.*

Proof. The maximum size of any dynamic tree is $O(n)$. Each send does $O(1)$ dynamic tree operations plus $O(1)$ per arc saturated. The number of saturated arcs is the same as the number of cuts, which is $O(nm)$. Since there are $O(nm)$ sends, $O(nm)$ saturated arcs, and $O(\log n)$ time per dynamic tree operation, the theorem follows. ∎

Actually, with more complicated analysis a bound of $O(nm \log(n^2/m))$ has been shown for a slightly different algorithm.

LECTURE VI

Minimum Cycle Mean in a Digraph

Scribe: Claudionor N. Coelho

Today's lecture was given by Robert Kennedy. The topic of the lecture was Karp's Minimum Mean Cycle Algorithm. The reference for the lecture is in [25]. Please note that the article has a typo (an "x" should be an "s") and an error in the algorithm for finding the cycle itself.

1. Problem Motivation

The need for minimum mean cycles can be motivated by the following problems:

(1) The "Tramp Steamer" Problem — You are a steamship owner making a living by shipping goods among different cities. There are several defined routes between cities each having an associated cost (negative if profitable, positive if not). In order to maximum your profit, you want to find a cyclic route that minimizes your cost per voyage.

(2) Minimum-cost circulations by cancelling minimum mean cycles, which will be covered by Prof. Goldberg in a few weeks. Minimum cycle mean is used as a subroutine.

2. Problem Definition

Given a directed and strongly connected graph $G = (V, E)$ and weight function $f : E \to \mathbf{R}$, let the weight of a path P be

$$w(P) = \sum_{e \in P} f(e)$$

and the mean weight of P be

$$m(P) = \sum_{e \in P} \frac{f(e)}{|P|},$$

where $|P|$ is the number of arcs of the path P.

The problem is to find a cycle C in G such that for any cycle C' in G, $m(C') \geq m(C)$. Note that if the graph is not strongly connected, we can determine the minimum mean cycle for each strongly connected component, and then take the least of these.

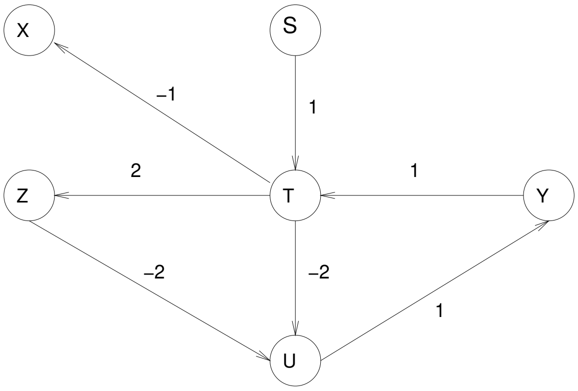


FIGURE VI.23. Example graph.

EXAMPLE 4. *For the rest of the notes we will use the graph of Figure* VI.23 *as an example. There are two cycles in G, $zuyt$ and tuy.*

$$m(zuyt) = (2 = 1 = 1 - 2)/4 = 1/2$$

$$m(tuy) = (1 + 1 - 2)/3 = 0/3 = 0.$$

Clearly, cycle tuy has the minimum mean weight.

Let $F_k(v)$ be defined as the total weight of a minimum path from an arbitrarily fixed "reference vertex" s to vertex v, having exactly k arcs. If no such path exists with exactly k arcs, then $F_k(v)$ is defined to be ∞. The following theorem relates a minimum mean cycle with those minimum paths from a reference node.

THEOREM 11. *λ^*, the minimum mean cycle weight, obeys*

$$\lambda^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\}$$

where $n = |V|$.

EXAMPLE 5. *In the example of figure* VI.23, *the path stz has total weight of 3 (i.e., $F_2(z) = 3$).*

There are two steps to the proof, the first to show that if $\lambda^* = 0$ then

$$\lambda^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\} = 0$$

and then to show that if $\lambda^* \neq 0$ the problem can be reduced to the case $\lambda^* = 0$ without loss of generality.

LEMMA 19. *If $\lambda^* = 0$, then*

$$\min_{v \in V} \max_{0 \le k \le n-1} \left\{ \frac{F_n(V) - F_k(V)}{n-k} \right\} = 0.$$

Proof. The proof technique will be to view cycles as tails of paths from s.

Since $\lambda^* = 0$, there is a cycle of weight zero, and no cycle of negative weight, therefore there must exist a minimum-weight path P_v from s to v whose length may be taken to be less than n. Since there are n nodes in the graph, if the length of the path P_v is greater than n, some node u appears twice, meaning that there is a cycle. Since $\lambda^* = 0$, the cycle must be non-negative. So, if we disregard this cycle in the path P_v, yielding P'_v, we will have that $w(P'_v) \le w(P_v)$, and P_v would not be a minimum-weight path from s to v, a contradiction.

Let $\pi(v)$ be the total weight of P_v, then

$$\pi(v) = \min_{0 \le k \le n-1} \{F_k(v)\}$$

and since $F_n(v) \ge \pi(v)$ we get,

$$F_n(v) - \pi(v) \ge 0$$

$$F_n(v) - \min_{0 \le k \le n-1} F_k(v) \ge 0$$

$$\max_{0 \le k \le n-1} \{F_n(v) - F_k(v)\} \ge 0$$

$$\max_{0 \le k \le n-1} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\} \ge 0.$$

Showing that there exists a v such that $F_n(v) = \pi(v)$ will prove the lemma. Let C be a cycle of zero weight and let v be a vertex on C. Note that P_v concatenated with any number of trips around C is no heavier than a path from s to v than P_v (*i.e.*, since C has zero weight two paths $P_v + x$ trips around C and $P_v + y$ trips around C is have the same weight). Let u be a node in the cycle C (u may be the same node as v or some other node of the cycle C). Construct a path P_u by following P_v, from s to v, and circling C until we reach the vertex u which makes P_u have length equal to n. If we can show that P_u is a lightest weight path from s to u then we can conclude that $F_n(u) = \pi(u)$, implying $F_n(u) - \pi(u) = 0$.

By contradiction: Assume that P'_u is a lighter weight path from s to u than P_u, and that $u \ne v$ (if $v = u$, P'_u is a lighter path than P_v, a contradiction). Since the cycle C has weight 0, we know that $w(P_{(v,u)}) = -w(P_{(u,v)})$. If $w(P_{(v,u)}) > w(P_{(u,v)})$, then $w(P_v) > w(P'_u) + w(P_{(u,v)})$, and P_v is not a minimum-weight path. Similarly, for the case where $w(P_{(v,u)}) < w(P_{(u,v)})$, $w(P'_u) > w(P_v) + w(P_{(v,u)})$, and P'_u is not lighter than P_u. ∎

Proof of theorem 11. To prove that when $\lambda^* \neq 0$ the theorem holds we use the following argument. By reducing each arc weight by a constant c, $F_k(e)$ is reduced by kc and hence

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\}$$

is reduced by c. If we use a weighting function $f' = f + c$ such that f' makes $\lambda^* = 0$ and apply the above lemma we have proved the theorem. ■

3. Algorithm

To find the minimum mean cycle in a graph G make a table for $F_k(v)$ using the following equations, for all v in V,

$$F_0(v) = 0, \ \text{ if } v = s,$$

$$F_0(v) = \infty \ \text{ if } v \neq s,$$

and for $0 < k \leq n$,

$$F_k(v) = \min_{w \in V} \{F_{k-1}(w) + f(w, v)\}.$$

Example 6. *For the example of figure* VI.23*, the following table gives the values for F_k.*

k	$F_k(s)$	$F_k(t)$	$F_k(u)$	$F_k(y)$	$F_k(x)$	$F_k(z)$
0	0	∞	∞	∞	∞	∞
1	∞	1	∞	∞	∞	∞
2	∞	∞	-1	∞	0	3
3	∞	∞	1	0	∞	∞
4	∞	1	∞	2	∞	∞
5	∞	3	-1	∞	0	3
6	∞	∞	1	0	∞	5

The computation requires $O(n|E|)$ operations, and once the quantities $F_k(v)$ have been tabulated, we can compute λ^* in $O(n^2)$ further operations. To find the minimum cycle, find vertex v that minimizes

$$\max_{0 \leq k \leq n} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\}$$

and the k that maximizes

$$\frac{F_n(v) - F_k(v)}{n-k}.$$

The length n path from s to v that attains $F_n(v)$ contains a cycle, that cycle has minimum mean weight.

Example 7. *To complete the example of figure* VI.23*, consider the following table.*

k	s	t	u	y	x	z
0	∞	∞	$-\infty$	$-\infty$	∞	$-\infty$
1	∞	∞	$-\infty$	$-\infty$	∞	$-\infty$
2	∞	∞	0.5	$-\infty$	∞	0.5
3	∞	∞	0	0	∞	$-\infty$
4	∞	∞	$-\infty$	-1	∞	$-\infty$
5	∞	∞	2	$-\infty$	∞	2
max			2	0		2

This table shows that y is the vertex that minimizes

$$\max_{0 \leq k \leq n} \left\{ \frac{F_n(v) - F_k(v)}{n-k} \right\}$$

and $k = 3$ maximizes

$$\frac{F_n(v) - F_k(v)}{n-k}.$$

Finally, the length n path is *stutuy* and the cycle *tuy* has $\lambda^* = 0$.

4. Correctness

We show that the cycle found on the length n path from s to w has minimum weight. By the same transformation as before, we may assume without loss of generality that $\lambda^* = 0$.

By contradiction: Assume that the cycle is not zero weight. Deleting the cycle yields a lighter path to w, say with length j. Then $F_n(w) > F_j(w)$ and thus $F_n(w) - F_j(w) > 0$, a contradiction because

$$\begin{aligned} 0 = \lambda^* &= \min_{v \in V} \max_{0 \leq k \leq n-1} \left\{ \tfrac{F_n(v) - F_k(v)}{n-k} \right\} \\ &= \max_{0 \leq k \leq n-1} \left\{ \tfrac{F_n(w) - F_k(w)}{n-k} \right\} \\ &\geq \left\{ \tfrac{F_n(w) - F_j(w)}{n-k} \right\} > 0. \end{aligned}$$

LECTURE VII

The Blocking Flow Algorithm and Bipartite Matching

Scribe: Tzu-Hui Yang

1. Dinitz' Blocking Flow Algorithm

In today's lecture, Robert Kennedy covered Dinitz' blocking flow algorithm [6] and the bipartite matching problem.

DEFINITION 2. *A flow is called a* ***blocking flow*** *if every s-t path traverses a saturated arc.*

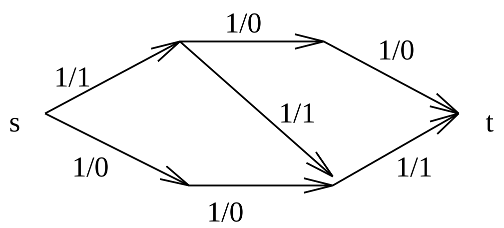


FIGURE VII.24. A blocking flow is not necessarily a maximum flow.

EXAMPLE 8. *Note that a blocking flow is not necessarily a maximum flow. In Figure* VII.24, *the flow is a blocking flow but not a maximum flow.*

DEFINITION 3. *A* ***layered network*** $L = (V, E)$ *is one in which every node* $v \in V$ *may be assigned to a layer number* $l(v)$ *and all arcs go from one layer to next,* i.e., *for all* $(v, w) \in E$, $l(w) = l(v) + 1$.

DEFINITION 4. *Given a network* $G = (V, E)$ *and a flow f, define* $G_l(f) = (V, A)$ *where* $A = \{(u, v) \in E_f | l(v) = l(u) + 1\}$ *and* $l(v)$ *is the breath first search distance of v from the source s in the residual graph* G_f.

Dinitz' blocking flow algorithm for finding a maximum flow is shown in Figure VII.25.

The following lemma implies that the algorithm terminates in at most $n - 1$ iterations.

LEMMA 20. *Let* $l_i(v)$ *denote the layer number of v on the i-*th *iteration. Then* $l_i(t) < l_{i+1}(t)$ *for all i,* i.e., $l(t)$ *increases at every iteration.*

$f \leftarrow 0$;
while there exists an augmenting path w.r.t. f
    Compute a blocking flow f' in $G_l(f)$;
    $f \leftarrow f + f'$.

FIGURE VII.25. Blocking flow algorithm

Proof. Let G_i be the residual graph before i-th iteration and let L_i be the layered network at i-th iteration. Clearly for $(v,w) \in G_i$, we have $l_i(w) \leq l_i(v) + 1$. Also, for $(v,w) \in L_i$, $l_i(w) = l_i(v) + 1$. Every arc of G_{i+1} is either an arc of G_i or the reverse of an arc of L_i. Hence for every $(v,w) \in G_{i+1}$, $l_i(w) \leq l_i(v) + 1$. By induction on $l_{i+1}(v)$, this implies that $l_i(v) \leq l_{i+1}(v)$. In particular, $l_i(t) \leq l_{i+1}(t)$.

Suppose $l_i(t) = l_{i+1}(t)$ and let P be a shortest path from s to t in G_{i+1}. Then for any node v on P, $l_i(v) = l_{i+1}(v)$. This implies that P is a path in L_i, contradicting the fact that at least one arc on P must be saturated by the blocking flow found in L_i. So $l_i(t) < l_{i+1}(t)$. ∎

We've proved that Dinitz' blocking flow algorithm takes less than n iteration. The question now is how long does it take to find a blocking flow in the layered network. The following simple algorithm is easily seen to run in $O(m^2)$ time:

Saturate shortest paths from s to t in G_l, one at a time, using BFS.

But we can do better. Figure VII.26 describes an $O(nm)$ blocking flow algorithm due to Dinitz, who observed that, if during a depth-first search the algorithm has to backtrack from v, there cannot be any path through v, so we can delete v and all its incoming arcs. The algorithm finds a path from s to t using depth-first search, pushing enough flow along the path to saturate some arc, deleting all saturated arcs, and repeating until t is no longer reachable from s.

To see that Dinitz' blocking flow algorithm is correct, observe that an arc (u,v) is deleted from G_l only if (u,v) is saturated or v has no outgoing arcs (equivalently, all paths from v to t are saturated). The algorithm continues to search for augmenting path until there is none available. It is clear that the algorithm produces a blocking flow.

In this algorithm, the function *init* is called no more than $m+1$ times. *Augment* is an $O(n)$ operation since the length of any s-t path is less than n. Since every invocation of either *augment* or *retreat* deletes at least one arc, the total number of calls to either *augment* or *retreat* is at most m. Finally, at most $n-1$ *advance* steps precede an *augment* or *retreat*, so there are at most $(n-1)m$ *advance* steps. Therefore, Dinitz' blocking flow algorithm finds

```
init:     p ← [s]; v ← s; goto advance
advance:  if there is no (v, w) ∈ G_l goto retreat
          else begin
                p ← p + [w]; v ← w
                if w = t goto augment
                else goto advance
          end
augment:  let Δ be the minimum residual capacity on augmenting path p
          add Δ to f on all arcs of p
          delete saturated arcs
          goto init
retreat:  if v = s STOP /* flow is blocking */
          else begin
                let (w, v) be the last arc on p
                p ← p − [v]; delete v and all arcs into v; v ← w
                goto advance
          end
```

FIGURE VII.26. Blocking flow algorithm.

a blocking flow in $O(nm)$ time and finds a maximum flow in $O(n^2m)$ time.

2. Bipartite Matching

DEFINITION 5. *A **matching** M in a graph G is a set of edges that are node disjoint. And the **size** of a matching, denoted $|M|$, is the number of edges in the matching.*

DEFINITION 6. *A **bipartite graph** $G = (V_1 \bigcup V_2, E)$ is one with the property that for all $(v, w) \in E$, $v \in V_1$ and $w \in V_2$, where $V_1 \bigcap V_2 = \phi$.*

Given an undirected bipartite graph G, the *bipartite matching problem* is to find in G a matching of maximum size.

For example, suppose V_W is a set of *workers*, and V_J is a set of *jobs*. Define $G = (V, E)$, with $V = V_W \bigcup V_J$, and $(u, v) \in E$ *iff* worker u is qualified to do job v. The problem of assigning workers to jobs so that as many jobs as possible are assigned is the bipartite matching problem.

For a bipartite graph $G = (V, E)$, with $V = V_1 \bigcup V_2$, there is a natural correspondence between matchings and flows through a network G' derived from G by directing all edges, say from V_1 to V_2, adding two new nodes, s and t, and arcs (s, v_1), (v_2, t) for all $v_1 \in V_1$, $v_2 \in V_2$, and setting all arc capacities equal to one. A matching M in G defines a flow with value $|M|$ on G': saturate all arcs corresponding to M, and all (s, v_1), (v_2, t) arcs incident to these arcs. Conversely, any integral flow in G' defines a matching in G under the obvious correspondence. The flow value is equal to the matching size, so a maximum flow corresponds to a maximum matching.

THEOREM 12. [9, 48] *Dinitz' blocking flow algorithm solves the bipartite matching problem in $O(\sqrt{n}m)$ time.*

Proof. Since all positive residual capacities are equal to one, all augmenting path arcs are deleted after an augmentation. This implies that the total augmentation cost is $O(m)$. For each (v, w) in the layered network, there is at most one advance over (v, w), since either an augmenting path containing (v, w) is found or the search retreats from w to v, and in both cases (v, w) is deleted from the graph. Thus each blocking flow computation takes $O(m)$ time.

It remains to show that the algorithm terminates in $O(\sqrt{n})$ iterations. Let f^* be an optimal flow. We'll show that after $\sqrt{n}$ iterations, $|f^*| - |f| \leq \sqrt{n}$. Then, since each blocking flow step increases the flow by at least one, the total number of iterations is $O(\sqrt{n})$.

Actually, it's convenient to prove something stronger: if $l(t) = k$, then $|f^*| - |f| \leq n/k$. The following observation is a key to the proof. Suppose we have a 0-1 flow in a network obtained from a bipartite matching flow; then every internal node has either in-degree of one or out-degree of one (in the residual graph).

Consider a flow f' that augments the current flow f to an optimal flow f^*. By the decomposition theorem, f' can be decomposed into a collection of paths and cycles, and the number of paths is equal to $|f^*| - |f|$. Note that the paths are node-disjoint by the above observation, and each path has length of at least k. Therefore the number of paths is at most n/k. ∎

DEFINITION 7. *In a bipartite matching problem, a* ***perfect matching*** *is a matching such that all nodes in both sets are matched.*

Hall's theorem is used to give an easy proof that no perfect matching exists. First, the concept of a neighborhood is defined.

DEFINITION 8. *Given a graph $G = (V, E)$ and a subset X of nodes, define $N(X)$, the* ***neighborhood*** *of X, as $N(X) = \{v | (x, v) \in E, x \in X\}$.*

THEOREM 13. ***Hall's Theorem***
There exists a perfect matching in a bipartite graph $G = (V_1 \bigcup V_2, E)$ if and only if for every $X \subseteq V_1$, $|X| \leq |N(X)|$.

Proof. ($\Rightarrow$) Suppose a perfect matching exists. Let X be any subset of V_1. Each element of X must be matched to an element of V_2 and different elements of X are matched to different elements of V_2. Therefore, $|X| \leq |N(X)|$.

($\Leftarrow$) Assume that there is no perfect matching. Consider an integral maximum flow in the corresponding matching network $G = (V, E)$, $V = V_1 \bigcup V_2 \bigcup \{s, t\}$, and the minimum cut$(S, T)$, where S is the set of nodes reachable from s in the residual graph G_f and $T = V - S$. Let $X = \{v | v \in V_1$ and there is a s-v path in $G_f\}$. Note that all unmatched nodes of V_1 are in X. (Figure VII.27 illustrates the proof.) The arcs from s to $V_1 - X$ and the arcs from $N(X)$ to t are saturated by the definition of X. Therefore, the minimum cut has capacity of at least $(|V_1| - |X|) + |N(X)|$, *i.e.*,

$$(capacity\ of\ minimum\ cut) \geq (|V_1| - |X|) + |N(X)|.$$

But we know this must be less than $|V_1|$ because there is no perfect matching, *i.e.*,

$$|V_1| > (capacity\ of\ minimum\ cut).$$

Combine the two inequalities and we get

$$|X| > |N(X)|.$$

Therefore, $\forall X \in V_1$, $|X| \leq |N(X)|$ implies that there exists a perfect matching. ■

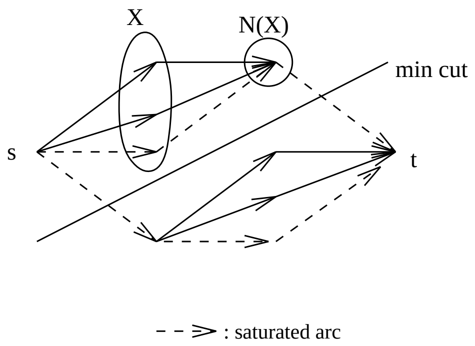


FIGURE VII.27. Proof of Hall's Theorem.

LECTURE VIII

Two Phase Push-Relabel Algorithm

Scribe: Chih-Shuin Huang

1. Two Phase Push-Relabel Algorithm

The two-phase push-relabel algorithm [17] maintains a preflow g and a valid labeling d for g. Like the single-phase push-relabel algorithms from previous lectures, the two-phase algorithm proceeds by examining nodes with positive flow excesses and pushing flow from them to nodes estimated to be closer to t in the residual graph. As a distance estimate, the algorithm uses the labeling d, which it periodically updates to more accurately reflect the current residual graph.

During the first phase, flow is pushed toward the sink. The first phase ends having determined a minimum cut and the value of a maximum flow. During the second phase, remaining flow excess will return to the source, converting the preflow into a maximum flow. The dominant part of the computation is the first phase. Now we describe the two phases in more detail.

1.1. Phase I. The first phase begins like the single-phase algorithm, except

(1) we define $d(s) = n$ and
(2) we define a node v to be *active* if $v \neq t$, $e_g(v) > 0$, and $d(v) < n$.

1.2. Phase II. The second phase returns the flow excesses to s by one of two methods. We can either

- decompose the preflow according to a modified version of the flow decomposition theorem, and return excesses to s along paths in the decomposition; or

- run the first phase of the algorithm "backwards", *i.e.*, fix the label of the source at zero, initialize the labels of all other nodes to zero, and work in the residual graph with initial solution equal to the preflow we found in the first phase.

The first of these options requires $O(nm)$ time to decompose the preflow into s-t paths, cycles, and paths from s to nodes with excess, and then $O(n^2)$ time to return the excesses to the source in the obvious way.

The second option may require the same asymptotic amount of time as the first phase, though in practice such an implementation of the second phase tends to be very fast, and is simpler to implement than flow decomposition. The remaining analysis will refer to this choice of second phase implementation.

