CS 6280: Fall 2008
Algorithm Design Review

Guozhang Wang

September 25, 2010

1 Graph Algorithms

1.1 Depth-First Search

In this section we present DFS and one of its applications: finding Strongly Connected Components of a directed graph. One note here is that finding connected components of an undirected graph can be done by simply applying DFS or BFS.

Problem 1.1 *How to explore a graph G?*

First Attempt:
For $\forall$ u, v $\in$ V(G), run DFS to see if $\exists$ path u $\longrightarrow$ v.
Running time: $O((m+n)^2)$

First Optimization:
For $\forall$ u $\in$ V(G), run DFS to find all vertices reachable from u.
Running time: $O((m \cdot n)$

Further Optimization:
Find a sink-SCC (out-degree 0), explore it, remove it from the graph, and so on.
Running time: $O((m+n)$ if the SCC can be found in linear time.

Problem 1.2 *How to find a sink-Strongly Connected Component?*

Lemma 1.1 *Let C and C' be two strongly connected components of a graph, and suppose that there is an edge from a node of C to a node of C'. Then the node of C visited first by depth-first search has higher* **post** *number than any node of C'.*

Given this lemma, we can design our algorithm as follows:

Algorithm 1.1 *Strongly Connected Component*

Step 1*: Perform depth-first search on the reverse graph G^R of G.*
Step 2*: Run the undirected connected components algorithm using DFS on G in order of decreasing* **post** *found in step 1.*

This algorithm is as linear-time as DFS.

1.2 Maximum Matching

In this section we show how to find maximum matching using Greedy, and show how to prove "Greedy Stays Ahead" and "Greedy Never Stuck".

Problem 1.3 *How to find the Max-Weight Bipartite Matching?*

The Solution: Find M_augmenting path P from a M matching and then update $M = M \oplus P$.

Lemma 1.2 *"Greedy Never Stuck"*
A matching M in a graph G is a maximum matching if and only if there is no augmenting path in M.

Lemma 1.3 *"Greedy Stays Ahead"*
If M is a matching of size k that is of maximum weight among all matchings of size k, and if P is an augmenting path with respect to M of maximum incremental weight, then $M \oplus P$ is a matching of size $k + 1$ that is of maximum weight among all matchings of size $k + 1$.

Both proofs can be found in [11]. One note is that Lemma 1.3 shows we can find the max-weight matching by finding the max-weight matching of size 1, and size 2, and so on. Once we cannot find an augmenting path for size k, we can make sure this max-weight matching of size k is the one we want.

Problem 1.4 *How to find the Max-Cardinality Bipartite Matching?*

First Attempt:
Using algorithm similar to greedy for max-weight bipartite matching.
Running time: $O((m \cdot (n + m))$

Further Optimization: Hopcraft-Karp Algorithm
Intuition: Try to find lots of augmenting paths at once, in order that algorithm can terminates in $O(\sqrt{n})$ time.
Running time: $O(m \cdot \sqrt{n})$

Lemma 1.4 *A maximal set S of vertex-disjoint minimum-length augmenting paths can be found in time $O(m)$.*

The proof of Lemma 1.3 can be found in [11] using "Hungarian Tree".

Problem 1.5 *How to find the Max-Cardinality Non-Bipartite Matching?*

Non-Bipartite graph is more difficult since finding an augmenting path suing DFS may fail due to the existence of "Flowers".

Solution for Flowers: Edmonds Algorithm

Intuition: Shrinking the "Blossom" of the Flower to make Greedy still works.

In each iteration, either shrink a Blossom (at least 2 vertices) or increase $|M|$. Therefore at most $O(n)$ shrinks for each of the $O(n)$ increases, therefore running time is $O(m \cdot n^2)$.

Details of the algorithm and proof of correctness can be found in lecture 1 and 2 of [5]. One note is that the deleting of blossoms is guaranteed not to change the maximum matching result.

1.3 Minimum Spanning Tree

In this section we present some different views of the standard Minimum Spanning Tree problem.

Problem 1.6 *How to find the Minimum Spanning Tree of a graph?*

Kruskal Algorithm:

Iteratively add e with min. weight to link distinct components.

Running Time: $O(m \log m)$ with fancy data structure.

Prim Algorithm:

Iteratively add e with min. weight from C to $\overline{C}$. C has a seed node r.