2. Correctness of the Two Phase Push-Relabel Algorithm

For a preflow g, let (S_g, T_g) be the node partition such that T_g consists all nodes from which t is reachable in G_g, and $S_g = V - T_g$. Familiar arguments yield a familiar result, namely that the distance labeling underestimates true distances to the sink in the residual graph:

LEMMA 21. *Throughout the first phase, if d is a valid labeling w.r.t. g, then $d(v)$ is less or equal to the distance from v to t in G_g, for all $v \in V$.*

LEMMA 22. *When the first phase terminates, (S_g, T_g) is a cut such that every pair v, w with $v \in S_g, w \in T_g$ satisfies $g(v, w) = u(v, w)$.*

Proof. Every node $v \in T_g$ has $d(v) < n$ since $d(v)$ is a lower bound on the distance from v to t in G_g, and this distance is either less than n or infinite. Thus every node $v \neq t$ with $e_g(v) > 0$ is in S_g. This includes s, which means that (S_g, T_g) is a cut. If v, w is a pair such that $v \in S_g$ and $w \in T_g$, then $u_g(v, w) = u(v, w) - g(v, w) = 0$ by the definition of T_g. ■

COROLLARY 2. *At the end of the first phase, no node v with $e_g(v) > 0$ can reach t.*

THEOREM 14. *The node partition (S_g, T_g) remains fixed during the second phase. When the second phase terminates, (S_g, T_g) is a minimum cut and g is a maximum flow.*

Proof. Because all excesses are on the source side of the cut (S_g, T_g) at the beginning of the second phase, and this cut is saturated, no excess will ever cross into T_g. Hence this cut is saturated at termination of the second phase, and therefore must be minimum.

Since every excess must be able to reach the source, the second phase maintains the invariant that $e_g(v) > 0$ implies $d(v) < n$, so throughout the second phase, every node with excess is active. This implies that when the second phase terminates, g is a flow. ■

It is a straightforward exercise to see that the time bounds proved for various processing strategies (FIFO, Maximum-Distance Discharge, etc.) apply to the corresponding two-phase implementations as well.

3. Some Remarks on the Two Phase Push-Relabel Algorithm

3.1. Exact Relabeling. It is possible to modify the algorithm so that when a push step is executed, the distance labels are exact distances to the sink in the residual graph. The modification involves a stronger interpretation of the next edge of a node, which requires the next edge to be unsaturated and point to a node with a smaller label. If a push step saturates the next edge of v, a new next edge must be found by scanning the edge list of v and relabeling if the end of the list is reached, like in the push/relabel step. If a relabeling step changes the label of v, the next edge must be updated for all nodes w such that (w, v) is the next edge of w. It is easy to see the above computation takes $O(nm)$ time during the first phase of the algorithm.

It is not clear that this exact labeling strategy really improves the performance of the algorithm, because the work of maintaining the exact labels may exceed the extra work due to inexact labels. However, the above observation suggests that as long as we are interested in an $O(nm)$ upper bound on an implementation of the generic algorithm, we can assume that the exact labeling is given to us for free.

3.2. Efficient practical implementation. There are two refinements that may speed up the algorithm by more quickly showing that nodes with positive excess have no paths to the sink. We will describe these two heuristic improvements in the following:

(1) *Global Relabeling:* We can periodically bring the distance labels up to date by performing a breadth-first-search backward from the sink. Applying this initially and every time the algorithm does cm work, for some constant $c > 1$ will not affect the worst-case running time of the algorithm. (One way to define work is to say that a push counts as one unit of work, and a relabeling of v counts as degree of v units.)
(2) *Gap Relabeling:* We can maintain a set of S of *dead* nodes, those with no paths to the sink. After the edges out of the source are saturated, the source becomes dead. We also maintain, for each possible value of a distance label x, the number of nodes whose distance label is x. It can be easily shown that if for some x this count is zero, no node with a distance label greater then x can reach the sink. If there is exactly one node with distance label of x and this node is relabeled, we can add this node and all nodes with distance label greater than x with are not currently in S

to S. This is what gap relabeling does. Gap relabeling can be implemented so that the worst-case running time of the push-relabel method is not affected.

3.3. What is the best algorithm for the Maximum Flow Problem? According to recent experimental studies, the best algorithm for the maximum flow problem in practice is the maximum distance discharge algorithm, plus the two heuristics. The first-in first-out algorithm with global relabeling is not too bad and is easy to implement.

4. Bipartite Matching Problem

For a bipartite graph $G = (V, E)$, with $V = X \cup Y$, there is a natural correspondence between matching and integral flows through a network G' derived from G by directing all edges, say from X to Y, adding two new nodes, s, t, arcs $(s, x), (y, t)$ for all $x \in X, y \in Y$, and setting all arc capacities equal to one. Also notice that $\forall v \in X$, in-degree$(v) = 1$, and $\forall w \in Y$, out-degree$(w) = 1$.

A matching M in G defines a flow with value $|M|$ on G': saturate all arcs corresponding to M, and all $(s, x), (y, t)$ arcs incident to endpoints of these arcs. Conversely, any integral flow in G' defines a matching in G under the obvious correspondence. The flow value is equal to the matching size, so a maximum flow corresponds to a maximum matching.

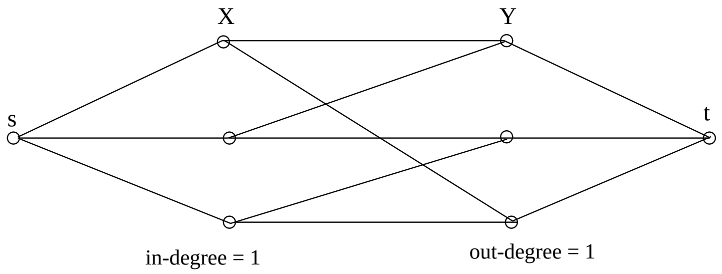


FIGURE VIII.28. Bipartite Matching Network

We will show that if we use global relabeling and a modified *minimum* distance discharge algorithm, the maximum matching can be found in $O(\sqrt{n}m)$ time. This bound is the same as the one for Dinitz' algorithm applied to bipartite matching.

The first phase of the algorithm works like that of the maximum distance discharge algorithm, but an active node with the smallest distance label is always chosen to be discharged, and *discharge*(v) terminates as soon as either the excess of v becomes 0 or v is relabeled. The first phase ends when the node with minimum distance label has distance label at least n (and hence no remaining excess can reach the sink).

LEMMA 23. *Let f^* be an integral maximum flow, let f be an integral preflow, and d be such that $\forall v \in V - \{t\}$, $e_f(v) > 0 \Rightarrow d(v) > \sqrt{n}$, then the residual flow value $|f^*| - e_f(t) = O(\sqrt{n})$.*

Proof. Consider f' that augments f to f^*. Note that t is not reachable from s in G_f, so f' can be decomposed into a collection of paths from nodes with excess to t, paths from nodes with excess to s, and cycles. The number of the paths of the first kind is equal to the residual flow value.

Define the *interior of a path* to be the path without its end nodes. Consider interiors of paths from nodes with excess to the sink. The interior of such a path has length at least $\sqrt{n} - 2$.

After the excess flow at $v \in X \cup Y$ (if any) is moved from v to s or to t along the corresponding paths of f', in the residual graph either in- or out-degree of v is one. From this, it is not hard to see that the path interiors are node-disjoint. Therefore the number of such paths to t (and hence the residual flow value) is at most $\frac{n}{\sqrt{n}-2} = O(\sqrt{n})$. ∎

THEOREM 15. *The first phase of the bipartite matching algorithm works in $O(\sqrt{n}m)$ time.*

Proof. Divide the computation of the first phase into two parts. The first part is up to the point when a node with distance label greater than $\sqrt{n}$ is chosen to be discharged. There are at most $\sqrt{n}$ distance label increases per node and no nonsaturating pushes, so the amount of work done by the algorithm is $O(\sqrt{n}m)$.

At the beginning of the second part of the first phase, the residual flow value is $O(\sqrt{n})$ by lemma 23. Note that immediately after a global update, the algorithm picks a flow excess and pushes it all the way to the sink without any relabelings because the true distances ensure that each node in the residual graph has an outgoing admissible arc, and the minimum distance label selection rule guarantees that no such arc can become saturated before the first excess active after a global update encounters it. Thus each time the algorithm does $O(m)$ work, the residual flow decreases by one. It follows that the second part takes $O(\sqrt{n}m)$ time. ∎

The second phase of the bipartite matching algorithm is very simple and fast. First, all excesses at nodes $y \in Y$ are returned to nodes in X. To see that this is trivial, note that every node with excess in Y must have an outgoing residual arc to a node in X. Note now that all excesses are unit, since the residual indegree of any node in X is zero if it has a unit of excess. Now, to see that it is trivial to return all the excesses to the source from X, note that because the first phase ends the first time the minimum distance label of a node with excess is at least n, $f(s, v) = 1$ for every $v \in X$. Therefore, the nodes with excess can return it directly to s, converting f into a flow. It is easy to perform the second stage in $O(m)$ time.

LECTURE IX

The Minimum Cut Problem

Scribe: Eric Veach

We have already discussed the problem of finding a minimum s-t cut, for a specific choice of s and t. Today we consider the problem of finding the smallest such cut over all choices of s and t, known as the *minimum-cut problem.*

There are several variations on this problem: the graph may be directed or undirected, and the edges may be weighted or unweighted. In the undirected, unweighted case, a minimum cut is the smallest set of edges which, when removed, separate the graph into two or more connected components.

This has obvious applications to network reliability – the minimum cut tells us how many communication links need to fail before the network is separated into two or more pieces. We could even use edge weights, for example to specify how much TNT is required to blow up each link. Then the value of a minimum cut would be the smallest amount of TNT a saboteur needs to smuggle past the security system.

Minimum Cut Algorithms

The simplest approach to computing a min-cut is to simply compute an s-t cut for each pair of nodes (s,t) and then take the smallest of these. This solves the problem using $O(n^2)$ max-flow computations. However it is quite easy to do better than this:

THEOREM 16. *A minimum cut can be computed using $n-1$ minimum s-t cut computations in the undirected case, or $2(n-1)$ computations in the directed case.*

Proof. Consider the undirected case. Our algorithm is to fix a node s, and then compute a minimum s-t cut for each of the other nodes t. We claim that one of these must be a minimum cut. To see this, let $(S, \overline{S})$ be a minimum cut (where $s \in S$). Since $\overline{S}$ is non-empty, it contains a node t. Now $(S, \overline{S})$ is an s-t cut, and thus when we consider s and t, the minimum s-t cut we compute will be at least as small as $(S, \overline{S})$. Of course it cannot be smaller, so we have found a minimum cut.

There is a slight variation where we "glue" s and t together after we compute their minimum s-t cut. That is, we identify s and t together into a single node, leaving multiple arcs in place. We take the new node as the source, pick a new sink, and repeat this until we run out of nodes ($n - 1$ cut computations). We note two facts about this process:

- If two nodes lie on the same side of a cut $(S, \overline{S})$, gluing them together does not alter the cut's value (we have not changed any of the edge weights in the cut).
- Gluing two nodes together can never decrease the size of the minimum cut, since any cut in the new graph is a cut in the original.

To see that this algorithm will find a minimum cut, let $(S, \overline{S})$ be a minimum cut and let t be the first sink chosen which lies on the opposite side of the cut from s. All the previous collapsing has not changed the value of $(S, \overline{S})$, and thus the s-t cut we find will be minimum.

Now consider the directed case. Again we fix a vertex s, and observe that the minimum cut must be of the form $(S, \overline{S})$ or $(\overline{S}, S)$, where $s \in S$. We handle these possibilities separately in two phases. The algorithm described above will find the minimum cut with s on the source side (this is the first phase). To find the minimum cut with s on the sink side, we simply reverse all the arcs of G and run the algorithm again. The minimum cut of G will be the smaller of these two cuts. Note that the same initial vertex s must be used for both runs. ■

We will describe below how the directed min-cut problem can be solved using approximately the same work required for two max-flow computations (in the push-relabel framework).

Note that the undirected min-cut problem appears to be easier than the general max-flow problem. Nagamochi and Ibaraki [33] have an $O(nm + n^2 \log n)$ deterministic algorithm, and David Karger Cliff Stein have recently discovered an $O(n^2 \log^3 n)$ randomized (Monte Carlo) algorithm.

The Hao-Orlin Algorithm

The key idea of the Hao-Orlin algorithm [23] is to choose the next sink in a clever way, so that we can make use of information from the previous max-flow computations. As before, we will fix an initial source s, and run the algorithm in two phases (for the directed case). The first phase finds a minimum cut such that s is on the source side, then we reverse all the arcs and run the algorithm again to find a minimum cut with s on the sink side. It is important that s remain fixed for both phases.

In each phase, we will choose a sequence of sinks $t_1, t_2, \ldots, t_{n-1}$. Note that the sequence will in general be different for the two phases. When t_i is selected as the sink, we start pushing flow towards t_i, stopping as soon as we have discovered a maximum preflow (*i.e.*, no more flow can reach the sink). We can immediately extract a minimum s-t_i cut $(S, \overline{S})$ from the maximum preflow, by defining $\overline{S}$ to be the set of nodes which have a path to t_i in the residual graph. Finally, we merge s and t_i together, and select a new sink.

In more detail, these are the changes we need to make to the standard push-relabel algorithm:

- The algorithm is initialized with $d(s) = n$. As will be shown below, this is sufficient to guarantee that there is never a path from s to t_i in the residual graph.
- To "merge" t_i with s, we simply set $d(t_i) = n$, and then saturate all arcs out of t_i to validate the distance labelling. This effectively makes t_i another source node. If the push-relabel implementation uses dynamic trees, all dynamic tree edges into t_i must be cut before it becomes a source node (this preserves the property that all dynamic tree arcs are admissible).
- We will specify later exactly how to choose the next sink node t_i. However, for the push-relabel algorithm to work correctly while t_i is the sink, we must ensure that the gap $d(s) - d(t_i)$ is large enough such that no s-t_i path can exist in the residual graph G_f (where s is the "merged" source node, consisting of a set of source nodes S).

  Define $\overline{d}(t_i)$ to be the distance label of t_i when t_i is chosen to be the current sink. Our sequence of sink nodes will satisfy the following:

  $$\overline{d}(t_1) = 0 \tag{1}$$

  $$\overline{d}(t_i) \leq 1 + \max_{j<i} \overline{d}(t_j) \quad \forall i \geq 2 \tag{2}$$

  From this it is easy to prove by induction that $\overline{d}(t_i) < i$ for all $i \geq 1$, and thus $d(s) - d(t_i) > n - i$ while t_i is the sink.

  Now suppose there is a simple path P from s to t_i in the residual graph G_f. Observe that G has been reduced to $n - i + 1$ nodes (since the previous $i - 1$ sinks have been merged into s), thus P has at most $n - i$ arcs. But for each arc (v, w) of P we have $d(v) \leq d(w) + 1$, thus $d(s) \leq d(t_i) + (n - i)$, a contradiction. It remains to show how each t_i can be selected to satisfy (2).

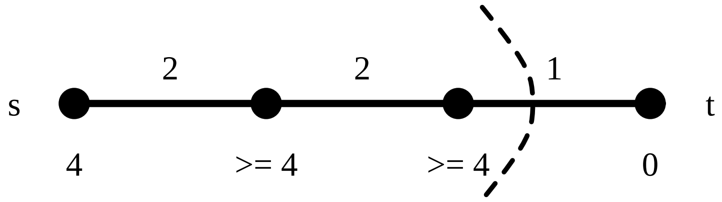


FIGURE IX.29. A problem with selecting the next sink

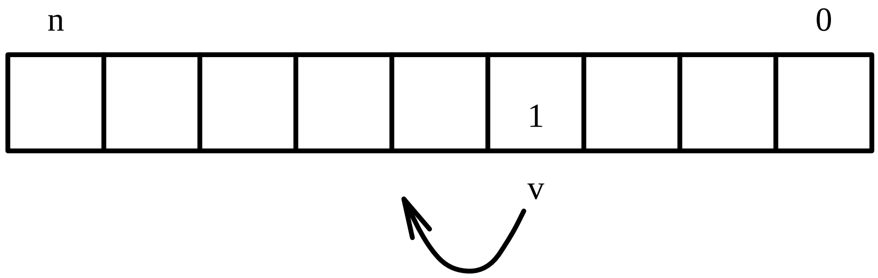


FIGURE IX.30. Creating a gap in the occupancy array

- We will slightly modify the definition of an *active node* and an *admissible arc* as described below.

Selecting the next sink

Consider finding a minimum cut between s and t in the graph of Figure IX.29. The numbers labeling the arcs are the capacities; those labeling the nodes are the distance labels. The usual push-relabel algorithm will terminate with the distance labels shown. At this point, we will merge t with s by setting $d(t) = n$.

But now to satisfy the rules above for choosing the next sink node, we need a sink whose distance label is at most one. No such node exists, so we are stuck. This example illustrates the care with which sink nodes must be selected.

We solve this problem using a variation of the *gap relabelling* idea (first introduced as a heuristic for the two-phase push-relabel algorithm). We define an *occupancy array* which has slots $S[i]$ for all the possible distance labels, $0 \leq i \leq n$. Slot i records the number of nodes whose distance label is i. It is clear that this structure can be maintained without increasing the running time of the algorithm.

Consider the relabelling of some node v, where $d(v) = i$. Suppose this relabelling reduces $S[i]$ to 0 (this is called a *gap*, see Figure IX.30). Then clearly any node w with $d(w) > i$ (including v itself) cannot have a path to the sink in the residual graph G_f, since each arc on the path reduces the distance label by at most one. Thus all such nodes w can be ignored for the purpose of computing a maximum preflow.

In fact, the algorithm will use this property to divide the nodes of G into two groups,

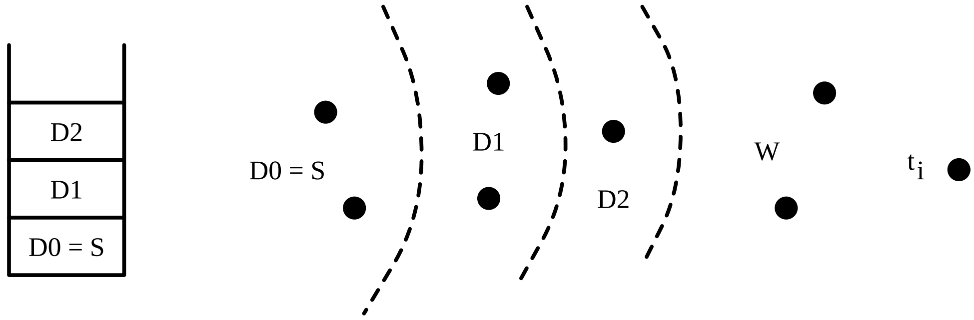


FIGURE IX.31. The stack of dormant nodes

the set D of *dormant* or *frozen* nodes, and the set $W = V - D$ of nodes which are *awake*. We further classify the dormant nodes into a stack $D_0, D_1, \ldots, D_k$, where $D_0 = S$ (the set of source nodes).

When the relabelling of a node v would produce a gap, instead of relabelling v we *freeze* the set of active nodes $w \in W$ with $d(w) \geq d(v)$. This creates a new set of dormant nodes, which we add to the stack (increment k and put them in D_k). These nodes are removed from the occupancy array, and generally ignored until we decide to unfreeze them. In particular, we redefine active nodes and admissible arcs to operate only on W, as follows:

- A node v is *active* if $v \in W \setminus \{t\}$ and $e_f(v) > 0$.
- An arc (v, w) is *admissible* if $v, w \in W$, $u_f(v, w) > 0$, and $d(v) = d(w) + 1$.

We can now describe the method for choosing the next sink. Basically we work with the nodes of W until we run out, then back up to the most recently frozen set of dormant nodes.

(1) Move t_i from W to S as previously described.
(2) If now $W = \emptyset$, set $W = D_k$ and decrement k (pop the top set of dormant nodes from the stack).
(3) Let t_{i+1} be the element of W with the smallest distance label.

Analysis of the algorithm

LEMMA 24. *There is never a residual arc from a node of D to a node of W, or from a node of D_i to a node of D_j where $i < j$.*

Proof. When a set of nodes D_i is frozen, there is no residual arc from D_i to W (there is a gap between their distance labels, and any residual arc decreases d by at most one). We have redefined the admissible arcs so that flow is pushed only between nodes of W, hence this property is maintained as long as D_i is dormant. ■

This means that when finding the minimum s-t_i cut, we need only consider the nodes of

W, since only these nodes could have a path to t_i in the residual graph. Note that in the dynamic tree implementation, there will be no tree edges leaving D because dynamic tree edges are always admissible.

LEMMA 25. *Each sink t_i satisfies $\overline{d}(t_i) \leq 1 + \max_{j<i} \overline{d}(t_j)$.*

Proof.
Case 1. $W \neq \emptyset$ (after t_{i-1} is removed).

Note that the sink's distance label never changes. Let t_i be the node of W with the smallest distance label, and suppose that $d(t_i) > \overline{d}(t_{i-1}) + 1$. Since the occupancy array contains only nodes of W, there must be a gap. But whenever a relabelling would create a gap in W, we prevent it by removing a set of nodes from W and freezing them. Thus such a gap cannot exist, and $d(t_i) \leq \overline{d}(t_{i-1}) + 1$.
Case 2. $W = \emptyset$.

In this case we set $W = D_k$, unfreezing a set of dormant nodes. Let v be the node of W with the smallest distance label, *i.e.*, the unique node whose attempted relabelling caused D_k to be frozen. At that time, W must have contained at least one node w with $d(w) = d(v) - 1$ (since otherwise there would already be a gap). Distance labels never decrease, so when w becomes a sink $\overline{d}(w) \geq d(v) - 1$ will still hold. Since every node of W will be chosen as a sink before v is unfrozen, it follows that $\max_{j<i} \overline{d}(t_j) \geq d(v) - 1$, and thus setting $t_i = v$ will give the desired bound. ∎

Running Time. The distance label for any node v never exceeds n, thus there are $O(n^2)$ relabellings. We can analyze the time complexity exactly as before, including the dynamic tree implementation. Note that the work for freezing and unfreezing is $O(n^2)$, since we freeze a node at most once per sink. This gives an $O(n^3)$ algorithm for the min-cut problem, which can be reduced to $O(nm \log(n^2/m))$ using dynamic trees.

LECTURE X

Minimum Cost Flow and Assignment Problems

Scribe: Makoto Hoketsu

1. Minimum Cost Flows

The *minimum cost flow problem* is as follows: given an instance of the maximum flow problem and function $c : E \to \mathcal{R}$, where $c(a)$ represents the cost per unit if flow on a, find a maximum flow that has the minimum total cost. The function c has the antisymmetry property $c(a) = -c(a^R)$ and we define the cost of a pseudoflow f to be

$$c(f) = \frac{1}{2}\sum_{a \in E} c(a)f(a) = (\text{by antisymmetry}) \sum_{a \in E, f(a)>0} c(a)f(a).$$

A minimum cost flow problem can be transformed into a *minimum cost circulation problem.*

DEFINITION 9. *A pseudoflow that satisfies conservation constraints at every node of the graph is called a* ***circulation****.*

In formulating the circulation problem, we allow arcs to have a lower bound on capacity in addition to an upper bound (as a result, not all circulation problems have a solution). The capacity bounds have the antisymmetry property $u(v,w) = -l(w,v)$. We can think about specifying the desired upper and lower bounds in a preferred direction; then the bounds in the reverse direction are implicit.

We can find if there exist a feasible flow or not using a maximum flow algorithm. Without loss of generality, we assume that $l(a) \le u(a)$ for every $a \in E$.

We can convert a circulation problem (V, E, l, u) into the maximum flow problem $(V \cup \{s,t\}, E', u')$, where $E' \supset E$. For each arc $(v,w) \in E$ we define

$$u'(v,w) = \begin{cases} u(v,w) & \text{if } l(v,w) \ge 0 \le u(v,w), \\ u(v,w) - l(v,w) & \text{otherwise} \end{cases}$$

For arcs $(v, w) \in E$ with $l(v, w) > 0$ we add arcs (s, w) and (v, t) to E' and define $u'(s, w) = l(v, w)$ and $u'(v, t) = l(v, w)$. We also add symmetric arcs with antisymmetric capacities. (Intuitively, we are providing a flow of $l(v, w)$ "around" arcs (v, w) with positive lower capacity bounds.)

LEMMA 26. *The minimum cut in the transformed network is trivial (contains only the node s) if and only if the original problem is feasible. The resulting maximum flow gives a solution to the circulation problem.*

EXERCISE 6. *Prove the lemma.*

We can transform the minimum cost flow problem into the minimum cost circulation problem as follows. First keep in mind that we can find a feasible flow using a maximum flow algorithm. In the following discussion, we will assume that there exists a feasible circulation. We add a new edge X from sink t to source s in G with $u(X) = mU$, $l(X) = 0$ and $c(X) = -nC - 1$, where $U = \max_{a \in E}\{u(a)\}$ and $C = \max_{a \in E}\{c(a)\}$ (Figue X.32). Now

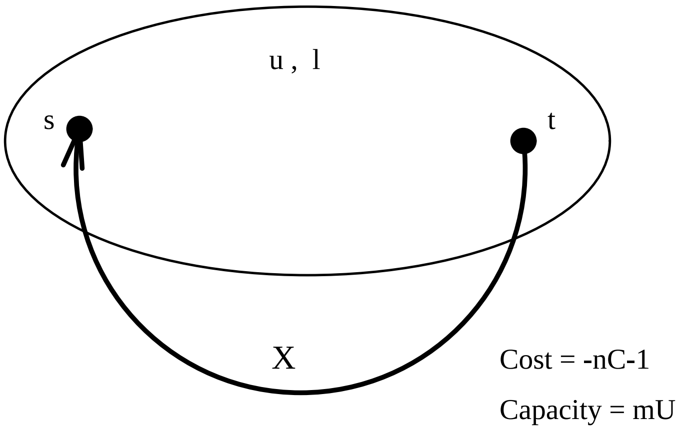


FIGURE X.32. Adding an arc from s to t.

Circulations are more uniform objects than flows.

THEOREM 17. *Every circulation can be decomposed into at most m cycles.*

DEFINITION 10. *A **negative cycle** is a cycle Γ that has a negative cost.*

THEOREM 18. *A circulation f is optimal if and only if there are no negative cycles in G_f.*

Proof. Clearly, if $\Gamma \in G_f$ is a negative cycle, we can increase the flow of each arc $a \in \Gamma$ by the amount $\min_{a \in \Gamma} \{u_f(a)\}$; the resulting pseudoflow is still a circulation and it has a lower cost than before, so f is not optimal.

In the other direction, let f^* be some optimal circulation and suppose that f is not optimal, *i.e.*, $c(f) > c(f^*)$. Now $f^* - f$ is a collection of cycles (the sum of two circulations is still a circulation, and a circulation with all of its flow reversed is still a circulation). But the sum of the costs of these cycles is negative, because $c(f^*) - c(f) < 0$, so there is at least one negative cost cycle. Since this negative cycle is in $f^* - f$, it must be in G_f. Therefore, nonoptimal flow f will always have negative cycles. ■

This theorem leads us to *Klein's Algorithm*:

While ∃ negative residual cycle, find one and cancel it.

Note that we can find a negative cycle using the shortest path algorithm. Sounds like you are looking for a augmenting path? You are right. By adding an arc (t, s) with capacity mU and a cost of -1 to an instance of maximum flow problem with cost of all arcs set to 0, this algorithm turns out to be equivalent to the Ford-Fulkerson augmenting path algorithm for the maximum flow problem. We know this is not very efficient.

It can be shown that if we find a cycle with the minimum mean cost using the algorithm described a couple of lectures ago, this algorithm for the circulation problem becomes strongly polynomial [19]. The corresponding algorithm for finding the maximum flow is the shortest augmenting path algorithm.

Ford and Fulkerson studied the minimum-cost flow problem in the 50's [10]. The first polynomial-time algorithm was discovered in the early 70's by Edmonds and Karp [8] and independently by Dinits [47]. The first strongly polynomial algorithm is due to Tardos [43].

2. Prices and Reduced Costs

Next we add a function $p : V \to \mathcal{R}$, representing the price per unit of commodity at the nodes. We define the *reduced cost* to be $c_p(v, w) = p(v) + c(v, w) - p(w)$. Intuitively, this is the cost of buying a unit of commodity at v, shipping it to w, and then selling it at w.

LEMMA 27. *For any cycle Γ and price function p, $c(\Gamma) = c_p(\Gamma)$.*

THEOREM 19. *A circulation f is optimal if and only if there is a price function p such that $(v, w) \in G_f$ implies that $c_p(v, w) \geq 0$.*

Proof. Suppose that such a cost function exists. Then there are no residual arcs of negative reduced cost, thus no residual cycles with negative reduced cost. Hence by Theorem 17 and Lemma 27, the circulation is optimal.

For the other direction, suppose that f is optimal. We add a new node s and connect it with zero-cost arcs to every node in the graph. Then we compute d, the shortest (minimum cost) path distance function from s in G_f. Since there are no negative cycles, d is well-defined. We know that $d(w) \leq d(v) + c(v, w)$ if $(v, w) \in G_f$ but then $c_d(v, w) \geq 0$. Clearly, d qualifies for the desired function. ■

3. Capacity Scaling

Let f, u, and l be integer-valued functions. Suppose that f is optimal with respect to p. If we add an arc (v, w) of unit capacity and arbitrary cost, then an integral optimal flow can be found using one application of Dijkstra's shortest-path algorithm (which runs in $O(m + n \log n)$ time using Fibonacci heaps) as follows.

(1) Compute the shortest path distance from w with respect to c_p in $G_f - (v, w)$.
(2) Set $p' = p + d$.
(3) If $d(v) + c(v, w) \geq 0$ then f is optimal with respect to p.
(4) Otherwise, let $\Gamma = (sp(w, v) \cup (v, w))$, where $sp(w, v)$ is the shortest path from w to v. Increase flow on Γ by 1.

Proof of correctness. Before the new arc is added, there are no negative arcs with respect to p', so there cannot be any negative cycles. Note that we can increase flow on Γ by one, because the flow and capacities are integral, and if we do so we saturate the new arc. Thus, if any negative cycles remain after the augmentation they can only be from the creation of a residual arc of negative reduced cost. But if $(x, y) \in \Gamma$ then $d(y) = d(x) + c_p(x, y)$, so $c_{p'}(x, y) = 0$. Thus, there are no negative cycles after the Step 4. ■

Without loss of generality, assume that a flow of zero is feasible. Assume that the largest capacity is U. Let P_i be the problem obtained from the input problem by rounding the input capacities to integer multiples of 2^i. We round upper bounds down, lower bounds up. Note that the above assumption implies that the upper bounds are nonnegative and lower bounds nonpositive, so the rounding preserves capacity antisymmetry.

Solution to P_0 is trivial. Given a solution of P_{i-1}, we solve P_i by adding a unit-capacity arc (v, w) for every arc which had a 1 in the ith bit of its capacity and then making the circulation optimal using the above approach. Doing this at most m times solves the problem P_i. Thus the time to solve the input problem is $O(m(m + n \log n) \log U)$.

4. Assignment Problem

The assignment problem is as follows. Given an undirected bipartite graph $G = (S \bigcup T, E)$ and the weight function $w : E \to \mathcal{R}$, find a maximum matching with the minimum weight.

The assignment problem can be reduced to the minimum cost circulation problem. We construct a circulation network G' from G by directing all arcs from S to T and assigning their costs to their weights. Add source s and sink t, zero cost arcs (s, v_1), (v_2, t) for all $v_1 \in S$, $v_2 \in T$. Set capacities of all arcs to one. We also add a return arc (t, s) of capacity m and cost $-nC - 1$, where C is absolute value of the biggest input edge weight. All lower bounds of capacities are set to zero, and for every arc, a reverse arc is added (with cost and capacities implied by antisymmetry). (Figure X.33)

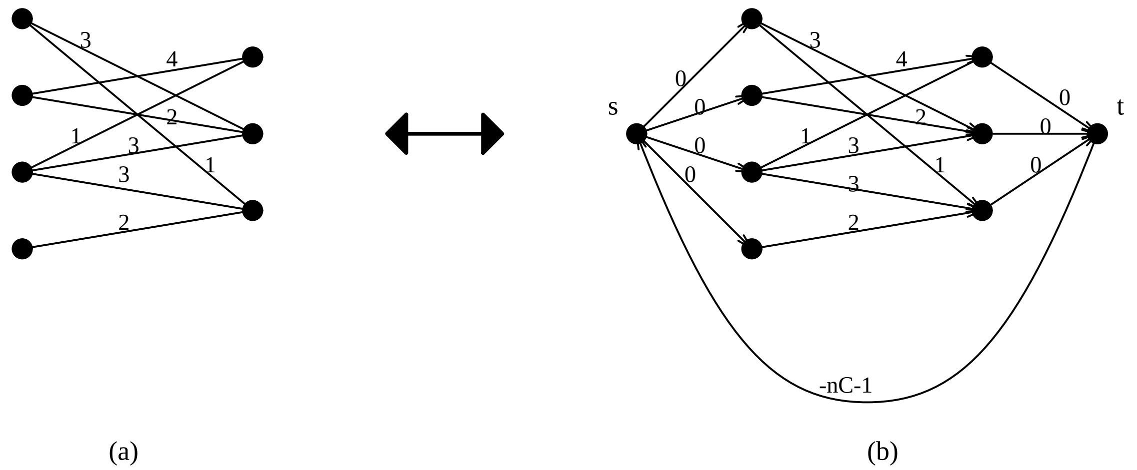


FIGURE X.33. (a) input graph G. (b) corresponding circulation network.

EXERCISE 7. *Show that there is a one-to-one correspondence between matchings in the input network and integral flows in the transformed network, and maximum matchings of minimum cost correspond to minimum-cost circulations.*

Note that the capacity-scaling algorithm introduced earlier solves the minimum-cost circulation problem on G' in $O(m \log n)$ shortest path computations, because $U = n$. We can do better, however.

Suppose we remove all arcs from s to nodes in S, and then add the arcs back one by one. Initially, the zero circulation is optimal w.r.t. the zero price function; after an arc addition, we restore optimality as described above (note that the added arcs have unit capacity). Since we add $|S| \leq n$ arcs, and the time to restore optimality at each iteration is dominated by a shortest path computation, we have the following result.

THEOREM 20. *The assignment problem can be solved in $O(n)$ shortest path computations on graph.*

The first algorithm to achieve this bound was the *Hungarian Method* [28].

LECTURE XI

Cost Scaling for Min-Cost Circulation Problem

Scribe: Michael Goldwasser

1. Cost Scaling

Next we describe cost-scaling algorithms for the minimum-cost circulation problem. The first cost-scaling algorithm is due to Röck [38] and independently by Bland & Jansen [2]. The method we study is due to Goldberg and Tarjan [20].

Approximate Optimality. A key notion is that of *approximate optimality*, obtained by relaxing the complementary slackness constraints. For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function* p if, for every arc (v, w), we have

$$f(v,w) < u(v,w) \Rightarrow c_p(v,w) \geq -\epsilon \quad (\epsilon\text{-optimality constraint}). \tag{3}$$

A pseudoflow f is *ϵ-optimal* if f is ϵ-optimal with respect to some price function p.

An important property of ϵ-optimality is that if the arc costs are integers and ϵ is small enough, any ϵ-optimal circulation is minimum-cost.

THEOREM 21. *If all costs are integers, $\epsilon < 1/n$, and circulation f is ϵ-optimal w.r.t. price function p, then f circulation f is optimal. [N.B. not necessarily w.r.t same p]*

Proof. We show that no negative cost residual cycles can exist. Let Γ be any simple cycle in G_f. Then $c(\Gamma) = c_p(\Gamma) = \sum_{a \in \Gamma} c_p(a) \geq |\Gamma| \cdot (-\epsilon) \geq -n\epsilon > -1$. Since $c(\Gamma)$ must be integral, then $c(\Gamma) \geq 0$. ∎

procedure ***min-cost***(V, E, u, c);
  [initialization]
  $\epsilon \leftarrow C$;
  $\forall v, \quad p(v) \leftarrow 0$;
  $f \leftarrow$ ***feasible circulation***;
  [loop]
  while $\epsilon \geq 1/n$ **do**
    $(\epsilon, f, p) \leftarrow$ ***refine***(ϵ, f, p);
  return(f);
end.

FIGURE XI.34. The cost-scaling method.

2. The Algorithm

We give here a high-level description of the cost-scaling method (see Figure XI.34). The algorithm maintains an error parameter ϵ, a circulation f and a price function p, with the invariant that f is ϵ-optimal with respect to p. The algorithm assumes we are given some feasible circulation, and then sets $\epsilon = C$ (or alternatively $\epsilon = 2^{\lceil \log_2 C \rceil}$) and $p(v) = 0$ for all $v \in V$. Any circulation is C-optimal, and so the invariant is initially true. The main loop of the algorithm repeatedly reduces the error parameter ϵ. When $\epsilon < 1/n$, the current circulation is minimum-cost, and the algorithm terminates.

The task of the subroutine *refine* is to reduce the error in the optimality of the current circulation. The input to *refine* is an error parameter ϵ, a circulation f, and a price function p such that f is ϵ-optimal with respect to p. The output from *refine* is a reduced error parameter ϵ, a new circulation f, and a new price function p such that f is ϵ-optimal with respect to p. The implementation of *refine* described here reduces the error parameter ϵ by a factor of two.

The correctness of the algorithm is immediate from Theorem 21, assuming that *refine* is correct. The number of iterations of *refine* is $O(\log(nC))$. This gives us the following theorem:

THEOREM 22. *A minimum-cost circulation can be computed in the time required for* $O(\log(nC))$ *iterations of refine, if refine reduces* ϵ *by a factor of at least two.*

3. Refinement Based on Push-Relabel Operations

Next we describe an implementation of *refine* that is a generalization of the push-relabel maximum flow algorithm.

The effect of *refine* is to reduce ϵ by a factor of two while maintaining the ϵ-optimality of the current flow f with respect to the current price function p. The generic *refine* subroutine is described on Figure XI.35. It begins by halving ϵ, hence satisfying the reduction on the

procedure $refine(\epsilon, f, p)$;
  [initialization]
  $\epsilon \leftarrow \epsilon/2$;
  $\forall (v,w) \in E$ **do if** $c_p(v,w) < 0$ **then begin** $f(v,w) \leftarrow u(v,w); f(w,v) \leftarrow -u(v,w)$; **end;**
  [loop]
  while $\exists$ **an update operation that applies do**
    select such an operation and apply it;
  return(ϵ, f, p);
end.

FIGURE XI.35. The generic *refine* subroutine.

$push(v,w)$.
Applicability: v **is active and** (v,w) **is admissible.**
Action: **send** $\delta = \min(e_f(v), u_f(v,w))$ **units of flow from** v **to** w.

$relabel(v)$.
Applicability: v **is active and** $\forall w \in V$ $(u_f(v,w) = 0$ **or** $c_p(v,w) \geq 0)$.
Action: **replace** $p(v)$ **by** $\max_{(v,w)\in E_f}\{p(w) - c(v,w) - \epsilon\}$.

FIGURE XI.36. The update operations.

error. However the circulation is no longer guaranteed to be ϵ optimal after the reduction, and so the routine then saturates every arc with negative reduced cost. This converts the circulation f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow), however it is no longer guaranteed to be a circulation. Finally, the subroutine converts the ϵ-optimal pseudoflow into an ϵ-optimal circulation by applying a sequence of the *update operations* *push* and *relabel*, each of which preserves ϵ-optimality.

The inner loop of the generic algorithm consists of repeatedly applying the two update operations, described in Figure XI.36, in any order, until no such operation applies. To define these operations, we need to redefine admissible arcs in the context of the minimum-cost circulation problem. Given a pseudoflow f and a price function p, we say that an arc (v,w) is *admissible* if (v,w) is a residual arc with negative reduced cost. In similar fashion as previous flow algorithms, we say a node is *active* if it has positive excess.

A *push* operation applies to an admissible arc (v,w) such that node v is active. It consists of pushing $\delta = \min\{e_f(v), u_f(v,w)\}$ units of flow from v to w, thereby decreasing $e_f(v)$ and $f(w,v)$ by δ and increasing $e_f(w)$ and $f(v,w)$ by δ. The push is *saturating* if $u_f(v,w) = 0$ after the push and *nonsaturating* otherwise.

A *relabel* operation applies to an active node v that has no exiting admissible arcs. It consists of decreasing $p(v)$ to the smallest value allowed by the ϵ-optimality constraints, namely $\max_{(v,w)\in E_f}\{-c(v,w) + p(w) - \epsilon\}$.

If an ϵ-optimal pseudoflow f is not a circulation, then either a pushing or a relabeling operation is applicable. It is easy to show that any pushing operation preserves ϵ-optimality. The next lemma gives two important properties of the relabeling operation.

LEMMA 28. *Suppose f is an ϵ-optimal pseudoflow with respect to a price function p and a node v is relabeled. Then the price of v decreases by at least ϵ and the pseudoflow f is ϵ-optimal with respect to the new price function p'.*

Proof. Before the relabeling, $c_p(v, w) \geq 0$ for all $(v, w) \in E_f$, *i.e.*, $p(v) \geq p(w) - c_p(v, w)$ for all $(v, w) \in E_f$. Thus $p'(v) = \max_{(v,w)\in E_f} \{p(w) - c(v, w) - \epsilon\} \leq p(v) - \epsilon$.

To verify ϵ-optimality, observe that the only residual arcs whose reduced costs are affected by the relabeling are those of the form (v, w) or (w, v). Any arc of the form (w, v) has its reduced cost increased by the relabeling, preserving its ϵ-optimality constraint. Consider a residual arc (v, w). By the definition of p', $p'(v) \geq p(w) - c(v, w) - \epsilon$. Thus $c_{p'}(v, w) = c(v, w) + p'(v) - p(w) \geq -\epsilon$, which means that (v, w) satisfies its ϵ-optimality constraint. ∎

Since the update operations preserve ϵ-optimality, and since some update operation applies if f is not a circulation, it follows that if *refine* terminates and returns (ϵ, f, p), then f is a circulation which is ϵ-optimal with respect to p. Thus *refine* is correct.

Next we analyze the number of update operations that can take place during an execution of *refine*. We begin with a definition. The *admissible graph* is the graph $G_A = (V, E_A)$ such that E_A is the set of admissible arcs. As *refine* executes, the admissible graph changes. An important invariant is that the admissible graph remains acyclic.

LEMMA 29. *Immediately after a relabeling is applied to a node v, no admissible arcs enter v.*

Proof. Let (u, v) be a residual arc. Before the relabeling, $c_p(u, v) \geq -\epsilon$ by ϵ-optimality. By Lemma 28, the relabeling decreases $p(v)$, and hence increases $c_p(u, v)$, by at least ϵ. Thus $c_p(u, v) \geq 0$ after the relabeling. ∎

COROLLARY 3. *Throughout the running of* refine*, the admissible graph is acyclic.*

Proof. Initially the admissible graph contains no arcs and is thus acyclic. Pushes obviously preserve acyclicity. Lemma 29 implies that relabelings also preserve acyclicity. ∎

LEMMA 30. *Let f be a pseudoflow and f' a circulation. Then for any node v with $e_f(v) > 0$, there is a node w with $e_f(w) < 0$ and a simple path Γ from v to w such that $\Gamma \in E_f$ and $\Gamma^R \in E_{f'}$.*

Proof. Let v be a node with $e_f(v) > 0$. Define $G_+ = (V, E_+)$, where $E_+ = \{a|f'(a) > f(a)\}$, and define $G_- = (V, E_-)$, where $E_- = \{a|f'(a) < f(a)\}$. Note $E_+ \subseteq E_f$ and $E_- \subseteq E_{f'}$. Furthermore, $(x, y) \in E_+ \Leftrightarrow (y, x) \in E_-$ by antisymmetry. Thus to prove the lemma it suffices to show the existence of some node with deficit excess which is reachable from v in G_+.

Let S be the set of vertices reachable from v in G_+ and let $\overline{S}$ be the complement of S. Then for all $(x, y) \in (S, \overline{S})$, $f(x, y) \geq f'(x, y)$, for otherwise $y \in S$. Now consider

$$\begin{aligned}
0 &= \textstyle\sum_{(x,y)\in(S,\overline{S})\cap E} f'(x, y) && \text{since } f' \text{ is a circulation} \\
&\leq \textstyle\sum_{(x,y)\in(S,\overline{S})\cap E} f(x, y) && \text{holds term by term} \\
&= \textstyle\sum_{(x,y)\in(S,\overline{S})\cap E} f(x, y) + \sum_{(x,y)\in(S,S)\cap E} f(x, y) && \text{by antisymmetry} \\
&= \textstyle\sum_{(x,y)\in(S,V)\cap E} f(x, y) && \text{by definition of } \overline{S} \\
&= -\textstyle\sum_{x\in S} e_f(x) && \text{by antisymmetry}
\end{aligned}$$

Since $v \in S$, $e_f(v) > 0$ so some node $w \in S$ must have $e_f(w) < 0$ as desired. ∎

LEMMA 31. *The price of a node v decreases by at most $3n\epsilon$ during an execution of refine.*

Proof. Let f' and p' be the input circulation and price functions on entry to refine such that f' is 2ϵ-optimal with respect to p'. Suppose a relabeling causes the price of a node v to decrease. Let f be the pseudoflow and p the price function just after the relabeling. Then $e_f(v) > 0$. Let $\Gamma \subseteq E_f$ be the simple path from v to w whose existence was shown by the previous lemma.

By ϵ-optimality of f, $-|\Gamma|\epsilon \leq \sum_{(x,y)\in\Gamma} c_p(x, y) = p(v) - p(w) + c(\Gamma)$.

By 2ϵ-optimality of f', $-2|\Gamma|\epsilon \leq \sum_{(y,x)\in\Gamma^R} c_{p'}(y, x) = p'(w) - p'(v) + c(\Gamma^R)$.