Running Time: $O(m \log n)$ with fancy data structure.

Borüvka Algorithm:

Iteratively add $e_1, e_2, ..., e_i$ to F, while e_i is linking edge from C_i to $\overline{C}$ with min. weight.

Running Time: $O(m \log n)$ without fancy data structure.

Correctness:

Lemma 1.5 *If edge weight are distinct (without loss of generality) and $V(G) = A \cup B$ a partition, and e is min-weight from A to B. Then every MST contains e.*

This lemma can be proved by contradiction construction.
Running time of Borüvka:

Lemma 1.6 *If we have k components at the start of the while loop, the size of $|e_1, e_2, ..., e_i|$ is no less than $\frac{k}{2}$.*

This lemma is true because each edge e can be used at most twice by components containing its end points. Given the lemma, if we have k components at the start of the while loop, it has less than $\frac{k}{2}$ components at the end. Therefore we only need $\log n$ phases of *Borüvka*, with each phase running time $o(m)$.

Problem 1.7 *What is the running time for* Borüvka *on planar graph?*

From *Euler's Formular*, if the planar graph has n vertices, m edges, and f faces, then $n - m + f = 2$. Then with $n \geq 3$, $m \leq 3n - 6$. So *Borüvka* running time becomes linear, by removing all but the cheapest edge between each pair of components after each stage of the algorithm.

Note that we can do better with Fibonacci Heap in Section 7.2, the running time of *Prim* is reduced to $O(m + n \log n)$. And by using hybrid of *Prim* and *Borüvka*. The algorithm, which has a running time of $O(m \log \log n)$ is as follows:

Algorithm 1.2 *Borüvka/Prim*

Step 1*: Run $O(\log \log n)$ phases of recursive Borüvka algorithm. Now the number of components is less than $\frac{n}{\log n}$.*

Step 2*: Run Prim algorithm using Fibonacci Heap on the component graph.*

REMARK: Borüvka works well on dense graph, and Prim works well on sparse graph. So we first run Borüvka to the graph until it is sparse enough for Prim.

1.4 Matroids

In this section we introduce the concept of *Matroids*, and related *Blue/Red* Rule, and some of their applications. The definitions of these concepts (including *basis*, *circuit*, *cocircuit*) can be found in [11].

Some Examples of Matroids:

- $G =$ undirected graph, $S =$ edge set of G, $I =$ set of acyclic sub-graphs.
- $S =$ any set, I is subsets with no more than k elements.
- $S =$ a set of vectors in a vector space, $I =$ linearly independent subsets.

- G = bipartite graph $G(U, V, E)$, $S = U$, I = "matchable" subset of U.

REMARK: Matroid is so abstract which makes its theorem very strong to be applied using Greedy in many places. But it is yet to be proved to be the necessary/sufficient condition for Greedy.

Definition 1.1 *An acceptable coloring of* (S, I)*, is a coloring Blue/Red,* $B \in I$*,* $R \in I^*$*,* $B \cup R = \emptyset$*.*

The Theorem for B/R rule guarantees that 1) "Greedy B/R Goes Ahead", 2) "Greedy B/R Never Stuck". Therefore if we find the property of Dual Matroid in any problem, we can use Greedy B/R Rule to solve it.

Problem 1.8 *Can matching problem be solved in Matroid framework?*

The matching problem is actually a *Matroid Intersection* problem, in which the Greedy Blue/Red rule does not work any more. Therefore we look at what other operations instead of Blue/Red coloring on an independent set preserve cardinality independence.

1.5 Network Flow

The Definition of *Network Flow* and *Min-Cut Max-Flow* Theorem can be found in [10].

Problem 1.9 *How to solve the max-flow problem?*

Guided by Min-Max Theorem we have a first solution.

First Solution:

Ford/Fulbson Algorithm: add a path in the residual graph whenever find one.

Running time: $O(mC)$, where C is the total out capacity of source node s. Note that this is *pseudo-polynomial* since it is polynomial in the magnitude of the input numbers but not in the number of bits needed to represent C (which is regarded as the "input" size).

First Optimization:

Scaling Max-Flow Algorithm: Choose the path with high bottleneck capacity [10].

Runnning time: $O(m^2 \log C)$

Second Optimization:

Edmonds-Karp Algorithm # 2: Choose the shortest path in the residual graph instead of arbitrary one. The shortest one can be chosen using BFS.