But $c(\Gamma) = -c(\Gamma^R)$ by cost antisymmetry. Furthermore, $p(w) = p'(w)$ since during refine, the initialization step is the only one that makes the excess of some vertices negative, and a node with negative excess has the same price as long as its excess remains negative. We now add the two inequalities above to obtain $-3|\Gamma|\epsilon \leq p(v) - p'(v)$. Rearrange to get $p'(v) - p(v) \leq 3|\Gamma|\epsilon \leq 3n\epsilon$ as desired. ∎

LECTURE XII

Cost Scaling and Strongly Polynomial Algorithms

Scribe: Jeremy Gaston

1. Overview

In this lecture we continue discussing the *cost-scaling* approach for the minimum-cost circulation problem. We then discuss how to make it strongly polynomial.

2. Cost-Scaling

Last time we showed that the price of a node v decreases by at most $3n\epsilon$ during an execution of *refine*. We now can count update operations.

LEMMA 32. *The number of relabelings during an execution of refine is* $O(n^2)$.

Proof. During one execution of refine, each node decreases in price by at most $3n\epsilon$. Each relabelling decreases price by ϵ or more, as we showed last time. Thus there are at most $3n$ relabelings per node and $3n(n-1)$ in total, hence $O(n^2)$. ■

Note that the work required for these relabelings is $O(nm)$, for the same reasons as in the max flow case.

LEMMA 33. *The number of saturating pushes during an execution of refine is at most* $3nm$, *hence* $O(nm)$.

Proof. Same as in the max flow case. ■

LEMMA 34. *The number of nonsaturating pushes during one execution of refine is at most* $3n^2(m+n)$, *hence* $O(n^2m)$.

Proof. For each vertex v, let $\Phi(v)$ be the number of vertices reachable from v in G_A, the admissible graph of residual arcs with negative reduced costs. Let $\Phi = 0$ if there are no active vertices, and let $\Phi = \sum\{\Phi(v)|v \text{ is active}\}$ otherwise. Throughout the running of refine, $\Phi \geq 0$. Initially $\Phi \leq n$, since G_A has no arcs (all negative reduced cost arcs were saturated) and so each active node can only reach itself.

Consider the effect on Φ of update operations. A nonsaturating push decreases Φ by at least one, for the following reason: suppose we perform a nonsaturating push across arc (v, w). Then we remove $\Phi(v)$ from our summation and at most add $\Phi(w)$. But $\Phi(w) < \Phi(v)$ since G_A is always acyclic by a previous lecture's corollary, so our summation decreases by at least one.

A saturating push can increase Φ by at most n, since at most one inactive vertex becomes active, and it can reach at most n nodes. Finally, if a vertex v is relabeled, Φ also can increase by at most n, since $\Phi(w)$ for $w \neq v$ can only decrease, since after w is relabelled, it has no admissible arcs entering it; thus, there are no new paths from any v to any u through w. The total number of nonsaturating pushes is thus bounded by the initial value of Φ plus the total increase in Φ throughout the algorithm, *i.e.*, by $n + 3n^2(n-1) + 3n^2m \leq 3n^2(m+n)$. ■

2.1. Maximum Node Discharge and Wave Methods. In these algorithm variations, we maintain the nodes in topologically sorted order, which is possible since the admissible graph is acyclic. We then define *discharge* as before, in which we take a node and push from it until we can't push anymore, and then we relabel it, if it still has excess. Next we will define the notion of a *maximal node*. In the admissible graph, a node is maximal if there is no path to it from any other node. In the topological sort, this would correspond to the leftmost node in a linear ordering of the sort. This ordering can easily be maintained by push and relabel operations. When we push, we don't need to do anything; when we relabel a node, it has no incoming arcs, so it can be placed in the leftmost position.

There are two obvious ways we can process this linear topological list of nodes. In either case, we march down the list, and if a node is active, we discharge it. If we have to relabel it, we either (1) restart at the leftmost node – the newly relabelled one – (Maximal Node Discharge algorithm), or (2) we keep marching down the list and only go back to the leftmost one when we've finished discharging the rest of the active nodes to the right (Wave algorithm). The following analyses apply to both methods. These analyses depend on the notion of a *phase*, which is simply the period between two relabelings, as defined before. In either algorithm, during a phase, we only move rightward along the list of nodes.

LEMMA 35. *Discharge uses relabel correctly.*

Proof. Same as in the max flow case. ■

As before, we process nodes in topological order with respect to the admissible graph G_A.

LEMMA 36. *There are $O(n^3)$ nonsaturating pushes in the wave/maximal-node-discharge implementation of refine.*

Proof. Same as in the max flow case. (Basically we have $O(n^2)$ phases, and during each one we have only one nonsaturating push per node.) ■

THEOREM 23. *The running time for the wave/maximal-node-discharge implementation is $O(n^3 \log Cn)$.*

Proof. We have $O(\log Cn)$ iterations and by the previous lemma, we have $O(n^3)$ nonsaturating pushes in each iteration, so the total running time is $O(n^3 \log Cn)$. ■

2.2. Running Times. We now list running times for several methods; some of these methods use sophisticated data structures.

$O(nm \log \frac{n^2}{m} \log nC)$	cost scaling [20]
$O(m \log U(m + n \log n))$	capacity scaling [8, 47]
$O(nm \log nC \log \log U)$	cost and capacity scaling [1]
$O(m \log n(m + n \log n))$	capacity scaling with edge contractions [34]

3. Strongly Polynomial Algorithms

We now discuss several strongly polynomial algorithms for the minimum-cost circulation problem based on cost scaling.

We start by taking a closer look at the notion of ϵ-optimality.

3.1. Fitting Price Functions and Tight Error Parameters. The definition of ϵ-optimality motivates the following two problems:

1. Given a pseudoflow f and a constant $\epsilon \geq 0$, find a price function p such that f is ϵ-optimal w.r.t. p, or show that there is no such price function.

2. Given a non-optimal pseudoflow f, find the smallest $\epsilon \geq 0$ such that f is ϵ-optimal.

LEMMA 37. *A graph G contains a shortest path tree if and only if G does not contain a negative-cost cycle. A spanning tree T rooted at s is a shortest path tree if and only if $c(v, w) + d(v) \geq d(w)$ for every arc (v, w) in G.*

Define a new cost function $c^{(\epsilon)}: E \to R$ by $c^{(\epsilon)}(v,w) = c(v,w) + \epsilon$. Extend the residual graph G_f by adding a single vertex s and arcs from it to all other vertices to form an *auxiliary graph* $G_{aux} = (V_{aux}, E_{aux}) = (V \cup \{s\}, E_f \cup (\{s\} \times V))$. Extend $c^{(\epsilon)}(v,w)$ to G_{aux} by defining $c^{(\epsilon)}(s,v) = 0$ for every arc (s,v), where $v \in V$.

THEOREM 24. *Pseudoflow f is ϵ-optimal if and only if G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost. If T is any shortest path tree of G_{aux} with respect to the arc cost function $c^{(\epsilon)}$, and d is the associated tree cost function, then f is ϵ-optimal with respect to the price function p defined by $p(v) = d(v)$ for all $v \in V$.*

Proof. Suppose f is ϵ-optimal. Any cycle in G_{aux} is a cycle in G_f, since vertex s has no incoming arcs. Let Γ be a cycle of length l in G_{aux}. Then $c(\Gamma) \geq -l\epsilon$, which implies $c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon \geq 0$. Therefore G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost.

Suppose G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost. Then by the previous lemma, G_{aux} has some shortest path tree rooted at s. Let T be any such tree and let d be the tree cost function. By the previous lemma, $c^{(\epsilon)}(v,w) + d(v) \geq d(w)$ for all $(v,w) \in E_f$, which is equivalent to $c(v,w) + d(v) - d(w) \geq -\epsilon$ for all $(v,w) \in E_f$. But these are the ϵ-optimality constraints for the price function $p = d$. Thus f is ϵ-optimal with respect to p. ∎

Using the theorem, we can solve the first problem by constructing G_{aux} and finding either a shortest path tree or a negative-cost cycle. Constructing G_{aux} takes $O(m)$ time. Finding a shortest path tree or a negative-cost cycle takes $O(nm)$ time using the Bellman-Ford shortest path algorithm. Now we show how to solve the second problem in $O(nm)$ time as well.

3.2. Tight Error Parameters. For $\epsilon > 0$, we say that f is ϵ-tight if f is ϵ-optimal and for any $\epsilon' < \epsilon$, f is not ϵ'-optimal. For a circulation f we define $\epsilon(f)$ to be zero if f is optimal and to be the unique number $\epsilon' > 0$ such that f is ϵ'-tight otherwise. We use $\epsilon(f)$ as a measure of the quality of f. Let $\mu(G,l)$ denote the minimum-mean cost of a cycle in G w.r.t. cost function c. Recall that the minimum-mean cycle can be found in $O(nm)$ time.

THEOREM 25. *Suppose a pseudoflow f is not optimal. Then $\epsilon(f) = -\mu(G_f, c)$.*

Proof. Assume f is not optimal. Consider any cycle Γ in G_f. Let the length of Γ be l. For any ϵ, let $c^{(\epsilon)}$ be the cost function defined above: $c^{(\epsilon)}(v,w) = c(v,w) + \epsilon$ for $(v,w) \in E_f$. Let ϵ be such that f is ϵ-tight, and let $\mu = \mu(G_f, c)$. By the theorem proved above, $0 \leq c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon$, *i.e.*, $c(\Gamma)/l \geq -\epsilon$. Since this is true for any cycle Γ, $\mu \geq -\epsilon$, *i.e.*, $\epsilon \geq -\mu$. Conversely, for any cycle Γ, $c(\Gamma)/l \geq \mu$, which implies $c^{(-\mu)}(\Gamma) \geq 0$. This implies $-\mu \geq \epsilon$. ∎

The following observation is helpful in the analysis to follow. Suppose f is an ϵ-tight pseudoflow and $\epsilon > 0$. Let p be a price function such that f is ϵ-optimal with respect to p, and let Γ be a cycle in G_f with mean cost $-\epsilon$. Since $-\epsilon$ is a lower bound on the reduced cost of an arc in G_f, every arc of Γ must have reduced cost of exactly $-\epsilon$.

3.2.1. Fixed Arcs

Intuitively, the main idea behind the strongly-polynomial bound is the observation that if an arc cost is "large" compared to ϵ, then all ϵ-optimal circulations must have the same value on this arc, and therefore refine does not change flow on this arc. We cannot use the input cost of arcs, however, since cost of any specific arc can be forced to any desired value by changing price of an end point of the arc. We use *optimal* reduced costs c^* instead, where $c^* = c_{p^*}$ for an optimal price function p^* (*i.e.,* there is a circulation f^* that is optimal w.r.t. p^*). Note that the algorithm does not "know" c^*, which is introduced only for the sake of analysis.

THEOREM 26. *Suppose an arc a is such $|c^*(a)| > n\epsilon$. Then, for any two ϵ-optimal circulations f', f'', we have $f'(a) = f''(a)$.*

Proof. By antisymmetry, it is enough to prove the theorem for the case $c_p(v, w) > n\epsilon$. (If flow is fixed on an arc, it's fixed on the reverse arc.) Let f^* be optimal with respect to p^*; note that $f^*(a) = l(a)$, otherwise the reverse arc would have negative reduced cost and residual capacity, implying that the flow is not optimal. Note that f^* is trivially ϵ-optimal for ϵ. Now, it is enough to show that if f is a circulation such that $f(a) \neq f^*(a)$, then f is not ϵ-optimal.

Consider $G_> = \{x \in E | f(x) > f^*(x)\}$. Note that $G_>$ is a subgraph of G_{f^*}, and a is an arc of $G_>$. Since f and f^* are circulations, $G_>$ must contain a simple cycle Γ that passes through a. Let l be the length of Γ. Since all arcs of Γ are residual arcs in G_{f^*}, the cost of Γ is at least $c(a) > n\epsilon$. Now consider the cycle $\overline{\Gamma}$ obtained by reversing the arcs on Γ. Note that $\overline{\Gamma}$ is a cycle in $G_< = \{x \in E | f(x) < f^*(x)\}$ and is therefore a cycle in G_f. By antisymmetry, the cost of $\overline{\Gamma}$ is less than $-n\epsilon$ and thus the mean cost of $\overline{\Gamma}$ is less than $-\epsilon$. Theorem 25 implies that f is not ϵ-optimal. ∎

We say that an arc $a \in E$ is *ϵ-fixed* if $|c^*(a)| > n\epsilon$. The above theorem says that all ϵ-optimal circulations agree on ϵ-fixed arcs. Thus if all arcs are fixed, the circulation must be optimal. Define F_ϵ to be the set of ϵ-fixed arcs. Since the cost-scaling method decreases ϵ, an arc that becomes ϵ-fixed stays ϵ-fixed. We show that when ϵ decreases by a factor of n, a new arc becomes ϵ-fixed.

procedure ***min-cost***(V, E, u, c);
    [initialization]
    $\epsilon \leftarrow C$;
    $\forall v, \quad p(v) \leftarrow 0$;
    $\forall (v, w) \in E, \quad f(v, w) \leftarrow 0$;
    [loop]
    while $\epsilon > 0$ **do begin**
(*)   **find** λ **and** p_λ **such that** f **is** λ**-tight with respect to** p_λ;
        if $\lambda > 0$ **then** $(\epsilon, f, p) \leftarrow$ ***refine***(λ, f, p_λ)
        else return(f);
end.

FIGURE XII.37. The strongly polynomial algorithm.

LEMMA 38. *Assume* $\epsilon' < \frac{\epsilon}{n}$. *Suppose that there exists an* ϵ*-tight circulation* f. *Then* F'_ϵ *properly contains* F_ϵ.

Proof. Since every ϵ'-optimal circulation is ϵ-optimal, we have $F_\epsilon \subseteq F'_\epsilon$. To show that the containment is proper, we have to show that there is an ϵ'-fixed arc that is not ϵ-fixed.

Since the circulation f is ϵ-tight, there exists a price function p such that f is ϵ-optimal with respect to p, and there exists a simple cycle Γ in G_f every arc of which has reduced cost of $-\epsilon$, by an earlier observation. Since increasing f along Γ preserves ϵ-optimality, the arcs of Γ are not ϵ-fixed.

We show that at least one arc of Γ is ϵ'-fixed. Let f^* be as above. Since the mean cost of Γ is $-\epsilon$, there is an arc x of Γ with $c^*(x) \leq -\epsilon < -n\epsilon'$. This arc is ϵ-fixed. ■

3.3. Strongly Polynomial Cost Scaling. Our original cost scaling method has the disadvantage that the number of iterations of *refine* depends on the magnitudes of the costs. If the costs are huge integers, the method need not run in time polynomial in n and m; if the costs are irrational, the method need not even terminate. However, a natural modification of the method produces strongly polynomial algorithms.

The main idea of this modification is to improve ϵ periodically by finding a price function that fits the current circulation better than the current price function. The changes needed to make the cost-scaling approach strongly polynomial are to add an extra computation to the main loop of the algorithm and to change the termination condition. Before calling *refine* to reduce the error parameter ϵ, the new method computes the value λ and a price function p_λ such that the current circulation f is λ-tight with respect to p_λ. The strongly polynomial method is described in Figure XII.37. The algorithm terminates when the circulation f is optimal, *i.e.*, $\lambda = 0$.

The time to perform line (*) is $O(nm)$. Since the implementation of *refine* that we have considered has a time bound greater than $O(nm)$, the time per iteration in the new version of the algorithm exceeds the time per iteration in the original version by less than a constant factor. Since each iteration at least halves ϵ, the bound of $O(\log(nC))$ on the number of iterations derived before remains valid, assuming that the costs are integral. For arbitrary real-valued costs, we shall derive a bound of $O(m \log n)$ on the number of iterations.

THEOREM 27. *The total number of iterations of the while loop in procedure* min-cost *is* $O(m \log n)$.

Proof. Consider a time during the execution of the algorithm. During the next $O(\log n)$ iterations, either the algorithm terminates, or the error parameter is reduced by a factor of n. In the latter case, Lemma 38 implies that an arc becomes fixed. If all arcs become fixed, the algorithm terminates in one iteration of the loop. Therefore the total number of iterations is $O(m \log n)$. ■

The best strongly polynomial implementation of the generalized cost-scaling method, due to Goldberg and Tarjan, is based on the dynamic tree implementation of *refine*, and runs in $O(nm^2 \log(n^2/m) \log n)$ time.

As a side note, it can also be shown that if f is ϵ-optimal w.r.t. p and $|c_p(a)| > 2n\epsilon$, then arc a is ϵ-fixed, by a proof similar to the one for Theorem 26. This is a practical result, because we know $c_p(a)$, whereas we don't actually know $c^*(a)$. Implementing this fact can allow us to get rid of arcs as they become fixed, and such implementations can speed up the algorithm.

The following material was not given in class, but its result (that Klein's cycle-canceling algorithm can now be made strongly polynomial) was briefly mentioned.

4. Minimum-Mean Cycle-Canceling

In this section we use ideas of behind the analysis of cost scaling to show that Klein's cycle-canceling algorithm becomes strongly polynomial if a careful choice is made among possible cycles to cancel. Recall that Klein's algorithm consists of repeatedly finding a residual cycle of negative cost and sending as much flow as possible around the cycle. This algorithm can run for an exponential number of iterations if the capacities and costs are integers, and it need not terminate if the capacities are irrational. Goldberg and Tarjan [19] showed that if a cycle with the minimum mean cost is canceled at each iteration, the algorithm becomes strongly polynomial. We call the resulting algorithm the *minimum-mean cycle-canceling* algorithm.

The minimum-mean cycle-canceling algorithm is closely related to the shortest augmenting path maximum flow algorithm. The relationship is as follows. If a maximum flow problem is formulated as a minimum-cost circulation problem in a standard way, then

Klein's cycle-canceling algorithm corresponds exactly to the augmenting path maximum flow algorithm, and the minimum-mean cycle-canceling algorithm corresponds exactly to the shortest augmenting path algorithm.

Let f be an arbitrary circulation, let $\epsilon = \epsilon(f)$, and let p be a price function with respect to which f is ϵ-optimal. Holding ϵ and p fixed, we study the effect on $\epsilon(f)$ of a minimum-mean cycle cancellation that modifies f. Since all arcs on a minimum-mean cycle have negative reduced cost with respect to p, cancellation of such a cycle does not introduce a new residual arc with negative reduced cost, and hence $\epsilon(f)$ does not increase.

LEMMA 39. *A sequence of m minimum-mean cycle cancellations reduces $\epsilon(f)$ to at most $(1 - 1/n)\epsilon$,* i.e.*, to at most $1 - 1/n$ times its original value.*

Proof. Let p a price function such that f is ϵ-tight with respect to p. Holding ϵ and p fixed, we study the effect on the admissible graph G_A (with respect to the circulation f and price function p) of a sequence of m minimum-mean cycle cancellations that modify f. Initially every arc $(v, w) \in E_A$ satisfies $c_p(v, w) \geq -\epsilon$. Canceling a cycle all of whose arcs are in E_A adds only arcs of positive reduced cost to E_f and deletes at least one arc from E_A. We consider two cases.

Case 1: None of the cycles canceled contains an arc of nonnegative reduced cost. Then each cancellation reduces the size of E_A, and after m cancellations E_A is empty, which implies that f is optimal, *i.e.*, $\epsilon(f) = 0$. Thus the lemma is true in this case.

Case 2: Some cycle canceled contains an arc of nonnegative reduced cost. Let Γ be the first such cycle canceled. Every arc of Γ has a reduced cost of at least $-\epsilon$, one arc of Γ has a nonnegative reduced cost, and the number of arcs in Γ is at most n. Therefore the mean cost of Γ is at least $-(1 - 1/n)\epsilon$. Thus, just before the cancellation of Γ, $\epsilon(f) \leq (1 - 1/n)\epsilon$ by Theorem 25. Since $\epsilon(f)$ never increases, the lemma is true in this case as well. ■

Lemma 39 is enough to derive a polynomial bound on the number of iterations, assuming that all arc costs are integers.

THEOREM 28. *If all arc costs are integers, then the minimum-mean cycle-canceling algorithm terminates after $O(nm \log(nC))$ iterations.*

An argument similar to that of the proof of Theorem 27 yields the following result.

THEOREM 29. *For arbitrary real-valued arc costs, the minimum-mean cycle-canceling algorithm terminates after $O(nm^2 \log n)$ cycle cancellations.*

Theorem 30. *The minimum-mean cycle-canceling algorithm runs in $O(n^2m^3\log n)$ time on networks with arbitrary real-valued arc costs, and in $O(n^2m^2\min\{\log(nC), m\log n\})$ time on networks with integer arc costs.*

Proof. Immediate from Theorems 28 and 29. ■

Radzik and Goldberg [37] show that the bounds stated in Theorems 28 and 30 can be improved by a factor of $\log n$, and the improved bounds are tight up to a constant factor.

LECTURE XIII

The Multicommodity Flow & Concurrent Flow Problems

Scribe: David Chang

1. The Multicommodity Flow Problem

The *multicommodity flow problem* is a classical problem of finding feasible flow on a network with commodity demands (each commodity having its own source and sink) and capacity constraints. An instance of the multicommodity flow problem consists of a graph $G = (V, E)$, capacity function $u : E \to R^+$, and demand specifications for each commodity. For each commodity $i = (1, \ldots, k)$, the specification contains the source s_i, the sink t_i, and a nonnegative demand d_i. We assume that $m \geq n$, G has no parallel arcs, and capacities and demands are integers. A multicommodity flow $f = (f_1, \ldots, f_k)$ is *feasible* if:

(1) Commodity demands are satisfied with *conservation* by every commodity i at every node v

$$\sum_{(u,v)\in E} f_i(u, v) - \sum_{(v,w)\in E} f_i(v, w) = \begin{cases} d_i & \text{if } v = t_i, \\ -d_i & \text{if } v = s_i, \\ 0 & \text{otherwise.} \end{cases}$$

(2) *Capacity constraints* are satisfied at every arc a

$$f(s) = \sum_{i=1,\ldots,k} f_i(a) \leq u(v, w).$$

Essentially, we have a network with many commodities, with every node being a possible source or sink for any given commodity. The problem is to find a feasible flow which ships the demand for each commodity from its source to its sink, without overflowing the shared capacities. In the case of only one commodity, this problem is very similar to the max-flow problem except instead of trying to find the maximum flow, we are looking for a flow of value d_i.

Note that although the multicommodity flow problem looks like a max-flow problem, there are two important aspects of the max-flow problem that do not apply. First, the max-flow/min-cut theorem does not hold. Second, the integrality theorem does not hold. Figure XIII.38 demonstrates these facts. Notice that every nontrivial min-cut has capacity of 1. However, each unit of commodity flowing from its source to its destination uses two units of available capacity and the total available capacity is 3. Therefore if all demands are set to 1, the problem is not feasible even though the total demand across any cut is equal to the capacity of the cut.

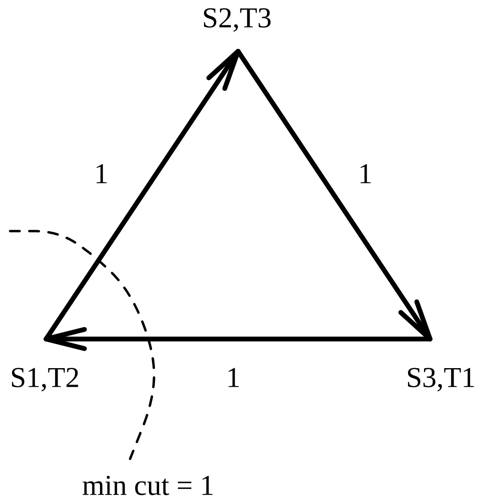


FIGURE XIII.38. Multicommodity problem with three commodities

We can generalize the multicommodity flow problem by adding a cost per unit of flow $c(a)$ for every arc a, and asking for a feasible flow of minimum cost. An even more general problem has (potentially different) costs $c_i(a)$ for every commodity.

2. Concurrent Flow Problem

The *concurrent flow problem* is an optimization version of the multicommodity flow problem. Instead of finding a feasible flow or showing that a feasible flow does not exist, the the goal is to maximize z, the fraction of the commodity demands, such that the problem with demands zd_i is feasible. An equivalent problem is to minimize λ such that the problem with demands d_i and capacities $\lambda u(a)$ is feasible. We will look at an algorithm minimizing λ.

2.1. Definitions and Notation.

DEFINITION 11. *The <u>congestion</u> of an arc is defined by* $\lambda(a) = f(a)/u(a)$.

Let $\lambda = \max_{a \in E} \lambda(a)$. Certainly if we increased capacities by the factor of λ, we would have a feasible multicommodity flow. However, it is conceivable that the flows can then be rerouted so that we do not have to increase capacities so much. Let λ^* be the optimal value of λ, or the minimum value by which we can multiply the capacities and still get a feasible problem.

Definition 12. *The length function $\ell : E \to R^+$ is a nonzero, nonnegative function.*

Definition 13. *The $dist_\ell(v, w)$ denotes the shortest path distance in G from v to w with respect to ℓ.*

To simplify notation, we sometimes view length, capacity, and flow functions as vectors indexed by arcs. Given two vectors a and b, let $a \circ b = \sum_{(v,w) \in E} a(v,w)b(v,w)$. We also use the standard scalar-vector multiplication.

Lemma 40. *For a multicommodity flow f satisfying demands d_i and capacities λu and a length function ℓ we have*

$$\lambda u \circ \ell \geq \sum_{i=1}^{k} dist_\ell(s_i, t_i) d_i.$$

Proof. $\lambda u \circ \ell$ represents the total "volume" of network (each term is "width" of an arc times its length) while $\sum_{i=1}^{k} dist_\ell(s_i, t_i) d_i$ represents the lower bound on flow volume since each commodity i flows on paths which are at least as long as the shortest path. ■

Definition 14. *Given a length function ℓ, we define the cost of the flow of commodity i by $C_i = f_i \circ \ell$.*

Let $C_i^*(\lambda)$ be the value of the minimum-cost flow f_i satisfying the demands of commodity i with costs ℓ and capacities λu.

Theorem 31. *For f, C_i, and $C_i^*(\lambda)$ as above,*

$$\lambda u \circ \ell \geq \sum_{i=1}^{k} C_i \geq \sum_{i=1}^{k} C_i^*(\lambda).$$

A multicommodity flow f minimizes λ iff there is a length function ℓ for which the above terms are equal.

Proof. The first inequality is follows since $\lambda u \circ \ell$ represents the total volume of the network and $\sum_{i=1}^{k} C_i$ represents the flow volume for some flow. The second inequality hold term by term since cost of a flow is at least as big as that of the minimum cost flow.

The second claim of the lemma is not needed for the analysis of the algorithm we are interested in. Its proof is left as an exercise. ■

COROLLARY 4. *For f, ℓ as above,*

$$\frac{\sum_{i=1}^{k} C_i^*(\lambda)}{u \circ \ell}$$

is a lower bound on λ^*.

Proof. From Theorem 31, we have

$$\lambda^* \ell \circ u \geq \sum_{i=1}^{k} C_i^*(\lambda^*) \geq \sum_{i=1}^{k} C_i^*(\lambda).$$

Rewriting, we get

$$\lambda^* \geq \frac{\sum_{i=1}^{k} C_i^*(\lambda)}{\ell \circ u}.$$

■

2.2. Relaxed Optimality. A multicommodity flow is ϵ-optimal if the following *relaxed optimality conditions* are satisfied.

$$\forall (v,w) \in E \quad (1+\epsilon) f(v,w) \geq \lambda u(v,w) \quad \text{or} \quad u(v,w)\ell(v,w) \leq \frac{\epsilon}{m}(u \circ \ell). \tag{4}$$

$$(1-2\epsilon)\sum_i C_i \leq \sum_i C_i^*(\lambda). \tag{5}$$

The first condition states that either an arc is close to being saturated, or its volume is small compared to the total volume. The second condition states that the total cost of f with respect to ℓ is close to optimal. From this point forward, we will assume that $\epsilon \leq 1/8$.

THEOREM 32. *If f, ℓ, and ϵ satisfy the relaxed optimality conditions, then λ is at most $(1+5\epsilon)\lambda^*$.*

Proof. From condition (4), we have

$$\sum_{(v,w)\in E} \left((1+\epsilon) f(v,w)\ell(v,w) + \lambda \frac{\epsilon}{m} u \circ \ell\right) \geq \lambda u \circ \ell.$$

since both $(1+\epsilon) f(v,w)\ell(v,w) \geq \lambda u \circ \ell$ and $\lambda \frac{\epsilon}{m} u \circ \ell \geq \lambda u \circ \ell$.
Rewriting, we get

$$(1+\epsilon) f \circ \ell \geq \lambda u \circ \ell (1-\epsilon)$$

or

$$\frac{1-\epsilon}{1+\epsilon} \lambda u \circ \ell \leq f \circ \ell = \sum_i C_i \leq \frac{1}{1-2\epsilon} \sum C_i^*(\lambda). \tag{6}$$

By Corollary 1 and the previous inequality,

$$\lambda^* \geq \frac{\sum_i C_i^*(\lambda)}{u \circ \ell} \geq \frac{(1-\epsilon)(1-2\epsilon)}{1+\epsilon} \lambda \geq (1-4\epsilon)\lambda.$$

Since $\epsilon \leq 1/8$, this implies $\lambda \leq (1 + 5\epsilon)\lambda^*$. ■

LECTURE XIV

The Concurrent Flow Algorithm

Scribe: Matt Carlyle

1. Preliminaries

Recall from last time that we had two conditions for ϵ-optimality in the concurrent flow problem:

$$\forall a \in E \quad (1+\epsilon)f(a) \geq \lambda u(a) \quad \text{or} \quad u(a)\ell(a) \leq \frac{\epsilon}{m}(u \circ \ell). \tag{7}$$

$$(1-2\epsilon)\sum_i C_i \leq \sum_i C_i^*(\lambda). \tag{8}$$

The first condition states that either an arc is close to being saturated, or its "volume" is small compared to the total "volume". The second condition states that the total cost of f with respect to ℓ is close to the optimal.

1.1. Exponential Length Function. The algorithm we develop for solving this problem to ϵ-optimality requires a length function which will help ensure that conditions 1 and 8 are met. One such function was proposed by Shahrokhi and Matula [39], who were the first to use an exponential length in the context of uniform capacities. We use the generalization of this idea which was introduced by Leighton et. al. [30]

Define

$$\ell_f(a) = \frac{e^{\alpha\lambda(a)}}{u(a)} \tag{9}$$

where

$$\lambda(a) = \frac{f(a)}{u(a)} \tag{10}$$

is the load on arc a, and λ is the maximum such load over all arcs.

Note that this penalizes large loads severely; this leads us to try to find an α large enough to force condition 1 to be satisfied. In other words, we want to find an α so that if $(1+\epsilon)f(a) < \lambda u(a)$ then $u(a)\ell(a) \leq \frac{\epsilon}{m}(u \circ \ell)$

Lemma 41. *If*

$$\alpha \geq \frac{(1+\epsilon)}{\lambda\epsilon}\ln(\frac{m}{\epsilon}) \tag{11}$$

then condition 1 *holds for* (f, ℓ_f).

Proof. We can rewrite

$$\begin{aligned} \frac{\epsilon}{m}(u \circ \ell) &= \frac{\epsilon}{m}\sum_{a \in E} e^{\alpha\lambda(a)} \\ &\geq \frac{\epsilon}{m} e^{\alpha\lambda} \end{aligned}$$

and since, by assumption, arc a is not close to being saturated,

$$\begin{aligned} u(a)l(a) &= e^{\alpha\lambda(a)} \\ &\leq e^{(\alpha\lambda)/(1+\epsilon)} \end{aligned}$$

so we now want:

$$\begin{aligned} \ln(\frac{\epsilon}{m}) + \alpha\lambda &\geq \frac{\alpha\lambda}{(1+\epsilon)} \\ \alpha\lambda(1 - \frac{1}{1+\epsilon}) &\geq \ln(\frac{m}{\epsilon}) \\ \alpha &\geq \frac{(1+\epsilon)}{\lambda\epsilon}\ln(\frac{m}{\epsilon}). \end{aligned}$$

■

2. The Algorithm

The algorithm maintains a multicommodity flow f satisfying the demands and the corresponding exponential length function ℓ. By Lemma 41, the first relaxed optimality condition is always satisfied. It starts with an initial solution whose associated λ is bounded by $k\lambda^*$, and iteratively improves the current solution until its value is within a factor of ϵ of our optimal flow value. It accomplishes this by using a potential function to ensure a rapid decrease in the current value of λ.

2.1. Potential Function. The algorithm will attempt to minimize λ indirectly, by actually minimizing a potential function that is dominated by the largest of the $\lambda(a)$ values. We define this function as follows:

$$\Phi = u \circ \ell_f \tag{12}$$

When Φ is defined in this manner it is easy to see that the values of $\lambda(a)$ which are close to the maximum will dominate. Hence, minimizing Φ effectively minimizes λ.

2.2. Initial Solution. The algorithm starts with an initial solution f such that $\lambda \leq k\lambda^*$. Such a solution is obtained by finding, for each commodity i, a maximum flow g_i in the network with source s_i, sink t_i, and capacity function u, and setting $f_i(a) = g_i(a)\frac{d_i}{|g_i|}$, where $|g_i|$ denotes the value of g_i. It is easy to see that for the resulting flow f, $\lambda \leq k\lambda^*$. The length function ℓ defined by the initial flow is also computed.

2.3. Improving the Current Solution. At each iteration, the algorithm iteratively improves the current flow, until both conditions for ϵ-optimality are satisfied. The main loop of the algorithm is as follows (parameters α, λ_0 and σ will be explained in the following section):

(1) If condition (8) holds then terminate.
(2) Compute f_i^*, the minimum-cost flow for commodity i, $\forall i \in K$, with capacities λu and costs ℓ_f.
(3) Compute $f^{(i)}$, the multiflow for commodity i, $\forall i \in K$, where

$$f^{(i)} = (f_1, f_2, \dots, (1-\sigma)f_i + \sigma f_i^*, \dots, f_k) \tag{13}$$

(4) Let i be the index of the $f^{(i)}$ which minimizes $\Phi = u \circ \ell_{f^{(i)}}$.
(5) $f \leftarrow f^{(i)}$.
(6) Reset parameters λ_0, α, and σ, as described below.
(7) Goto step 1.

The parameters λ_0, α and σ are set as follows. Let λ_0 be the value of λ after initialization or during the last change in α and σ. The values are set to $\alpha = 2(1+\epsilon)\lambda_0^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$ and $\sigma = \frac{\epsilon}{4\alpha\lambda_0}$, and are updated when λ decreases to $\lambda_0/2$ or less.

2.4. Algorithm Analysis. Intuitively, an iteration of the algorithm replaces a σ fraction of a commodity by the same fraction of the optimal flow of this commodity.

The next lemma shows that when f_i is rerouted, Φ decreases by almost $\alpha\sigma(C_i - C_i^*(\lambda))$. Let Φ and Φ_a be the values of the potential function before and after the rerouting, respectively.

Lemma 42.

$$\Phi - \Phi_a \geq \alpha\sigma(C_i - C_i^*(\lambda) - \epsilon C_i).$$

Proof. Let ℓ and ℓ' be the length functions before and after the rerouting. By Taylor's theorem, for $|t| \leq \epsilon/4 \leq 1/4$ we have

$$e^{x+t} \leq e^x + te^x + \frac{\epsilon}{2}|t|e^x.$$

Taking $x = \frac{\alpha\sigma}{u(a)}f(a)$ and $t = \frac{\alpha\sigma}{u(a)}(f_i^*(a) - f_i(a))$, we obtain

$$\ell'(a) \leq \ell(a) + \frac{\alpha\sigma}{u(a)}(f_i^*(a) - f_i(a))\ell(a) + \frac{\epsilon\alpha\sigma}{2u(a)}|f_i^*(a) - f_i(a)|\ell(a).$$

We have

$$\begin{aligned} \Phi - \Phi_a &= (\ell - \ell') \circ u \\ &\geq \alpha\sigma(f_i - f_i^*) \circ \ell - \frac{\epsilon\alpha\sigma}{2}|f_i^* - f_i| \circ \ell \\ &\geq \alpha\sigma(C_i - C_i^*(\lambda)) - \epsilon\alpha\sigma C_i. \end{aligned}$$

The last line follows by definition of C_i and $C_i^*(\lambda)$ and by the fact that $C_i \geq C_i^*(\lambda) \geq 0$. ■

Note that if the current flow does not satisfy the second relaxed optimality condition, then the value of $C_i - C_i^*(\lambda)$ is large for some i and Φ decreases significantly. The following lemma formalizes this statement.

Lemma 43. *Suppose $\sigma = \Theta\left(\frac{\epsilon}{\alpha\lambda}\right)$ and f does not satisfy the second relaxed optimality condition. Then the decrease in Φ due to a rerouting step is $\Omega(\frac{\epsilon^2}{k}\Phi)$.*

Proof. Since the maximum decrease in Φ is at least the average, we have

$$\begin{aligned} \Phi - \Phi_a &\geq \frac{1}{k}\alpha\sigma\sum_i(C_i - C_i^*(\lambda) - \epsilon C_i) \\ &= \frac{\alpha\sigma}{k}(\sum_i((1-2\epsilon)C_i - C_i^*(\lambda)) + \epsilon\sum_i C_i) \\ &= \frac{\epsilon\alpha\sigma}{k}\sum_i C_i \\ &\geq \frac{\alpha\sigma}{k}\frac{1-\epsilon}{1+\epsilon}\epsilon\lambda\Phi \\ &= \Omega(\frac{\epsilon^2}{k}\Phi). \end{aligned}$$

The fourth line follows from the assumption that f does not satisfy the second relaxed optimality condition. The last line follows from the assumption on σ. ■

LEMMA 44. *The concurrent flow algorithm as presented here runs in*

$$O\left(\frac{k^2}{\epsilon^3}\log k \log\frac{m}{\epsilon}\right) \tag{14}$$

min-cost flow computations.

Proof. Lemma 43 implies that, after i iterations, Φ decreases by a factor of $\Omega[(1-\frac{\epsilon^2}{k})^i]$, which means that, after $O(\frac{k}{\epsilon^2})$ iterations Φ will decrease by a constant factor. An iteration of the algorithm uses k min-cost flow computations.

Also recall that

$$\begin{aligned}\Phi &= u\circ\ell\\ &= \sum_a u(a)\ell_f(a)\\ &= \sum_a e^{\alpha\lambda(a)}\end{aligned}$$

so

$$\Phi \le me^{\alpha\lambda_0}$$

and

$$\Phi \ge e^{\frac{\alpha\lambda_0}{2}}$$

so Φ can change by at most $me^{\frac{\alpha\lambda_0}{2}}$.

Because of this, the algorithm will halt after $O(\frac{k}{\epsilon^2}\log me^{\frac{\alpha\lambda_0}{2}})$ iterations.

$$\log\left(me^{\frac{\alpha\lambda_0}{2}}\right) = \log m + \frac{1+\epsilon}{\epsilon}\log\frac{m}{\epsilon}$$

so the algorithm halts after $O(\frac{k}{\epsilon^3}\log\frac{m}{\epsilon})$ iterations.

We have to perform the algorithm $\log k$ times, each iteration of which is k min-cost flow computations.

Therefore the total number of min-cost-flow computations is $O(\frac{k^2}{\epsilon^3}\log k\log\frac{m}{\epsilon})$

■

It can be shown that if a commodity is selected at each iteration uniformly at random and rerouted if Φ decreases, then the bound on the expected number of iterations remains the same, but each iteration uses one min-cost flow computation instead of k. See [15].

LECTURE XV

The General Matching Problem

Scribe: Sandeep Juneja

Matching in bipartite graphs has already been discussed in earlier lectures. Today we extend the discussion to include non-bipartite matching on undirected graphs. The references for this lecture are [7, 35, 44]. The lecture was given by Sherry Listgarten.

1. The Matching Problem

A *matching* M of a graph $G(V, E)$ is a subset of edges with the property that no two edges of M share an endpoint. In Figure 1, for example, $M_1 = \{[b, c], [d, e], [f, h], [g, j]\}$ and $M_2 = \{[a, b], [c, e], [d, g], [f, h], [i, j]\}$ are matchings. The size of a matching is the number of edges in the matching; the *maximum matching* problem is that of finding a matching of maximum size. It is clear that for any matching M, $|M| \leq |V|/2$. In Figure 1, $|V| = 10$ and $|M_2| = 5$, so M_2 is a maximal matching. Edges in a matching M are called *matched edges*. All other edges are called *free* (or *unmatched*) *edges*. If $\{u, v\}$ is a matched edge then v is the *mate* of u (i.e., $v = mate(u)$). All nodes that are not incident upon any matched edge are called *free*. The remaining nodes are *matched*.

A path (resp. cycle) in a graph with matching M is called an *alternating path (resp. cycle)* if its edges alternate between matched and free. An *augmenting path* is a simple alternating path that begins and ends at free nodes. In Figure 1, path $p = [a, d, e, f, h, g, j, i]$ is an augmenting path.

Theorem 33. *A matching M in $G(V, E)$ is maximal $\Longleftrightarrow$ There are no augmenting paths in $G(V, E)$.*

Proof. $\Rightarrow$ Suppose there exists an augmenting path P with respect to a matching M in $G(V, E)$. Then a matching of size $|M| + 1$ can be constructed by simply matching the unmatched edges of P, and unmatching the matched edges of P. More formally, the symmetric difference $M \oplus P$, the set of edges in exactly one of M and P, is a matching of size $|M| + 1$. The size is as claimed because P has one more unmatched edge than it has matched edges. The set of edges is a matching because $M \backslash (P \cap M)$ is a matching, the set of unmatched edges in P is a matching, and the endpoints of P are free with respect to M. Thus M is not a maximum matching.

$\Leftarrow$ Suppose M is a maximum matching, while matching M^- is not (i.e., $|M| > |M^-|$). We will show that there is an augmenting path with respect to M^- in $G = (V, E)$ by

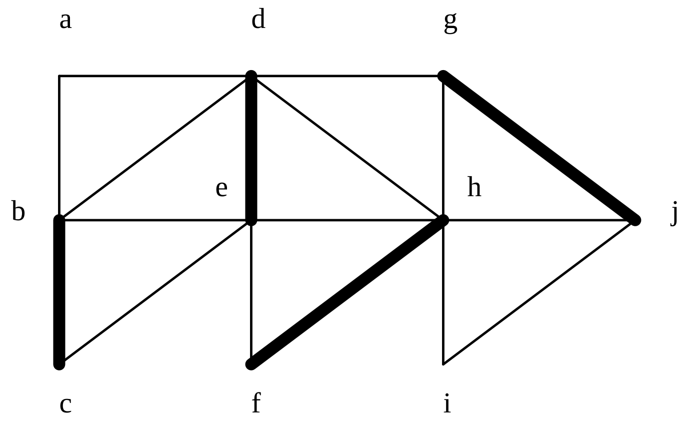


FIGURE XV.39. Example of a Matching

showing that one exists in the subgraph $G' = (V, M \oplus M^-)$. Each vertex in G' has degree at most two, since it can be adjacent to at most one of M and M^-. So G' consists of paths and cycles, with edges alternately from M and M^-. Since cycles have an equal number of edges from each matching, and since $|M| > |M^-|$, there must be a path in G' that begins and ends with an edge in M. This is an augmenting path with respect to M^-. ■

Hence we have the following algorithm to find a maximal matching.

```
Input: M = ∅
WHILE there is an augmenting path P with respect to M DO
    M ← M ⊕ P
```

Since $|M| \leq |V|/2$, at most $n/2$ iterations are required in the above algorithm.

2. Finding Augmenting Paths

There remains the problem of finding augmenting paths. The natural way to find one is to search from a free vertex, advancing only along alternating paths until another free node is encountered in the path. Unfortunately, using a simple tree search to implement this approach does not always work for non-bipartite matching. In Figure 2, for instance, we may start at a free node a, travel along the path $[a, c, d, e, f]$, and "discover" that there is no augmenting path from a, having seen all nodes reachable from that vertex by an alternating path. But in fact there is an augmenting path, namely $[a, c, d, f, e, g]$. The problem stems from the existence of an odd cycle ($[d, e, f]$ in this case), which belongs to a class of odd cycles called *blossoms*.

DEFINITION 15. Blossom*: a blossom B in a graph $G(V, E)$ with matching M is an odd cycle with r matched edges and $r + 1$ unmatched edges ($r \geq 1$) that is reachable from some*

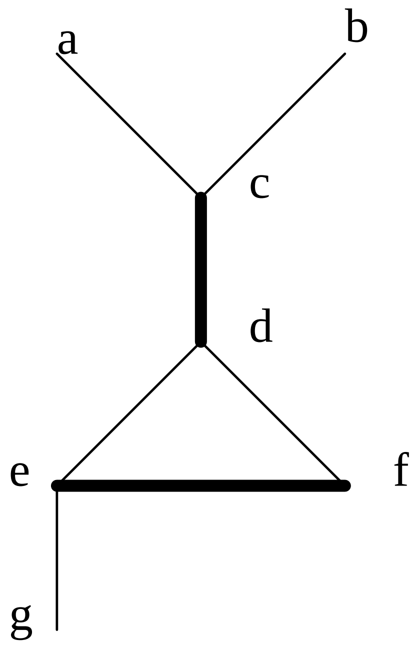


FIGURE XV.40. Cycle [d,e,f] is a blossom

free vertex by an even-length alternating path.