Running time: Since algorithm will terminate after $O(m \cdot n)$ loops at most, its running time is $O(m^2 n)$.

Third Optimization:

Edmonds-Karp Algorithm # 1: Choose the max "width" path in the residual graph instead of arbitrary one. The max "width" path can be chosen by Modified *Dijktra Algorithm* in $O(m + n \log n)$.

Running time: Since algorithm will terminate after $O(m \cdot \ln F)$ loops at most, its running time is $O(m \ln F(m + n \log n))$, where F is the max flow value.

Note Max-Flow Min-Cut Theorem can be applied in many other applications. Here are some examples:

- *Menger Theorem*: In a graph G with vertices s.t. $\exists$ k edges(vertices)-disjoint $s - t$ path $\iff \nexists$ an $s - t$ edge(vertex) cut of cardinality l k
- *König Theorem*: In a bipartite graph max matching size = min vertex cover size.
- *Generalized Hall Theorem*: In a bipartite graph $G(U, V, E)$, a subset $U_0 \in U$ is covered by a matching $\iff \forall U_1 \subseteq U_0, |N(U_1)| \geq |U_1|$ where $N(U)$ is the set of neighbors of U.
- *Dilworth Theorem*: A chain is a totally ordered subset of a partially ordered set. An *antichain* is a set of pairwise incomparable element. $\forall$ finite sets:

  min chain cover = max antichain size

2 NP-Completeness

2.1 Turing Machine

In this section we demonstrate P and NP in the view of Turing Machine.

Definition 2.1 *Given alphabet $\sum$, string $y \in \sum^Z$, set of states $Q(I, H)$, and transition function, Turing Machine on input $X \in \sum^*$ (denoted $m(x)$) produces the unique transcript s.t. string $y_0 = x$, state $q_0 = I$, position $p_0 = 0$. Machine accepts x at time no more than t if the transcript of $m(t)$ has some $s \leq t$ with $q_s = H$.*

Definition 2.2 *$L \in \sum^*$ belongs to P if $\exists$ machine m and time $t(n)$ bound by $O(n^O(1))$ such that*

$\forall x \in \sum^*$, *$x \in L \iff m(x)$ accepts at time no more than $t(|X|)$*

Definition 2.3 *$L \in \sum^*$ belongs to NP if $\exists$ machine m and time $t(n)$ bound by $O(n^O(1))$ such that*

$x \in L \iff \exists y =< x, z >$ *s.t. m(y) accepts at time no more than* $t(|X|)$

Definition 2.4 *If L, $L' \in \sum^*$ we write $L' \leq_m^p L$ of $\exists$ machine m and time bound $t(n) = O(n^O(1))$ such that*

$x \in L' \iff$ *the transcripts of m(x) satisfies* $y_t(|x|) \in L$

2.2 NP-Completeness

To prove L $\in$ NP-Complete, we need to prove:

- L $\in$ NP
- Pick an NP-Complete Language L', $L' \leq_m^p L$ (reduction r)
- (r) runs in polynomial time
- X $\in L' \implies$ r(x) $\in$ L
- X $\notin L' \implies$ r(x) $\notin$ L

REMARK: Since $\implies$ is required but $\impliedby$ is not in the step of reduction proof, we can build a special/extreme case of the targeted problem to reduce from the already known NP-hard problem.
Here are some classic NP-hard problems and their proof sketches:

- The first NP-hard problem: SAT (prove from general programming logic of any NPC algorithms)
- Independent Set (prove from simple gadget and reduce from SAT)
- Vertex Cover (reduce from Independent Set, which is its "dual" problem)
- Set Cover (reduce from Vertex Cover, we can see as general case of VC)
- 3-Coloring (prove from a more complex gadget and reduce from 3-SAT)

Here are some classic NP-hard problems and their proof sketches:

- The first NP-hard problem: SAT (prove from general programming logic of any NPC algorithms)

- Independent Set (prove from simple gadget and reduce from SAT)
- Vertex Cover (reduce from Independent Set, which is its "dual" problem)
- Set Cover (reduce from Vertex Cover, we can see SC as general case of VC)
- 3-Coloring (prove from a more complex gadget and reduce from 3-SAT)
- Set Packing with set size no more than 3 (reduce from Independent Set in graph of max degree 3)
- Trichromatic Set Packing (reduce from Independent Set in 3-edge-colored graphs)j
- Triple Matroid Intersection (reduce from 3-matching)
- Exact Cover (reduce from Independent Set and gadget construction)
- Subset Sum (reduce from Exact Cover and binary representation of numbers)
- Partition (reduce from Subset Sum)
- Knapsack (reduce from Partition)

2.3 Tree Decomposition for NP-hard Problems

Key Idea: Move the exponential scale from the input size to the tree width. If the tree width is much smaller than the input size then we can solve this special case of NP-hard problem in polynomial time.