DEFINITION 16. Base of a blossom*: the vertex of the blossom where the two unmatched edges meet (i.e., where the even-length alternating path mentioned in the above definition meets the blossom).*

Edmonds [3] discovered a solution to the difficulty of doing a quick tree search in the presence of blossoms. His algorithm searches through a graph from its free vertices, and shrinks each blossom it discovers into a single "condensed" vertex. This is illustrated in Figure 3. Note that if b is such a condensed vertex, the new graph contains an edge $\{v, b\}$ iff v was not in the blossom, but was connected to some vertex that was in the blossom.

DEFINITION 17. Shrunken blossom*: a single vertex in the shrunken graph corresponding to a blossom in the original graph.*

We would like to know that shrinking blossoms does not affect our ability to find augmenting paths in the original (unshrunken) graph. This is done with the following theorem:

THEOREM 34. $\exists$ *an augmenting path a graph* $G \Longleftrightarrow \exists$ *an augmenting path in the graph* G' *formed by shrinking some blossom in* G.

$\Longleftarrow$ **Proof.** We consider graph G' which differs from G in the respect that a blossom in G exists in G' as a shrunken node b. Let P be an augmenting path in G'. If P does not contain b then P is an augmenting path in G as well. Assume P contains b, and let x be

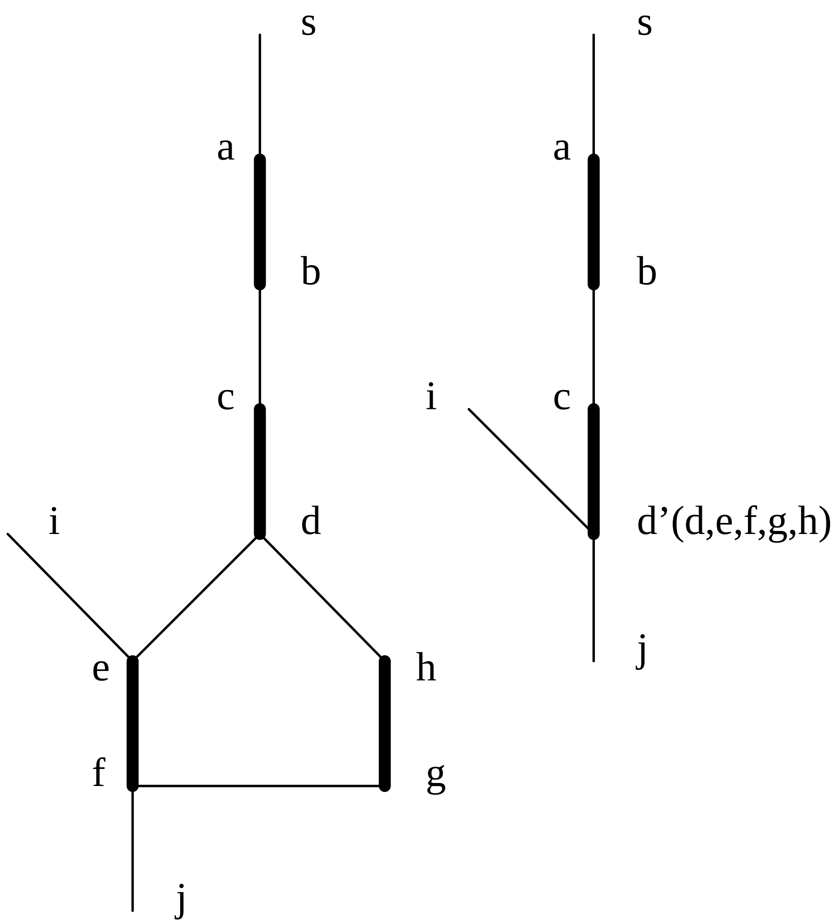


FIGURE XV.41. Shrinking a blossom.
(a) Blossom defined by path from s to d and cycle $[d, e, f, g, h, d]$.
(b) Shrunken blossom. Augmenting path from s to i (resp. j) corresponds to augmenting path in the original graph going around blossom clockwise (resp. counterclockwise).

the base of b. If $\{b, v\}$ is a matched edge in P, v must be adjacent to x, so replace the edge with the matched edge $\{x, v\}$. If $\{b, v\}$ is an unmatched edge in P, then v must be adjacent to some node w in the blossom. There is a path P' in the blossom from x to w, going in either a clockwise or counter-clockwise direction (or no direction, if $w = x$). Replace the unmatched edge $\{b, v\}$ with the path P' from x to w, plus the unmatched edge $\{w, v\}$.

$\Rightarrow$ We will prove this direction by proving the correctness of Edmonds' augmenting path algorithm using blossom shrinking, which is presented below. ■

The algorithm consists of an exploration of the graph from its free vertices along alternating paths; blossoms are shrunken as they are encountered. Each vertex is in one of three states: *unreached*, *odd*, or *even*. (Odd or even vertices are said to be *reached*.) For each node v the algorithm maintains $p(v)$, the parent of v in the forest that is created. Initially every matched vertex is unreached and every free vertex v is even, with $p(v) = \textbf{null}$. The algorithm consists of repeating the *Examine Edge* step below until an augmenting path is found or there is no unexamined edge $\{v, w\}$ with v even.

3. Blossom Shrinking Algorithm

Initialize: $M \leftarrow \emptyset$. Mark free vertices even, other vertices unreached, all edges unexamined.
Examine Edge: Choose an unexamined edge $\{v, w\}$ with v even, mark it examined, and apply the appropriate case below:

- **(a): w is unreached, and thus matched:** Mark w odd and $mate(w)$ even; define $p(w) = v$ and $p(mate(w)) = w$ (i.e., add w and $mate(w)$ to the same tree as v).
- **(b): w is odd:** Do nothing. The parity of w is consistent within the tree.
- **(c): w is even and v and w are in different trees:** Stop; there is an augmenting path from the root of the tree containing v to the root of the tree containing w.
- **(d): w is even and v and w are in the same tree:** Edge $\{v, w\}$ closes a blossom. Let u be the nearest ancestor of v and w. Condense every vertex that is a descendant of u and an ancestor of v or w into a blossom node b; define $p(b) = p(u)$ (and $p(x) = b$ for each vertex x that is condensed into b). Mark the blossom even, and an edge into it unexamined iff one of the corresponding edges in the previous (unshrunken) graph is unexamined.

Note that we regard a blossom b as containing not only the vertices on the cycle forming b, but also all vertices belonging to any blssoms on that cycle; that is, containment is transitive.

THEOREM 35. *The blossom-shrinking algorithm succeeds (stops in case (c))* $\Longleftrightarrow$ *There is an augmenting path in the original graph.*

Proof. If the algorithm succeeds, there is an augmenting path in the shrunken graph, since the algorithm maintains the invariant that all even vertices are at the end of an even-length alternating path starting at a free vertex. This path can be expanded to an augmenting path in the original graph by expanding blossoms in the reverse order of their shrinking, in the manner described in the proof of Theorem 2.

To prove the converse (i.e., to show that if the algorithm doesn't halt in (c) then the original graph has an augmenting path), we first make the following observations:

- **O1:** If the algorithm does not terminate in case (c), all even vertices (including blossoms) that were adjacent at some point during the algorithm are condensed into the same vertex (blossom) at the end of the algorithm.

O2: If v is a matched vertex, it is reached iff mate(v) is; if they are reached, one is marked even, the other odd.

O3: A blossom contains exactly one vertex with no matched edge in the blossom. This is true even for nested blossoms.

To verify the observation **O1**, suppose to the contrary that $\{v, w\}$ is an edge such that v and w are both even. Eventually either v and w will be condensed into a common blossom, or an edge corresponding to $\{v, w\}$ in the shrunken graph will be examined. Such an examination causes v and w to be condensed into a common blossom.

Now suppose the algorithm fails but there is an augmenting path $p = [x_0, x_1, ..., x_{2l+1}]$ in the original graph. We will derive a contradiction in the following manner. Consider the situation after the algorithm halts. We will show that x_i is even (or odd, but shrunken into an (even) blossom) for all even i, $0 \leq i \leq 2l$. Then symmetry states that x_i is even (or shrunken) for all i, $0 \leq i \leq 2l + 1$. Observation **O1** implies that these vertices must be condensed into the same blossom by the end of the algorithm, and since both x_0 and x_{2l+1} are free, **O3** implies a contradiction.

So, we aim to show that x_i is even (or shrunken), for all even i. Suppose not; let i be the least even index such that x_i is neither even nor shrunken. Thus $i > 0$ (x_0 is free, and therefore even) and x_{i-2} is even (or shrunken). The latter implies that x_{i-1} is reached; and by observation **O2**, x_{i-1} must be even, since otherwise x_i would be even. Thus x_{i-2} and x_{i-1} are even (or shrunken). Let j be the smallest index such that x_j,x_{j+1},...,x_{i-1} are even (or shrunken). By observation **O1** all of these vertices are in the same blossom at the end of the algorithm. But this blossom has two vertices with no matched edge in the blossom, namely x_j and x_{i-1}. (We know that j is even, since all x_{2k} $(0 \leq k < i)$ are even or shrunken. So either x_j is free, or its mate is not in the blossom. And the mate of x_i is not shrunken (or even), so it is not in the blossom.) This is not possible, by **O3**, so the algorithm must halt with success if there is an augmenting path in the original graph. ■

This theorem implies the "if" part of the previous one. Let G' be formed from G by shrinking a blossom b. Suppose that we run the algorithm in parallel on G and G'. On G, we can begin by following the path to and around the blossom and shrinking it. On G', we can begin by following the path to b. Now the algorithm is exactly the same on G and G', so it succeeds on G iff it succeeds on G'. But it succeeds on each iff there is an augmenting path in each, so G has an augmenting path iff G' does.

4. Complexity

The augmenting path method, using blossom-shrinking to find augmenting paths, finds a maximum matching in polynomial time. There are $O(n)$ augmentations as previously mentioned, each of which calls the preceding algorithm. The initialization steps take $O(m)$ time. We can determine which case applies in the *Examine Edge* step at a cost of $O(1)$ per edge by labeling vertices with their respective trees. Cases (a) and (b) take $O(1)$ time per edge and thus $O(m)$ time total.

Case (c) can happen only once. We have to reconstruct the original augmenting path when this happens. Clearly any augmenting path can have at most $O(n)$ blossoms, and it takes $O(n^2)$ effort to expand blossoms. To see this, let w and b be adjacent nodes in a shrunken graph. Both w and b could be condensed blossoms, and expanding them requires finding an edge that connects two vertices in the blossoms. This is an $O(n^2)$ operation. Therefore case (c) can take $O(n^3)$ time. For case (d), identifying the vertices in a blossom is an $O(n)$ operation. Since each blossom shrinking reduces the number of vertices, case (d) occurs at most n times, thus taking a total of $O(n^2)$ operations. The algorithm for finding an augmenting path therefore takes $O(n^3)$ total time, and a maximum matching can be found in $O(n^4)$ time.

It is not difficult to implement case (c) to run in $O(n^2)$ time, giving an $O(n^3)$ algorithm. With special data structures for set union, this algorithm can be implemented in $O(nm)$. In addition, there is another algorithm that runs in $O(\sqrt{n}m)$, thereby matching the best known algorithm for bipartite matching.

LECTURE XVI

The Travelling Salesman Problem

Scribe: Amod Agashe

1. NP-Complete Problems

Up to now we have looked at algorithms that run in polynomial time. However, there are important problems which do not have (or are not known to have) polynomial time algorithms. Class NP is an important class of problems that can be solved in polynomial time on a non-deterministic Turing machine. NP-complete problems are the "hardest" problems in NP and there are no known polynomial-time algorithms to solve them.

Since there are a number of interesting NP-complete problems which occur in practice we can use one of three ways of dealing with NP-complete problems (and other hard problems as well):

(1) **Exact Algorithms** — If a problem has a "fast" average case algorithm, "small" input size or if we have a "large" computation time available to us, then we can afford to compute exact solutions by directly applying a non-polynomial algorithm. This is a technique often used when the problem requires an exact solution, *e.g.* some problems in cryptography.

(2) **Approximation Algorithms** — When a problem does not require an exact solution we can often use a polynomial time algorithm that will give us an approximate solution, with some guarantee of how close the approximate solution will be to the exact solution. For many applications, solutions between 1% to 10% of optimal are quite acceptable.

(3) **Heuristic Algorithms** — These are algorithms which give solutions which cannot be shown to be exact or even approximate (recall that approximate solutions come with a guarantee of how close they are to the exact solution). However, in practice,

these algorithms often give solutions very close to the exact solution and are faster than approximation or exact algorithms.

2. Traveling Salesman Problem : Preliminaries

A very good reference for the Traveling Salesman Problem is [29].

The traveling salesman problem (TSP) is one of the most studied of all NP-complete problems. It can be motivated by the problem of a traveling salesman who has a number of clients in different cities and while visiting each of his clients exactly once he wants to minimize his total traveling distance.

This problem has some practical applications. For example, in certain forms of VLSI manufacturing one needs to burn some connections on a chip. The cost of this process is proportional to the time taken by the instrument to move around. We can pose this as a TSP by modelling the connections as vertices and the cost of movement as a length function.

Definition 18. *A Hamiltonian cycle is a cycle in a graph that contains each node of the graph exactly once.*

Definition 19. *The Traveling Salesman Problem is as follows:*
Input: a complete graph $G = (V, E)$ and a non-negative length function l on the edges.
Output: a minimum length hamiltonian cycle.

Definition 20. *If $\ell(u, v) = \ell(v, u)$ for all edges in E, then the graph is said to be symmetric.*

Definition 21. *If $\ell(a, c) \leq \ell(a, b) + \ell(a, c)$ for all edges in E then the triangle inequality holds for the graph.*

In many applications, we can convert any TSP problem to one in which the triangle inequality holds by setting the length between two nodes as the length of the shortest path between the nodes.

Definition 22. *Euclidean TSP: This is a specific case of the general TSP where the nodes are points in Euclidean space and the lengths of the edges are the Euclidean distances. The Euclidean TSP problem is symmetric and satisfies the triangle inequality.*

Definition 23. *A planar TSP is a TSP in R^2.*

In this class we will consider the TSP which is symmetric and in which the triangle inequality holds. Note that all the special cases of the TSP problem (symmetric, asymmetric etc.) are NP-complete. Then why are we considering special cases at all? The reason is

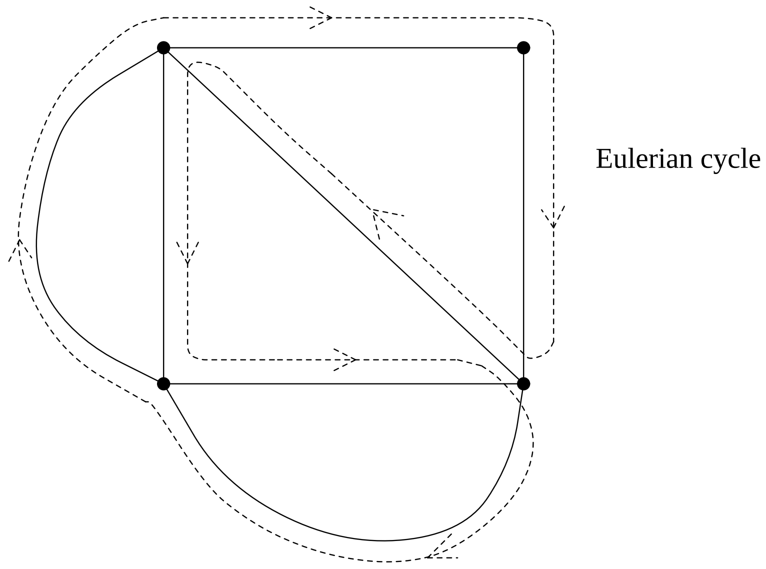


FIGURE XVI.42. Eulerian cycle.

that the standard reductions of one problem to another apply only to the exact solutions. We are interested in approximate solutions and heuristics, and here the different cases can give entirely different results. In this lecture, we will talk about approximation algorithms and heuristics for the TSP. In the next lecture, we will talk about exact solutions.

3. Approximation Algorithms for the TSP

3.1. Preliminaries.

DEFINITION 24. *An algorithm is* <u>*k-approximate*</u> *if its solution has the property that* $X_A \leq kX^*$ *where* X^* *is the length of the optimal tour and* X_A *is the length of the tour given by the approximate solution.*

DEFINITION 25. *A Eulerian cycle is a cycle that contains edge in the graph exactly once. (Note: in this lecture, we will be dealing only with connected graphs, hence a Eulerian cycle also passes through every vertex at least once.) See Figure* XVI.42.

DEFINITION 26. *A graph is said to be Eulerian if it contains a Eulerian cycle.*

THEOREM 36. *G is an Eulerian graph $\Leftrightarrow$ G is connected and every node has even degree.*

Proof. ($\Rightarrow$) We are given that G contains an Eulerian cycle. Consider a node v. Starting at v, go around the Eulerian cycle and eventually ending at v. We can pair up the edges going in and out of v by matching an edge we used to leave v with an edge we used to return to v next time. Note that this works because the cycle visits an edge exactly once. Thus v

FIGURE XVI.43. Base case for Theorem 36.

has even degree. We can repeat the argument for any other vertex. Hence all vertices have even degree.

($\Leftarrow$) By Induction on the number of edges. We are given that the graph is connected and all vertices have even degree.

Base Case: Number of edges = 1. Here we have only two cases as shown in Figure XVI.43.

In case 1, the graph is connected and every vertex has even degree; and it has an Eulerian tour. In case 2, all vertices have odd degree and there is no Eulerian cycle. Therefore our hypothesis is correct for the base case.

Inductive step: Assume the theorem is true for all i, $1 \leq i \leq k-1$.

Case 1: Every node has degree 2. Since G is connected, G must be a cycle, which is the Eulerian cycle we are looking for.

Case 2: There exists a node of degree greater than two. Then there must be a cycle through this node (because all node degrees are even). If we remove this cycle from G and apply the inductive hypothesis to each of the connected components left in the graph, we get Eulerian cycles in the components. (Note that removing a cycle keeps node degrees even.)

We can patch these Eulerian cycles together using the removed cycle as follows. Traverse the removed cycle. The first time you enter a component, say at vertex v, traverse its Eulerian cycle to come back to v, and continue traversing the removed cycle as shown in Figure XVI.44.

Since every edge is either on the removed cycle or in one of the connected components, the resulting cycle contains all the edges in the graph and hence is an Eulerian cycle. ∎

LEMMA 45. *If there is a cycle going through every node and the triangle inequality holds, then there is a Hamiltonian cycle of no greater length.*

Proof. Consider a node v. Suppose it is visited more than once in the given cycle and suppose that when we visit it the second (or any later) time, the sequence of vertices is

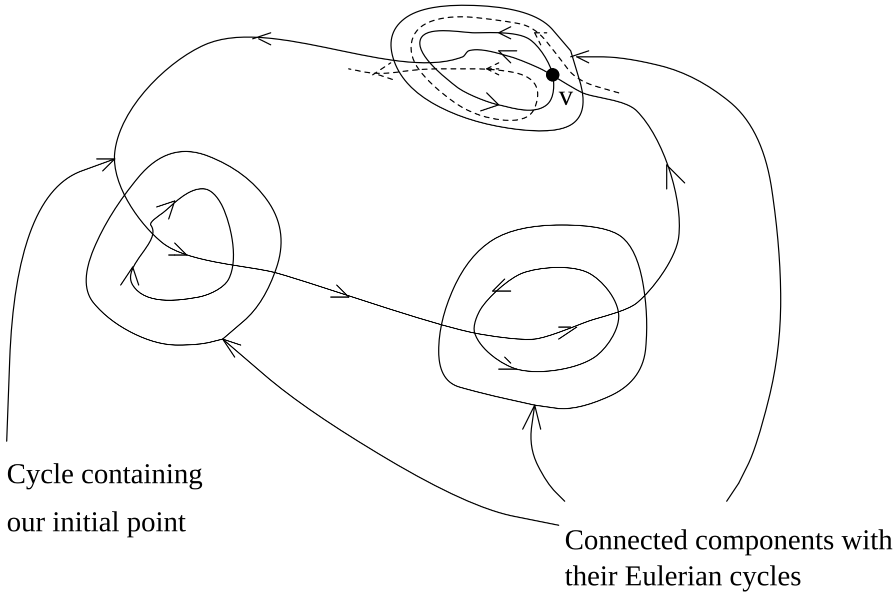


FIGURE XVI.44. Inductive step in Theorem 36

...u,v,w....Then, we can modify the cycle by skipping v, *i.e.*, going directly from u to w as shown in Figure XVI.45.

Note that due to the triangle inequality, we get a cycle with no greater length. We can repeat this procedure till all the vertices are visited only once. We end up with a Hamiltonian cycle of no greater length. Note: we shall henceforth refer to the procedure we just used as "short-cutting the cycle". ■

3.2. 2-Approximation Tree Algorithm.

(a) Find the minimum spanning tree (MST) T under l_{ij}.
(b) Consider the graph $G' = (V, T \cup T)$. It is connected and every vertex has even degree. Therefore, by Lemma 45, it has a Eulerian cycle.
(c) Find an Eulerian cycle in G' and short-cut it to get a Hamiltonian cycle.

THEOREM 37. *Our tree algorithm is a 2-approximation algorithm for Euclidean TSP.*

Proof. Let X^* be the length of the optimal tour and X the length of the tour given by the approximate solution. Let $\ell(T)$ be the length of the MST T of step (a). Then, $\ell(T) \leq X^*$, since the shortest tour can be transformed into a tree simply by erasing an edge, and the shortest spanning tree is at least as short as the result. The length of the Eulerian cycle of step (c) is $2\ell(T)$, and by Lemma 45, since short-cutting gives a Hamiltonian cycle of no greater length, $X \leq 2\ell(T)$. Since $l(T) \leq X^*$, we have $X \leq 2X^*$. ■

Note that we can find a MST in $O(m + n \log n)$ time by using Fibonacci heaps and there

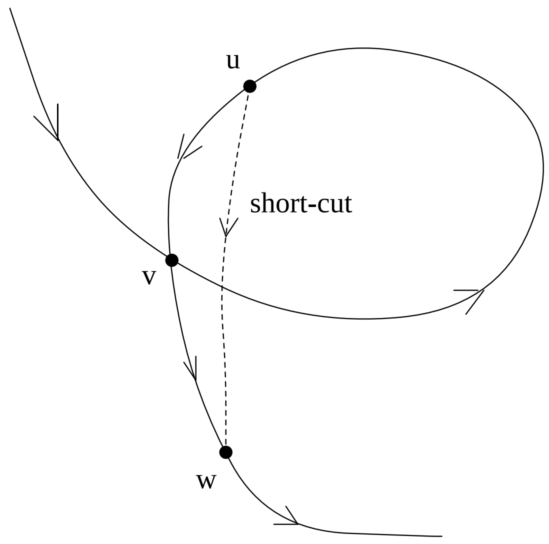


FIGURE XVI.45. Short-cutting.

exists an algorithm to find a Eulerian cycle in $O(m)$ time. Hence the running time of our Tree algorithm is $O(m + n \log n)$.