For definitions of Tree Decomposition and its properties, read Chapter 10 in [10].

Problem 2.1 *How to solve the Max Weight Independent Set problem with bounded tree width?*

Solution: Dynamic programming in a bottom-up way on the decomposed tree.

Problem 2.2 *How to compute the tree width? Or how to check if a graph has a small tree width?*

Observation 2.1 *If* $Treewidth(G) \leq w$ *and we delete or constract an edge of* G*, the resulting graph has treewidth* $\leq w$*.*

We have the following theorem which motivate us to find a *k-linked* set of size $3w$.

Theorem 2.5 *If G contains a (w+1)-linked set of set at least 3w, then G has tree-width at least w.*

3 Randomized Algorithms

Randomized algorithms are applied in many scenarios including exact optimal algorithms (Section 3.1), approximate algorithms (Section 3.2), online algorithms, algorithms for massive data (Section 6.2, 6.3), etc. In the following sections we will give some instances of these categories.

3.1 Randomized Minimum Spanning Tree

Problem 3.1 *Can we get a linear time algorithm for MST problem?*

Idea: Do something with random sampling to sparsify the graph as we run Borüvka algorithm. The sparsifying process should be very fast ($O(m)$ time) and never throw away MST edges.

The first requirement can be achieved by only processing each edge one time during sampling. The second requirement can be achieved by only throwing out F_heavy edges, which can not belong to any MST. F_heavy edges can be found by finding a forest from the sampled sub-graph, and the average quantity of the F_heavy edges that can be filtered in one phase is guaranteed by the property of sampling.

The algorithm details can be found in [7]. One note is that the Borüvka algorithm need to be executed twice at the first step only for the proof of the linearity of the randomized algorithm.

3.2 Randomized Vertex Cover

In this section we show that many randomized algorithms are designed also as approximation algorithm.

Problem 3.2 *Can we get a linear time approximate algorithm for un-weighted vertex cover problem?*

First Solution - Greedy: Always pick max-degree node, delete all incident edge. It is a 2-approximate algorithm.

Second Solution - Dumb Greedy: Pick an edge arbitrarily, add both its end points, and delete all incident edges to one of the end points. It is a 2-approximate algorithm.

Third Solution - Randomized: Pick an edge arbitrarily, add one of its end points randomly, and delete all its incident edges. It is a 2-approximate algorithm.

Note that although the randomized algorithm has the same approximation ratio, in practice it performs better than the dumb greedy algorithm.

4 Approximation Algorithms

4.1 Approximation Ratio

The tough part of approximation algorithms is the analysis: we need to somehow present upper/lower bound of the OPT algorithm. The first problem is a case where we can explore the intrinsic property of OPT itself; the second problem is a case where we can explore the property of the approximate algorithm and relate it to OPT.

Problem 4.1 *How to linearly solve the Travel Salesman problem with 2-approximation.*

Proposition 4.1 *If T is minimum spanning tree of the graph, then $Cost(T) l Cost(OPT)$*

Proposition 4.2 *If $x_1, ..., x_n$ are the vertices of T in the order that DFS visits them, then:*
$Cost(tour x_1, ..., x_n) \leq 2 \cdot Cost(T)$

Combining the above two propositions we get:
$Cost(tour x_1, ..., x_n) < 2 \cdot Cost(OPT)$

Therefore DFS of MST is 2-approximate for this problem.

Problem 4.2 *How to linearly solve the Center Selection problem with 2-approximation.*

Solution: Pick a node arbitrarily, delete all its neighbors and continue. If the center size $|C| \leq k$ output C, else claim there is no such set of k centers with covering radius at most r. One note is that we choose $2r$ in order to prove that the claim is correct.

4.2 Linear Programming for Approximation

From Section 3.2 we know for un-weighted vertex cover problem we have a 2-approximate Greedy algorithm. However, the same Greedy strategy no longer works for this problem.

Problem 4.3 *Can we get a linear time approximate algorithm for Weighted Vertex Cover problem?*

We can prove that Weighted Vertex Cover $\leq_m^p$ Integer Programming, so Integer Programming is NP-Completeness. Now the idea is: relax the Integer Programming to Linear Programming by dropping the int-valued constraint on x, and hopefully there is an approximation bound for this relaxation. *New Solution*: Rounded Linear Programming.

We can prove that 1) the result is indeed a vertex cover; 2) it is a 2-approximated solution. The details of the proof is in [10].

4.3 Linear Programming Duality

Given a linear program: $minc^T \cdot xsubjecttoAX \geq b, x \geq 0$ (Primal) there is another LP: $maxb^T \cdot ysubjecttoA^T y \leq c, , y \geq 0$ (Dual). The detailed definition of the LP Duality can be found in lecture 21 of [6].

REMARK: Many problems can be transferred to LP, and LP's dual can sometimes transformed to some well-known problems that have solutions. Then we can transform back and solve the original problem. Two examples are Network Flow and Weighted Vertex Cover.

Theorem 4.1 *Primal and its Dual problems have same optimal value unless both optimal value are infinite.*

Corollary 4.2 *If x, y are feasible for (P) (D) respectively, and if $C^T x \leq \alpha \cdot b^T y$, then each of x, y is an α-approximation to the optimum of its respective LP.*

One note is the proof sketch: $min(P) \geq max(D)$ is simple, but $min(P) \leq max(D)$ needs constructive method, while both involves matrix computation.

4.4 Approximation Scheme for Knapsack Problem

In this section we present another NP-hard problem that can be solved by fully polynomial-time approximation algorithm. The *rounding scheme* presented can be applied for similar numerical problems.

Problem 4.4 *How to approximately solve the Knapsack problem in linear time?*

First Solution - Combined Greedy:

Intuitively there are two Greedy algorithm, either sort in non-decreasing 1) value/size ratio, or 2) value itself. Although they cannot achieve any

approximation ratio, we can get a 2-approximate algorithm by combining the two: run their algorithms in parallel, and choose the better one.

Second Solution - Rounding:

Represent each value by re-scaling and rounding to a different unit of accuracy. Then use Dynamic Programming method on the re-scaled value.

One note is that by getting the approximate factor $1 - \epsilon$, we need to set the re-scale factor u to $\frac{\epsilon \cdot LB}{n}$ where $LB \leq OPT$. Without any other knowledge we can only set $LB = \max_j v_j$, which is not a very good bound. If we have additional knowledge about the optimal solution's lower bound, we can have a better running time than $O(\frac{n^3}{\epsilon})$ for Dynamic Programming.

5 Online Algorithms

5.1 Competitive Analysis

Here is online algorithm's setup:

- "Adversary" chooses an input sequence $\vec{x} : x_1, ..., x_T$.
- "Algorithm" chooses response sequence $\vec{y} : y_1, ..., y_T$. A deterministic algorithm will choose $y_i = F_i(x_1, ..., x_i)$, A randomized algorithm will choose $y_i = F_i(r, x_1, ..., x_i)$ where r is the random bits.
- Assumption: $r, (x_1, ..., x_T)$ are independent; and F_i is polynomial time (but not required).

Definition 5.1 *[2] ALG is r-competitive if $\exists$ b (may depend on problem size but not on input $\vec{x}$ nor time T), $\forall \vec{x}$:*

$$cost(\vec{x}, OPT) \leq r \cdot cost(\vec{x}, OPT) + b$$

Problem 5.1 *How to solve the Ski Rental problem ($1 each time rent, $k first time buy, once bought, no more rent at all) online?*

First Solution: Buy-At-Once (BAO).

BAO is Ø-competitive: b = k, r = 0.

Second Solution: Rent j = f(k) times then buy if still necessary (Juan).

Juan has strict competitive ratio: $\max \frac{j+k}{j+1}, \frac{j+k}{k}$, since $cost(\vec{x}, OPT) = \min \sum x_i, k$ and there are only three possible situations.

Problem 5.2 *How to solve the List Update problem (cost = search cost + reordering cost) online?*

First Solution: Frequent Count (FC). Count how many times each a_i has been requested, sort by decreasing frequency.