3.3. Christofides' Algorithm.

(a) Find the MST T.
(b) Find the nodes of T of *odd degree* and find the minimum length perfect matching M in the complete graph consisting of these nodes only. Let $G' = (V, T \cup M)$.
(c) Find a Eulerian cycle of G' and short-cut it to get a Hamiltonian cycle.

Remarks:

In part (b): Note that the number of nodes of T of *odd degree* is even (since the sum of the degrees of the nodes in T is $2n$, *i.e.,* even. In this sum, the sum over those nodes which have even degree is even; hence the sum over nodes with odd degrees is also even. This can only happen if the latter are even in number). Hence we can find a perfect matching.

In part (c): G' is connected (since it has a spanning tree) and every node in it has even degree (since if a node had even degree in T, it has the same degree in G'. If it has an odd degree in T, it has one more edge, coming from the matching M, incident upon it and hence it has even degree in G'). Hence, by Lemma 45, it has a Eulerian cycle.

THEOREM 38. *Christofides' algorithm is a 1.5-approximation algorithm for the Euclidean TSP.*

Proof. From the discussion above, it is clear that the algorithm finds a tour. Recall that the graph G' consists of T and M; hence the cost of the resulting tour τ satisfies $\ell(\tau) \leq \ell(T) + \ell(M)$, the length of the Eulerian tour, by Lemma 45. Also, $\ell(T) \leq \ell(\tau^*)$

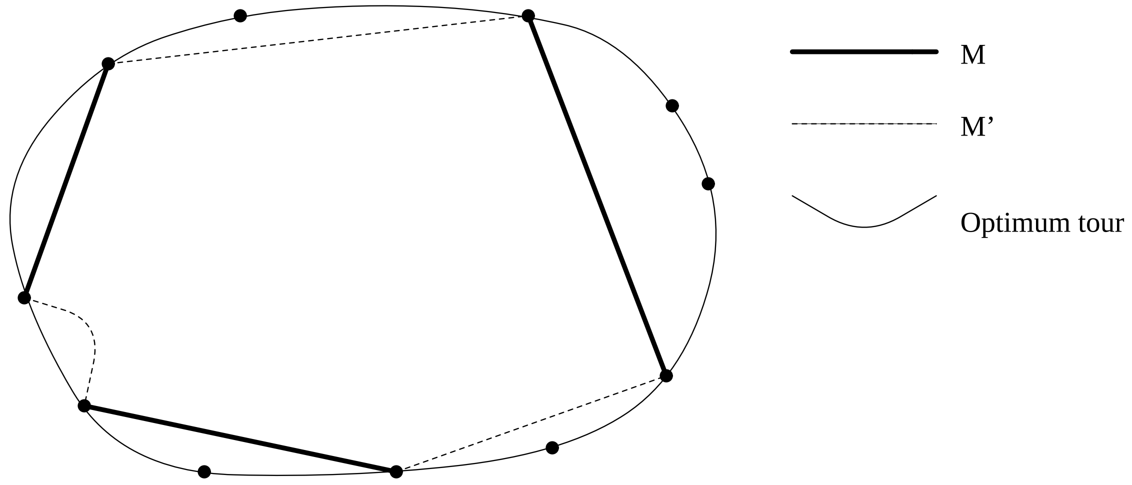


FIGURE XVI.46. Two matchings for Theorem 38

where τ^* is the shortest tour, as discussed in the previous proof. Also let $\{i_1,i_2,...,i_{2p}\}$ be the set of odd-degree nodes in T, in the order that they appear in τ^*. In other words, $\tau^* = [s_0,i_1,s_1 i_2...i_{2p},s_{2p}]$ where $s_0,s_1,...,s_{2p}$ are sequences (possibly empty) of nodes from $\{1,...,n\}$ Consider the two matchings of the odd-degree nodes M_1=$\{[i_1, i_2],[i_3, i_4],...,[i_{2m-1}, i_{2m}]\}$ and M_2= $\{[i_2, i_3],[i_4, i_5],...,[i_{2m}, i_1]\}$ as shown in Figure XVI.46.

By the triangle inequality, $\ell(\tau^*) \geq \ell(M_1) + l(M_2)$. However, M is the optimal matching, so $\ell(\tau^*) \geq 2\ell(M)$, *i.e.,* $\ell(M) \leq 0.5\ell(\tau^*)$. But then, $\ell(\tau) \leq 1.5\ell(\tau^*)$, and we have our result. ■

Christofides' algorithm runs in polynomial time. Step (a) can be done in $O(n^2)$ time. Step (b) can be done in $O(n^4)$ time. Step (c) can be carried out in linear time.

4. Heuristic Algorithms

In the following, we will consider the Euclidean TSP.

(1) Tour Construction Heuristics: the idea here is to start with an empty tour and build it up until it contains all the nodes.

(a) Nearest Neighbor:

- pick an arbitrary vertex u, set tour τ=$\{u\}$
- for i=$1, 2, ...n-1$ do
  : pick a node v in $V-\tau$ *closest* to last node added
  : insert node v into tour τ along with the shortest edge of the previous step

———————————————————————————-

Note: In the following, *insert* v into tour τ means :

- find the edge $\{i, j\}$ on τ which minimizes $\ell(i, v) + \ell(v, j) - \ell(i, j)$
- delete $\{i, j\}$ from τ and add $\{i, v\}$ and $\{v, j\}$ to τ.

———————————————————————————————————-

(b) Nearest Insertion:

- pick an arbitrary vertex u, set tour $\tau = \{u\}$.

- if the current partial tour τ does not include all the vertices, then find those vertices v, w such that w on the tour and v not on the tour, for which $\ell(v, w)$ is minimum (*i.e.,* choose node v that is closest to the set of tour nodes). Insert v into the tour.

(c) Furthest Insertion:

- as above but choose the node v furthest from the tour nodes.

(d) Random Insertion:

- as above but choose the node v randomly.

Generally, FI is better than NN and NI.

(2) Tour Improvement Heuristics :

To be done in the next lecture.

LECTURE XVII

Branch and Bound Algorithms for TSP

Scribe: Eugene Wong

1. Heuristic Algorithms

1.1. Tour Construction Heuristics. Recall from the last time, in *tour construction heuristics* we start with an empty tour and build it until it contains all the nodes. The following are some examples of this type of heuristics:

(1) Nearest Neighbor
(2) Nearest Insertion
(3) Further Insertion
(4) Random Insertion

1.2. Tour Improvement Heuristics. An example of this are the *k-optimal procedures*. These procedures pick k edges on the current tour and look at all possible ways of replacing them with k edges not on the tour. If the best tour obtained in such a way is shorter than the current one, this best tour replaces the current tour. We stop when the current tour cannot be improved any more. If k is a constant, this procedure runs in polynomial time because each subset of k edges needs to be examined once. The degree of the polynomial grows with k, however, so only 2-opt and 3-opt procedures are used in practice.

A tour improvement heuristic can start with a random tour. It also can start with a tour constructed by another heuristic. The possibilities are numerous. For example, we can construct a tour using random insertion, then apply 2-opt, and then 3-opt. We can repeat this 20 times and take the best tour found. See [29] for details and experimental data.

2. Branch And Bound Method

2.1. Integer Programming Formulation of TSP. We can formulate the TSP as an *integer programming problem.*

Let $x_{ij} = \begin{cases} 1 & \text{if arc (i,j) is in the tour,} \\ 0 & \text{otherwise} \end{cases}$

Let c_{ij} be the cost associate with each arc $(i, j) \in E$. The formulation is as follows:

$$\max \sum_{i,j \in V} c_{ij} x_{ij} \tag{15}$$

such that

$$x_{ij} \in \{0, 1\} \quad \forall i \in V \tag{16}$$

$$\sum_j x_{ij} = 1 \quad \forall i \in V \tag{17}$$

$$\sum_i x_{ij} = 1 \quad \forall j \in V \tag{18}$$

$$\sum_{\substack{i \in S \\ j \in \bar{S}}} x_{ij} \geq 1 \quad \forall \quad S \subset V,\ S \neq \emptyset. \tag{19}$$

The constraints say that every node has in and out degrees of one (this forces the set of selected arcs to form a collection of node-disjoint cycles) and that every nontrivial cut is crossed by at least one arc (to assure that there is only one cycle).

2.2. Branching. Let I_k be a subset of *included arcs* and E_k be a subset of *excluded arcs* of the k^{th} subproblem. Every solution to the subproblem (I, E) must include all arcs in I and no arcs from E. We can enumerate all possible assignments of x_{ij}'s as follows.

(1) Pick $e \in E - (I_k \cap E_k)$.
(2) Branch to the problem $I_{k0} = I_k$ and $E_{k0} = E_k \cup \{e\}$.
(3) Branch to the problem $I_{k1} = I_k \cup \{e\}$ and $E_{k1} = E_k$.

During this enumeration, we discover all tours and pick the shortest. The enumeration process can be viewed as searching a tree with nodes labeled by subproblems.

The main idea of the branch and bound method is as follows. Suppose we have a tour of length U and a lower bound L on the solution value for a subproblem (I, E). If $L > U$, we do not have to examine the subtree rooted at the node corresponding to (I, E).

One way to implement branch and bound is as follows.

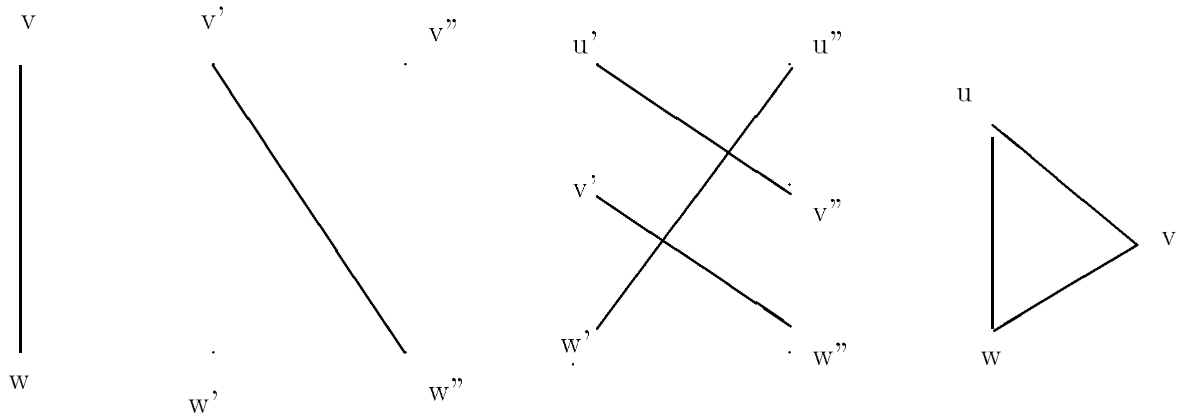


FIGURE XVII.47. The reduction of the node-disjoint cycles problem to the assignment problem.

(1) Initialization
Set $U = \infty$, $S = (\emptyset, \emptyset)$, $T =$ undefined.
(2) Select
If $S = \emptyset$ stop, otherwise remove (I, E) from S.
(3) Bound
Solve relaxation of (I, E) and let L be the resulting lower bound.
If $L \geq U$ goto 2.
If $L < U$ and relaxed solution is a tour, update U and T.
(4) Branch
Add to S pairs $(I_1, E_1), ..., (I_k, E_k)$ corresponding to (I, E).

Note that the most time consuming step is the lower bound computation.

A *selection* rule specifies how the next subproblem is selected from S. Depth-first search is the most commonly used selection rules.

2.3. Assignment Problem Relaxation. One way to relax TSP is to remove constraints (19). This defines the problem of finding a minimum cost collection of node-disjoint cycles covering all nodes. This problem reduces to the assignment problem by splitting every node v into two nodes v' and v'' and replacing every arc (v, w) by an edge $\{v', w''\}$. Figure XVII.47 illustrates this construction. The proof that this reduction works is left as an exercise.

To use the relaxation in the context of the branch and bound method, we need to be able to deal with the included and excluded edges in the assignment problem. For most algorithms, this is easy to do. In particular, the assignment problem algorithm we studied uses a reduction to the minimum-cost circulation problem. During the reduction, we can set upper bounds on capacities of arcs corresponding to the excluded edges to zero and set lower bounds on capacities of arcs corresponding to the included arcs to one.

2.4. Branching Rules. We discuss two branching rules for branch and bound with the assignment problem relaxation. See [29] for more.

Consider a solution to the relaxed problem. Let $(a_1, ..., a_k)$ be a cycle with the smallest number of arcs in the solution. One possible branching rule is to exclude one of the arcs on the cycle.

Rule 1: Replace (I, E) by $\{(I, E \cup \{a_1\}), \ldots, (I, E \cup \{a_k\})\}$.

If this rule is used, the search tree may have several nodes corresponding to the same subproblem. The following rule avoids this problem. The rule includes first $t-1$ arcs on the cycle and excludes the t-th arc.

Rule 2: Replace (I, E) by $\{(I, E \cap \{a_1, \ldots, a_{t-1}\}, E \cap a_t) \ \ for \ \ 1 \leq t \leq k\}$

During a branch and bound computation, it is often possible to use the relaxed solution obtained at a parent node as a starting point of computing the relaxed solution at the current node. This may reduce the running time considerably.

LECTURE XVIII

Held-Karp Lower Bound for TSP

Scribe: James McDonald

This lecture was a continuation of the discussion on the Traveling Salesman Problem. The previous lecture showed how TSP could be solved using branch and bound by relaxing TSP to the assignment problem. This lecture also considered branch and bound, but relaxed TSP to a maximization problem of minimum 1-trees.

1. Problem Motivation

Since TSP is NP-complete, the question arises whether instances of TSP can be relaxed to instances of other problems that are easier to solve, yet whose results are approximately correct solutions to the original TSP problem. The general tradeoff is that we want to relax the problem enough so that we can solve it quickly, but as little as possible, to keep the bound tight.

For unconstrained TSP, we cannot get a good upper bound in polynomial time if $P \neq NP$. In [14], page 147, theorem 6.13 states that upper bounds computed in polynomial time are arbitrarily bad: "If $P \neq NP$, then no polynomial time approximation algorithm A for the traveling salesman problem can have $R_A^\infty < \infty$", where R_A^∞ is the asymptotic best guaranteed ratio of the solution given by the approximation algorithm to the one that is optimal.

However, if we can assume the triangle inequality holds for distances between nodes, then some approximation techniques are possible. For example, a previous lecture showed how we can use minimum spanning trees to get a solution less than or equal to twice the optimal solution, and Christofides' algorithm improves that ratio to 1.5.

This lecture shows how Held and Karp use 1-trees to place a lower bound on solutions to a symmetric TSP with the triangle inequality.

2. Problem Definition

Given a set of nodes and a distance function c relating any two nodes, find a tour of minimum length, where a tour is defined as a cycle that includes every node. Assume that the distance function obeys the triangle inequality, *i.e.*, $c(x, y) + c(y, z) \geq c(x, z)$. Further, assume the distance function is symmetric, *i.e.*, $c(x, y) = c(y, x)$.

For example, the triangle inequality will be true if we are considering nodes given coordinates in a Euclidean space, and we use the normal Euclidean distance function.

3. Equivalence of TSP to an Integer Linear Program

TSP and IP (integer programming) are each NP-complete (hence can be reduced to each other), so as in the previous lecture, we recast TSP as an integer programming problem:

(1)	$\min \sum_{i\in V, j\in V} c_{ij}x_{ij}$	c_{ij} represents the cost of an arc	
	given the constraints:		
(2)	$x_{ij} \in \{0, 1\}$	$\forall i, j \in V$	
(3)	$\sum_{j\in V} x_{ij} = 2$	$\forall i \in V$	Every node has degree 2.
(4)	$\sum_{i\in S, j\in \bar{S}} x_{ij} \geq 2$	$S \subset V, S \neq 0$	The graph is two-connected.
	This last condition (connectivity) is equivalent to		
(5)	$\sum_{i\in S, j\in S} x_{ij} \leq 2(\lvert S\rvert - 1)$	$S \subset V, S \neq 0$	No subtours.

4. Branch and Bound

As explained in the previous lecture, a *branch and bound* procedure for a discrete optimization problem (*e.g.* TSP) works by breaking the feasible set of solutions into successively smaller subsets, calculating bounds on the value of an objective function over each subset, and using those bounds to eliminate subsets.

In [29], Balas and Toth list four essential ingredients for any branch and bound procedure for a discrete optimization problem P of the form $min\{f(x)|x \in T\}$, where the first requirement is by far the most important:

(1) A relaxation problem R for P, of the form $min\{f(x)|x \in S\}$, such that $S \subset T$.
(2) A branching rule that partitions a feasible subset S_i into smaller subsets.
(3) A lower bounding procedure that bounds from below the optimal solutions to each R_i that is the relaxation of P restricted to S_i.
(4) A selection rule for choosing the next subproblem to be processed (*e.g.* depth first search).

5. Relaxation of TSP to Lagrangian dual

5.1. 1-Trees. Given a distinguished node $t \in V$, a *1-tree* (also called a unicycle) is a spanning tree of $V - \{t\}$ plus a pair of arcs connecting t to the tree. We define the *cost* of a 1-tree to be the sum of costs of its edges. A *minimum* 1-tree is a 1-tree of the minimum weight. Note that if a minimum 1-tree is a tour, it is an optimal solution to TSP, although the converse is not true. Without loss of generality, we assume from now on that the special node t is node 1.

A minimum 1-tree can be computed as follows.

ALGORITHM 2. Finding minimum 1-tree.

(1) Find an MST of $G - \{1\}$.
(2) Add the two cheapest edges connecting node 1 to that MST.

The result is a minimum 1-tree (with respect to node 1) containing an MST with node 1 at a leaf. It is obvious that this algorithm is correct. The running time of the algorithm is dominated by the first step, which takes $O(m + n \log n)$ time.

However, the cost of such a 1-tree is not a particularly good lower bound for TSP. In [29], Balas and Toth report that on a set of 140 problems with $40 \leq \text{n} \leq 100$ the costs of minimum 1-trees were only about 63% of the values for optimal TSP solutions.

5.2. Penalty Function to Introduce Bias Towards Tours. We can improve this lower bound by noticing that if we introduce a cost for each node to be added to each adjacent edge, we do not alter the ranking of tours (they all change by twice the total cost added for all nodes), but the ranking of 1-trees may change. In particular, if a larger penalty is added for nodes of high degree, the search for minimal 1-trees will be biased towards tours.

With this in mind, we revise the constraints given above as follows:

(1) $\min \sum_{i \in V, j \in V} c_{ij} x_{ij}$
(2) $x_{ij} \in \{0, 1\}$ $\quad \forall i, j \in V$
(3) $\sum_{i \in S, j \in S} x_{ij} \geq 1$ $\quad S \subset V, S \neq 0$ $\quad$ Connectivity.
This is a relaxation of the requirement for two-connectivity.
(4) $\sum_{i \in V, j \in V} x_{ij} = 2 * n$ $\quad$ The average degree is 2.
This is a relaxation of the previous constraints on degrees ($\sum_j x_{ij} = 2$)
(5) $\sum_{j \in V} x_{1,j} = 2$ $\quad$ The degree of the distinguished node 1 is 2.

Given these constraints, we could have nodes with degree 1 and other nodes with degree exceeding 2. Call the latter bad nodes, and introduce a penalty function λ and a reduced cost function $\bar{c}$:

$\lambda : V \rightarrow \Re$

$\bar{c}_{ij} = c_{ij} + \lambda_i + \lambda_j$.

Note that minimizing $\sum \bar{c}_{ij}$ is equivalent to minimizing $\sum c_{ij}x_{ij} + 2\sum_{i\in V}\lambda_i$.

LEMMA 46. *If C^* is a minimum weight tour with respect to $\bar{c}_{ij}$, then C^* is a minimum weight tour with respect to c_{ij}.*

Proof. The proof is obvious, as indicated above: since every node is accounted for twice in every tour, once in the edge entering it and one in the edge leaving it, the weight of every tour is increased by the same amount. ∎

However, even though the rank of a tour is not changed, the rank of a 1-tree can change, since nodes are not all of degree 2. Consider the example in Figure XVIII.48.

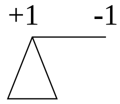


FIGURE XVIII.48. Penalty function affects weight of a 1-Tree.

6. Revision of Relaxed Problem on 1-Trees to Use Penalties

Now we revise the problem so that we are looking for a penalty function λ that produces the maximal weights for minimal 1-trees. This should yield an improved lower bound.

Let L(λ) be the weight of minimum 1-tree using $\bar{c}$ cost function induced by λ.

Let L $= max_\lambda L(\lambda)$

L is called the Lagrangian dual of TSP, and gives a much tighter lower bound than the simple 1-tree problem with the TSP cost function, but it is harder to compute. The fastest known algorithm for the problem, due to Vaidya [45], uses interior-point techniques and runs in $O(n^2 M(n)\log n)$ time, where $M(n)$ is the time needed to multiply two $n \times n$ matrices. This problem can be also solved using the ellipsoid method and using a generalization of the approximate multicommodity flow algorithm discussed earlier to packing and covering problems [36].

6.1. Subgradient Optimization. In [31], Held and Karp considered several methods to maximize L, including a straightforward ascent method, which started the area of subgradient optimization.

Let $d(n)$ be the degree of n in a minimum 1-tree. We want to make nodes with $d(v) > 2$ more expensive, and the node with $d(v) < 1$ less expensive. At each iteration, update λ as follows:

$$\lambda^{k+1} \leftarrow \lambda^k + \sigma^k(d_i^k - 2)$$

where d_i^k are determined by the maximum 1-tree found using λ^k.

In [29], Balas and Toth define σ^k as $\frac{\alpha(U-L(\lambda^k))}{\sum_{i\in V}(d_i^k-2)^2}$. If σ is small enough, this procedure will converge to an optimal value for λ. In practice, σ is often set to a constant, and an the algorithm is stopped when the improvement at each step becomes small. The resulting solution is usually a very good approximation to the minimum 1-tree.

In [29], Balas and Toth also define a subgradient of a convex function $f(x)$ at $x = x^k$ as a vector s such that $f(x) - f(x^k) \geq s(x - x^k)$ for all x. The n-vector whose components are $d_i^k - 2$ is a subgradient of L(λ) at λ^k, hence the name subgradient optimization.

6.2. Held-Karp lower bound is not strict. The lower bound achieved by the Held-Karp method is not arbitrarily close to the optimal TSP solution, as Figure XVIII.49 illustrates:

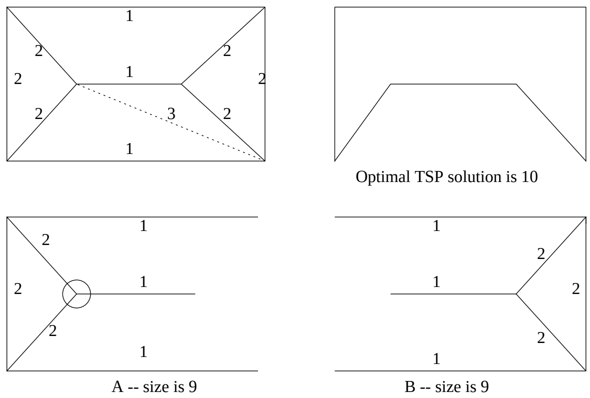


FIGURE XVIII.49. Held-Karp lower bound is sometimes suboptimal.

The solution to TSP is 10, which can be found by enumeration, but the Held-Karp lower bound is 9, as seen in the two 1-trees A and B. The problem here is that some nodes are of degree 1 in A iff they are of degree 3 in B, and vice versa. This can be used to show that in the Held-Karp lower bound for this problem is 2.

This example does not specify the distinguished node. To fix this problem, we introduce a node in the middle of the central edge of the example, and set the length for the two new edges to 0.5. The distinguished node is the new node, which has degree 2 in both trees.

6.3. A Branching Rule. Many branch-and-bound rules can be used with the Held-Karp lower bound. For example, we can choose an edge $\{i, j\}$ in a 1-tree found by the lower bound procedure, such that $\{i, j\} \notin I$ and $\lambda_i + \lambda_j$ is maximized. Then branch on the inclusion/exclusion of $\{i, j\}$:

$$(E \cup \{\{i, j\}\}, I) \quad (E, I \cup \{\{i, j\}\}).$$

Note that for the current 1-tree is an optimal solution to the second subproblem (but this subproblem has a bigger set of included edges).

7. Misc

Shmoys and Williamson [40] show that the Christofides approximation is actually less than or equal to 1.5 times the Held-Karp lower bound, as depicted in Figure XVIII.50.

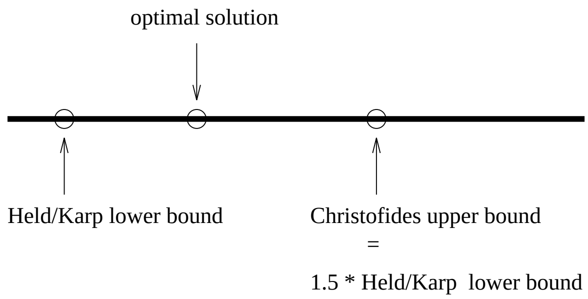


FIGURE XVIII.50. Christofides upper bound = 1.5 × Held-Karp lower bound

LECTURE XIX

Interior Point Method for Linear Programming

Scribe: Waqar Hasan

1. Overview

The Simplex Method is not known to run in polynomial-time. The first polynomial-time algorithm for LP was published by the Soviet mathematician, L.G. Khachian [26]. While an important theoretical contribution, Kachian's ellipsoid algorithm is not very practical.

Karmarkar [24] introduced a practical polynomial method for solving LP. We shall study Gonzaga's [21] affine scaling algorithm. Both Karmarkar's algorithm and Gonzaga's algorithm are interior point methods. Gonzaga's algorithm directly minimizes the potential function that was used by Karmarkar in the analysis of his algorithm.

Whether there exists a strongly polynomial algorithm for solving LP is still an open question.

The Linear Programming Problem.: The Linear Programming problem may be stated as:

$$\begin{aligned} min\, \overline{c}^t x & \\ s.t.\, \overline{A}x &= b \\ x &\geq 0. \end{aligned}$$

where
$\overline{A}$ is a $m \times n$ matrix,
x, c are vectors of length n,
b is a vector of length m.

k:=0
repeat
  scaling:
    $A := \overline{A}D^{-1}, c := \overline{c}D^{-1}, y := Dx$
    $f(y) \ = \ q\log(c^t y) - \sum_{i=1}^{n}\log(y_i)$
  projection
    $h \ := \ P(-\nabla f(e))$
  step
    $y^* := e + \lambda h, where\ \lambda = \frac{0.3}{\|h\|}$
  goback
    $x^{k+1} := D^{-1}y^*$
    $k := k+1$
until $\overline{c}^t x^k \ \leq \varepsilon \sim 2^{-L}$ where L is length of input.

FIGURE XIX.51. The Affine Scaling Algorithm.

2. Gonzaga's Algorithm

We shall make the following assumptions. These assumptions are quite technical, and their main purpose is to simplify the presentation. We will justify some of these assumptions later.

(1) The input problem is bounded.
(2) $\overline{A}$ has full rank.
(3) The optimal value of the objective function is 0.
(4) An initial interior point, x^0, is available.
(5) $f(x^0) = O(L)$, where L is the bit length of the input and f is the potential function defined below.

We note that the second assumption implies that $(\overline{A}\,\overline{A}^t)^{-1}$exists. We begin by looking at a potential function, f, given by

$$f(y) \ = \ q\log(\overline{c}^t y) - \sum_{i=1}^{n}\log(y_i).$$

where $q \ = \ n + \sqrt{n}$ is a weighting factor. We note that the first term goes to $-\infty$ as $\overline{c}^t y \to 0$, while the second terms inflicts an increasing penalty as we move towards the boundary formed by the nonnegativity constraints. We compute the gradient of f to be

$$\nabla f(y) \ = \ \frac{q}{\overline{c}y}\overline{c}^t - (\frac{1}{y_1}, \ldots, \frac{1}{y_n})^t.$$

Upon evaluating the gradient at the point $e \ = \ (1, \ldots, 1)^t$ we have

$$\nabla f(e) \ = \ \frac{q}{\overline{c}^t e}\overline{c}^t - e^t.$$

FIGURE XIX.52. Diagonal Matrix

The algorithm generates a sequence of points $\{x^k\}$ such that $f(x^k) \to 0$. On the k-th iteration let

$D = diag(\frac{1}{x_1^k}, \dots, \frac{1}{x_n^k})$ be the diagonal matrix (see Figure XIX.52).

$P = I - A^t(AA^t)^{-1}A$, with A as given below.

We note that for any vector x, Px is in $N(A)$, the null space of A, since $APx = Ax - AA^t(AA^t)^{-1}Ax = Ax - Ax = 0$. Also we have that $P(Py) = (I - A^t(AA^t)^{-1}A(I - A^t(AA^t)^{-1}Ay = y - 2A^t(AA^t)^{-1}Ay + A^t(AA^t)^{-1}AA^t(AA^t)^{-1}Ay = y - A^t(AA^t)^{-1}Ay = Py$. Also, if $x \in N(A)$, we have $Px = x$. P is known as the projection matrix onto the null-space of A.

The algorithm is described in Figure XIX.51.

We will now discuss the steps of the algorithm in more detail.

- The *scaling* accomplishes an affine transformation of the problem such that the point x^k is mapped into the point e in the new coordinate system by the mapping $y \longleftarrow Dx$, thus *centering* the current interior point. The idea is that the interior point may have been very near the boundary formed by the nonnegativity constraints, but is now centered and a larger step is possible in the new coordinates. The equation on the second line of the scaling step expresses the function f in these new coordinates.
- The *projection* step of the algorithm projects onto the null space of A the vector $-\nabla f(e)$, which, being the negative of the gradient in the new coordinates, is the direction of maximum decrease of the potential function. This is done to ensure that the next step provides us with a valid update.
- The *step* step produces the vector y^* by adding a multiple of the projected vector from the previous step. Since the vector which is added is in the null space of A, we are assured that $Ay^* = Ae = b$. So far, we have transformed the problem into new coordinates, and moved in the direction of the projected direction of greatest decrease.
- In *goback*, we transform the variables back to the original coordinate system.

The algorithm improves the potential function, f, in the transformed space rather than in the original coordinates. It turns out that the improvement in the potential function is actually preserved. In other words a decrease of say 105 in one space is a decrease of 105 in the other. To see this, we observe that, for a given iteration,

$$\begin{aligned} f(y) &= q\log(c^t y) - \sum_{n=1}^{n} \log(y_i) \\ &= q\log(\overline{c}^t x) - \sum_{n=1}^{n} \log(d_i x_i) \\ &= q\log(\overline{c}^t x) - \sum_{n=1}^{n} \log(x_i) + \sum_{n=1}^{n} \log(d_i) \\ &= f(x) + constant. \end{aligned}$$

We denote the first term of $f(y)$ as $f_1(y)$ and the second as $f_2(y)$. That is, we have $f_1(y) = q\log(\overline{c}^t y)$, $f_2(y) = \sum_{i=1}^{n} \log(y_i)$.

We will now obtain an approximation of the potential function. Using Taylor's theorem we have

$$\log(1+\delta) = \delta - \frac{\delta^2}{2(1+\theta)^2}.$$

where θ is between 0 and δ, allowing the possibility that δ is negative. So we have

$$\log(1+\delta) \geq \delta - \frac{\delta^2}{2(1-|\delta|)^2}$$

and hence

$$\log(1+\delta) \geq \delta - 2\delta^2$$

if $1 - |\delta| \geq \frac{1}{2}$.

We will now derive bounds for the terms of the new potential function.

Since $f_1(y) = q\log(\overline{c}^t y)$ is a concave function, a linear approximation provides an overestimation. This implies

$$f_1(e+\lambda h) \leq f_1(e) + \lambda \nabla f_1(e)^t h$$

.

Using the previous result, for an n-vector h, and a number λ such that $1 - |\lambda h_i| \geq \frac{1}{2}$ for each component of λh, we have

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

where $\|h\| = \sqrt{h_1^2 + \ldots + h_n^2}$ and we have used the facts that $f_2(e) = 0$ and $\nabla f_2(e) = e$. Combining, the bounds on f_1 and f_2, we get

$$f(e+\lambda h) \leq f(e) + \lambda \nabla f(e)^t h + 2\lambda^2 \|h\|^2.$$

LECTURE XX

Analysis of Gonzaga's Algorithm

Scribe: David Karger

Recall the last thing we proved last time. We had chosen to change our candidate solution by the quantity $h = -P\nabla f(e)$. In other words, we found the direction of the maximum decrease of the potential function f, selected a vector which in that direction, and then projected that vector into the null-space of A (using the projection matrix P) so that moving along the projection would not cause us to violate the constraint matrix. We then proceeded to prove that

$$f(e+\lambda h) - f(e) \le \lambda \nabla f(e)^t h + 2\lambda^2 \|h\|^2. \tag{20}$$

This required on the assumption that $\lambda h_i < 1/2$ for every i. Since h has been constructed to point in the opposite direction from $\nabla f(e)$, we expect the first term on the right to be rather negative. On the other hand the second term, which has a λ^2 coefficient as opposed to the λ coefficient in the first term, should be too small to cancel out the negativity of the first term. This should let us claim that the potential decreases by a significant amount. We proceed to do this.

Lemma 47. $\|h\| \ge 1$.

Proof. Let $\widehat{y}$ be an optimal solution, so that $c^t\widehat{y} = 0$. Recall that $\widehat{y} - e \in$ null-space of A, since $A\widehat{y} = Ae = b$. Thus $P^t(\widehat{y} - e) = (\widehat{y} - e)$

$$
\begin{aligned}
h^t(\widehat{y}-e) &= -(P\nabla f(e))^t(\widehat{y}-e)\\
&= -(\nabla f(e))^t P^t(\widehat{y}-e)\\
&= -(\nabla f(e))^t(\widehat{y}-e)\\
&= (-\frac{qc}{c^t e}c+e)^t(\widehat{y}-e)\\
&= -\frac{qc^t}{c^t e}\widehat{y}-e^t e+e^t\widehat{y}+\frac{qc^t e}{c^t e}\\
&= \sqrt{n}+e^t\widehat{y},
\end{aligned}
$$

where we have used the facts that $c^t\widehat{y} = 0$ and $q = \sqrt{n}+n$.

Now, since $y_i \geq 0$, we have

$$e^t y = \sum_{i=1}^{n} y_i \;\geq\; \sqrt{\sum_{i=1}^{n} y_i^2}$$

and thus

$$h^t(\widehat{y}-e) \;\geq\; \sqrt{n}+\|\widehat{y}\|,$$

and an application of the Cauchy-Shwartz inequality implies

$$\|h\|\,\|\widehat{y}-e\| \;\geq\; \sqrt{n}+\|\widehat{y}\|$$

which implies

$$\|h\|\,(\|\widehat{y}\|+\|e\|) \;\geq\; \sqrt{n}+\|\widehat{y}\|$$

and on dividing both sides by $\sqrt{n}+\|\widehat{y}\|$ we arrive at

$$\|h\| \;\geq\; 1.$$

■

This lets us show an improvement in the value of f. Recall that we want to drive f down towards $-\infty$. The following result gives an idea of how fast this happens:

LEMMA 48. *$f(y^*) - f(e) = f(e+\lambda h) - f(e) \leq -0.12$ for $\lambda = \frac{0.3}{\|h\|}$.*

Proof. Clearly we have $\max_i |\lambda h_i| \leq 0.5$ so that (1) applies. *i.e.,*

$$f(y^*) - f(e) \;\leq\; \lambda(\nabla f(e))^t h + 2\lambda^2\|h\|^2.$$

Consider the first right-hand term:

$$
\begin{aligned}
(\nabla f(e))^t h &= -(\nabla f(e))^t \nabla f(e) \\
&= -(P\nabla f(e))^t P\nabla f(e) - ((I-P)\nabla f(e))^t P\nabla f(e) \\
&= -(P\nabla f(e))^t P\nabla f(e) \\
&= -\|h\|^2.
\end{aligned}
$$

The third equality follows from the easily verified result that $P(I-P) = 0$, where I and 0 are the identity and zero matrix respectively.

So we have

$$
\begin{aligned}
f(y^*) - f(e) &\leq -\lambda\|h\|^2(1-2\lambda) \\
&\leq -\lambda\|h\|(1-2\lambda) \\
&= -\lambda\|h\|(1-\frac{.6}{\|h\|}) \\
&= \frac{-0.3}{\|h\|}\|h\|(1-0.6) \\
&= -0.12
\end{aligned}
$$

where we have used the fact that $\|h\| \geq 1$ to obtain the second inequality. ■

We now assume that the problem is bounded, the feasible region is bounded and that $f(x^{(0)}) = O(L)$, where L is the length of the input.

LEMMA 49. *The number of iterations needed to get within distance ε of an optimum vertex is $O(nL)$ if $\varepsilon = 2^{-L}$.*

Proof. Let k denote the number of iterations performed until the stopping criterion is met, and let $x^{(i)}$ denote the value of x at the i^{th} iteration. We have $\sum_{i=1}^n \log x_i^{(k)} \leq M$ for some constant M, due to the boundedness of the feasible region. Let $\alpha = 0.12$. Then by the previous analysis of the decrease in f,

$$f(x^{(k)}) \leq f(x^{(0)}) - k\alpha = O(L) - k\alpha \tag{21}$$

and by definition of f and based on the stopping rule,

$$f(x^{(k)}) = q \log \overline{c}^t x^{(k)} - \sum \log x_i^{(k)} \geq -qL - L. \tag{22}$$

Putting these last two statements together results in

$$k\alpha \leq O(L) + qL \tag{23}$$

and hence, recalling the definition of q,

$$k \leq \frac{O(L)}{\alpha} + \frac{qL}{\alpha} = O(nL) \tag{24}$$

■

Once the objective value is less than ε, we will apply a procedure which finds a vertex where the objective function value is at least as good as that of the current solution. If ε is small enough, this vertex is optimal because the limited precision of the input numbers prevents a non-optimal vertex from having a value this close to optimal.

This motivates the following problem. Given a feasible point x, find a vertex v such that $\overline{c}(v) \leq \overline{c}(x)$. This can be done as follows. Consider the direction $e_n = (0, \ldots, 0, 1)$. Set t to be $+e_n$ or $-e_n$ so that if we move from x along t, the objective function is nondecreasing (the objective function, being linear, cannot decrease in both of these directions). Move along t until you hit the first constraint and obtain a feasible point with objective value at most $\overline{c}(x)$. Project the problem in the constraint to obtain a smaller-dimensional problem and continue.

Lemma 50. *Assume all input numbers are B-bit integers. If* $\varepsilon = 2^{-2nL}$, *then if* $\overline{x}$ *is a vertex with* $\overline{c}^t\overline{x} < \varepsilon \rightarrow \overline{c}^t\overline{x} = 0$.

Proof. For the vertex $\overline{x}$ we have that $\overline{A}\overline{x} = b$. Hence there is a invertible submatrix $\overline{A}'$ of $\overline{A}$ and a subvector b' of b such that $\overline{A}'\overline{x'} = b'$. By Cramer's rule, a non-zero component is given by a ratio, the numerator of which is an element of b', and hence is integer, and whose denominator is a properly chosen determinant. We note that the size of this determinant is at most $n^n(2^B)^n$. Suppose $\overline{c}^t\overline{x} > 0$. Since the common denominator of the components of $\overline{c}$ is at most $n^{n^2}2^{n^2B} \leq 2^{2nL}$ for n large enough, and the lemma flows. ■

By a more careful analysis, it can be shown that it is enough to set $\varepsilon = 2^{-L}$.

In the general case that the minimum of the objective function is not 0, we replace the original problem with the problem of minimizing the gap between the primal objective function and the dual objective function, with constraints given by combining the constraints of the primal and dual. If a solution exists, duality theory assures us that the optimal solution of the constructed problem corresponds to the optimal solution of the original problem.

To obtain the initial basic feasible vector we may apply the algorithm to the Phase I version of the problem as is typically done in the simplex method when seeking a basic feasible vector.

Bibliography

[1] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding Minimum-Cost Flows by Double Scaling. *Math. Prog.*, 53:243–266, 1992.

[2] R. G. Bland and D. L. Jensen. On the Computational Behavior of a Polynomial-Time Network Flow Algorithm. *Math. Prog.*, 54:1–41, 1992.

[3] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Netwrok Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] D. D. Sleator and R. E. Tarjan. Self-adjusting Binary Search Trees. *Journal of the ACM*, 32:652–686, 1985.

[6] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[7] J. Edmonds. Paths, Trees and Flowers. *Canada J. Math.*, 17:449–467, 1965.

[8] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19:248–264, 1972.

[9] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. on COmp.*, 4:507–518, 1975.

[10] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[11] L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow through a Network. *Canad. J. Math.*, 8:399–404, 1956.

[12] L. R. Ford, Jr. and D. R. Fulkerson. A Simple Algorithm for Finding Maximal Network Flows and an Application to the Hitchcock Problem. *Canad. J. Math.*, 9:210–218, 1957.

[13] D. Gale and L. S. Shapley. College Admissions and the Stability of Marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[14] M. R Garey and D. S. Johnson. *Computers and Intractability, A guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[15] A. V. Goldberg. A Natural Randomization Strategy for Multicommodity Flow and Related Algorithms. *Info. Proc. Let.*, 42:249–256, 1992.

[16] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.

[17] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. In *Proc. 18th ACM Symp. on Theory of Comp.*, pages 136–146, 1986.

[18] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *Journal of the ACM*, 35:921–940, 1988.

[19] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. *JACM*, 36:873–886, 1989.

[20] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990.

[21] C. C. Gonzaga. An Algorithm for Solving Linear Programming in $O(n^3L)$ Operations. In N. Megiddo, editor, *Progress in Mathematical Programming*, pages 1–28. Springer Verlag, Berlin, 1989.

[22] D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, 1989.

[23] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut of a Graph. In *Proc. 3rd ACM-SIAM Symp. on Discr. Alg.*, pages 165–174, 1992.

[24] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4:373–395, 1984.

[25] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Mathematics*, 23:309–311, 1978.

[26] L. G. Khachian. Polynomial Algorithms in Linear Programming. *Zhurnal Vychislitelnoi Matematiki i Matematicheskoi Fiziki*, 20:53–72, 1980.

[27] D.C. Kozen. *The Design And Analysis of Algorithms*. Springer-Verlag, 1992.

[28] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[29] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan, and D. B. Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[30] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast Approximation Algorithms for Multicommodity Flow Problems. In *23rd ACM Symp. on Theory of Comp.*, 1991.

[31] M. Held and R. M. Karp. The Traveling Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18:1138–1162, 1970.

[32] E. W. Mayr and A. Subramanian. The Complexity of Circuit Value and Network Stability. In *Proceedings of the Fourth Annual Conference on Structure in Complexity Theory*, 1989.

[33] H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Disc. Math.*, 5:54–66, 1992.

[34] J. B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In *Proc. 20th ACM Symp. on Theory of Comp.*, pages 377–387, 1988.

[35] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[36] S. Plotkin, D. Shmoys, and E. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering Problems. Technical Report STAN-CS-92-1419, Department of Computer Science, Stanford University, 1992.

[37] T. Radzik and A. V. Goldberg. Tight Bounds on the Number of Minimum-Mean Cycle Cancellations. In *Proc. 2nd ACM-SIAM Symp. on Discr. Alg.*, pages 110–119, 1991.

[38] H. Röck. Scaling Techniques for Minimal Cost Network Flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191. Carl Hansen, Münich, 1980.

[39] F. Shahrokhi and D. Matula. The Maximum Concurrent Flow Problem. *Jour. ACM*, 37:318–334, 1990.

[40] D. B. Shmoys and D. P. Williamson. Analyzing the Held-Karp TSP Bound: a Monotonicity Property with Application. *Inf. Proc. Let.*, 35:281–285, 1991.

[41] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[42] A. Subramanian. A New Approach to Stable Matching Problem. Technical Report STAN-CS-89-1275, Stanford University, August 1989.

[43] É. Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, 5(3):247–255, 1985.

[44] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[45] P. M. Vaidya. A New Algorithm for Minimizing Convex Functions Over Convex Sets. In *Proc. of the 30th Annual IEEE Symp. on Found. of Comp.Sci.*, pages 338–343, 1989.

[46] Е. А. Диниц. Метод Поразрядного Сокращения Невязок и Транспортные Задачи. Сб. *Исследования по Дискретной Математике*. Наука, Москва, 1973. English transcription: E. A. Dinic, "Metod Porazryadnogo Sokrashcheniya Nevyazok i Transportnye Zadachi," *Issledovaniya po Diskretnoi Matematike*, Science, Moscow. Title translation: The Method of Scaling and Transportation Problems.

[47] Е. А. Диниц. Метод Поразрядного Сокращения Невязок и Транспортные Задачи. Сб. *Исследования по Дискретной Математике*. Наука, Москва, 1973. English transcription: E. A. Dinitz, "Metod Porazryadnogo Sokrashcheniya Nevyazok i Transportnye Zadachi," *Issledovaniya po Diskretnoi Matematike*, Science, Moscow. Title translation: The Method of Scaling and Transportation Problems.

[48] А. В. Карзанов. Оценка Алгоритма Нахождения Максимального Потока, Примененного к Задаче о Представителях. Сб. *Вопросы Кибернетики. Труды Семинара по Комбинаторной Математике. (Москва 1971)*. Советское Радио, Москва, 1973. English transcription: A. V. Karzanov, "Otsenka Algoritma Nokhozhdeniya Maksimal'nogo Potoka, Primenennogo k Zadache o Predstavitelyakh," *Voprosy Kibernetiki, Trudy Seminara po Kombinatornoi Matematike*, Soviet Radio, Moscow. Title translation: A Bound on a Maximum Flow Algorithm Applied to the Problem of Distinct Representatives.