FC can be bad given a sequence that initially request each element $a_i i$ times and then request the last element n times.
Second Solution: Move-To-Front (MTF). Always move x_i to front after finding it.

Similar amortized analysis can give us the following conclusion:

$cost(MTF) \leq 2 \cdot cost(OPT) + \binom{n}{2}$

5.2 Online Prediction

Problem 5.3 *How to solve the Reverse 20 Question problem?*

Solution: Guess the answer consistent with majority of "survivors". # of mistakes $\leq \lceil \log n \rceil$.

Problem 5.4 *[9] How to solve the Sneaky Reverse 20 Question problem (adversary can tell lies)? Goal is that* $(\#mistakes) \leq a \cdot (\#lies) + b$.

First Solution: Weighted Majority with ϵ.
# of mistakes < # lies $\cdot \frac{2}{1-\epsilon} + \ln n \cdot \frac{2}{\epsilon}$.

Second Solution: Hedge with ϵ.

The main difference between *Weighted Majority* and *Hedge* is that *Hedge* use a sampling method with probability proportional to weight while *Weighted Majority* always choose the weighted majority vote.
# of mistakes < # lies $\cdot \frac{1}{1-\epsilon} + \ln n \cdot \frac{2}{\epsilon}$.

5.3 LP Duality for Game Theory

In Game Theory the goal is to maximize total payoff given payoffs at each step after the algorithm chooses an action. This can be regarded as a generalized problem for Sneaky Reverse 20 Questions. And we have *Hedge* to solve it. Now we give a special case of this game problem: Zero Sum Games, and its properties.

Theorem 5.2 *For Zero Sum Games,* $\max_p \min_q u(p,q) = \min_q \max_p u(p,q)$

The proof of this Min-Max theorem involves two phases: 1) $\max_p \min_q u(p,q) \leq \min_q \max_p u(p,q)$ is simple; 2) $\max_p \min_q u(p,q) \geq \min_q \max_p u(p,q)$ needs a constructive proof using *Hedge* to achieve this inequality. Following this theorem we get another theorem for LP Duality.

Theorem 5.3 *If we build the payoff matrix as* $u(i,j)^T = \frac{a_{ij}}{b_i c_j}$ *then if Player 1 has a v-forcing strategy (the payoff is at least v) then the value of (P) is at most* v^{-1}*; if Player 2 has a w-forcing strategy then the value of (D) is at least* W^{-1}*.*

REMARK: Min-Max value is proved to be achieved using *Hedge*, therefore Player 1 and 2 have the same Min-Max value-forcing strategy at best. Therefore by generating a game with corresponding u(i, j) we can get an approximate optimal solution for the LP problem. Examples include Network Flow, Ski Rent, Adwords [3], etc.

6 Algorithm for Massive Data

6.1 Streaming Model

Input: A sequence of $x_1, x_2, ..., x_T \in \sum$, where $|\sum| = n$.

Algorithm: A Turing Machine with *space bound* $s(n,T)$ (possibly also time bound $t(n,T)$ per input element). Typically $s(n,T) = O(1), O(polu(\log n, \log T)), O(\log n)$.

Output: After observing entire sequence, output a YES/NO decision for certain question, or a number, or an element of the stream, or a "sketch" of the stream.

Problem 6.1 *How to sample a uniformly random element, without knowing* T*, and with limited space?*

Solution: $x \leftarrow x_1$, for $i = 2, 3, ..., T$, replace $x \leftarrow x_i$ with probability $\frac{1}{i}$.

For this algorithm $s(n,T) = O(\log n + \log T)$. One note is that the correctness of the algorithm can be proved using induction [12].

Problem 6.2 *How to find the majority element with limited space? Specifically, if* $\exists x \in \sum$ *such that more than* $\frac{T}{2}$ *elements of* $x_1, ..., x_T$ *are equal to* x*, then output* x*. Else do something arbitrary.*

Solution: Maintain x and a counter $c[x]$ (initialize as NULL and 0), for each x_i: if $x = NULL$ then $x \leftarrow x_i$; if $x = x_i$ then $c[x] + +$; if $x \neq x_i$ then $c[x] - -$; if $c[x] = 0$, then $x \leftarrow NULL$.

Space requirement: $s(n,T) = O(\log n + \log T)$. One note is that every time the algorithm throws away its current x or decrease its counter it also throws away a distinct x_i. If $\exists$ a majority element, the algorithm can't throw away all copies of it [8].

The following two problems illustrate some strategy using randomized algorithm to solve stream problems. Details can be found in [1] and lecture 17, 18 of [4].

6.2 Approximately Counting Distinct Element

Problem 6.3 *Let $m = \#x \in \sum | \exists i s.t. x_i = x$. Output an α-approximate result to m, i.e. a number $\tilde{m}$ such that $\frac{m}{\alpha} < \tilde{m} < \alpha m$.*

REMARK: For these kind of problems a "strict" approximate result is even hard to achieve, therefore we are trying to find a "relaxed" solution such that the algorithm will perform "well" (for some ratio $\alpha > 2$) with a very high probability ($\leq 1 - \frac{2}{\alpha}$).

First Solution - Random Hashing:

Use a random (means uniformly distributed and mutual independent) hash function $h : \sum \to \{1, 2, ..., l\}$. Initial $Answer \leftarrow NO$. For each x_i, if $h(x_i) = 1$ then $Answer \leftarrow YES$. At the end output $Answer$.

Approximation Ratio: $Pr(Answer = NO | m \geq 2 \cdot l) < \frac{1}{7.4}$; $Pr(Answer = NO | m < l) > \frac{1}{4}$.

Space Requirement: $O(\log n)$.

Second Solution - Random Round:

Choose uniform random $h(x) \in (0, 1)$ independently for every x $\in \sum$. For $y \in (0, 1)$ let $r(y)$ = the unique r such that $2^{-r} \leq y < 2^{1-r} = 1 +$ the initial # of zeros in the binary expansion of y. Initialize $r_{max} = 1$, for each x_i compute $r(h(x_i))$, if $r(h(x_i)) > r_{max}$ then $r_{max} \leftarrow r(h(x_i))$. Output $2^{r_{max}}$.

Approximation Ratio: Fix a value r and some approximation ratio $\alpha > 2$. If $\alpha m < 2^r$ then with probability $1 - \frac{1}{\alpha}$ outputs answer $< 2^r$; If $\alpha 2^r < m$ then with probability $1 - \frac{1}{\alpha}$ outputs answer $\geq 2^r$.

Conclusion: Answer is correct up to factor α with probability $\geq 1 - \frac{2}{\alpha}$. To improve failure probability from $\frac{2}{\alpha}$ to $\delta > 0$, repeat $\log_{\frac{\alpha}{2}} \frac{1}{\delta}$ times in parallel. To improve factor from α to $1 + \epsilon$, use estimator based on $\frac{1}{\epsilon}$ order statistic instead of minimum.

6.3 Approximately Computing Second Frequency Moment

Definition 6.1 *For a stream $x_1, ..., x_T$, define the frequency m_x for $x \in \xi$ by $m_x = \#i | x = x_i$. And $F_i = \sum_{x \in \xi} m_x^i$.*

F_0 = # distinct elements in stream

F_1 = # elements in stream = T

F_2 = second frequency moment.

F_2 represents the skewness of the stream: $Pr($ two random elements match) $= \frac{F_2}{T^2}$.

Problem 6.4 *How to approximate second frequency moment F_2 with limited space?*

Solution:

Sample random $y_x \in \{\pm 1\}$ for every $x \in \sum$. Compute $Z := \sum_{i=1}^{T} Y_{x_i}$, let $W = Z^2$. Claim $\mathbb{E}[W^2] = F_2$, $Var[W] < 2 \cdot F_2^2$

Space Requirement: $\log T$ bits to store Z, $\log n$ bits to store y_x.

Conclusion: To improve the variance, use k independently parallel repetitions to get estimation $W_1, ..., W_k$ and take their average $\bar{W} = \frac{1}{k}(W_1 + ... + W_k)$. Therefore $Var[W] < \frac{2}{k} \cdot F_2^2$. A further improvement is to compute the median of $\log \frac{1}{\delta}$ averages, thus if $k = O(\log \frac{1}{\delta}/\epsilon^2)$ then with probability > 1 - δ we get a $(1 + \epsilon)$-approximation of F_2.

7 Data Structure

7.1 Binomial Heap

The functions supported and their running time is shown below, note $h = heap$, $i = (key, value)$. The requirement for the heap is that any tree whose cost has ranked i must have $exponential(i)$ many nodes.

Table 1: Binomial Heap Functionality

Function	Running Time
$makeheap(i)$	$O(1)$
$findmin(h)$	$O(1)$
$deletemin(h)$	$O(\log(n))$
$insert(h, i)$	$O(1)^*$
$meld(h, h')$	$O(\log(n))$

Amortized Analysis: If we regard $insert(h, i) = meld(h, makeheap(i))$, and since $meld(h, h')$ is just like binary addition, each addition that cause a bit to carry will leave a 0 at that bit for the convenience of the coming addition. For example, $meld(01, 1') = 10$ will cause a carry to update the second bit, and leave the first bit 0, so that next time $meld(10, 1') = 11$ will have no cause at all. So the amortized cost for insert is actually $O(\log(n))$.

We can have *Eager* and *Lazy* implementation for meld. While Eager Meld guarantee that each kind of tree B_i has at most one appearance in the heap

by linking the trees as soon as meld is executed, Laze Meld simply use a linked list so that meld is executed in constant time, and an additional *Cleanup Phase* is needed to run through all the trees in the heap and insert them into a new heap, linking them as necessary

7.2 Fibonacci Heap

The functions supported and their running time is shown below, note $h = heap$, $i = (key, value)$.

Table 2: Binomial Heap Functionality

Function	Running Time
$makeheap(i)$	$O(1)$
$findmin(h)$	$O(1)$
$deletemin(h)$	$O(\log(n))$
$insert(h, i)$	$O(1)^*$
$meld(h, h')$	$O(\log(n))$
$decreaseKey(h, i, \triangle)$	$O(1)^*$

By using $deletemin(h)$ and $decreaseKey(h, i, \triangle)$, we can get a new function $delete(h, i)$ in $O(\log n)$.
The trouble with $decreaseKey$ is that after decreasing $value(i)$, heap-ordering could be violated. And if we simply cut out the sub-tree rooted at i and meld it into h, the contents of the heap may not be binomial trees any more.
Solution: Whenever two children are cut away from a node, cut out that node as well.
Amortized Analysis: The similar analysis give us the conclusion that this solution guarantees the exponential children requirement.

8 Conclusion

- *For proving algorithm correctness*:

    1. Enumerate all possible cases, and prove the correctness one-by-one: Algorithm 1.1, Juan for Ski Rent, etc.
    2. Look for some structures of the problem and try to match them with the algorithm property: computing tree-width, etc.
    3. If the algorithm has the similar strategy at each step, consider using induction to utilize this structure: random sampling in data stream, etc.

- *For proving NP-Completeness*

1. Since we are only required to reduce any case in the known NP-hard problem to probably a "special" case of the target problem. Our reduction can add more restriction on our target problem to make it special enough: Graph Cover $\Longrightarrow$ Set Cover, Independent Set $\Longrightarrow$ Set Packing, etc.
2. Choosing the appropriate reduce problem is tricky. Here are some guidance: 1) Similar properties of the two problems: optimization(packing, covering), satisfaction(matching, constraint sat, coloring); also numerical problems, set problems, graph problems, etc.
3. Gadget construction for graph problems is tricky. One possible way to make the gadget matching the problem constraint is to have a "simple but huge" gadget, so that the size of the gadget is big enough to present complex constraint: 3-SAT $\Longrightarrow$ Scheduling, etc.
4. Sometimes the problem context is not familiar and might be complicated. One possible way to do that is building additional constraint to eliminate the possible "out-of-control" complicated situations: Set Cover $\Longrightarrow$ tree isomorphism, etc.

- *For proving approximation ratio*

1. Try to relate some intrinsic property of the input to OPT: Travel Salesman Problem, Load Balancing problem, etc.
2. Try to utilize some properties of the approximate algorithm, such as local-optimality(Multi-way Cut problem), coverage(Center Selection problem, Cluster Cover problem), etc.
3. Approximation ratio can also be regarded as probabilities of failure/success, which can be computed from Bayesian.
4. Approximation ratio can also be regarded as mean and variance if the algorithm is randomized. The computation of the mean and variance mostly involves the independence of the sampling.

- *For problem solving strategy*

  – When an existing method cannot apply to a new problem, what shall we do?

  1)Try a totally new method: Integer Programming for weighted Vertex Cover, etc.

  2)Find what differences of the new problem to the old one make it fail and try to remove this difference: Algorithm 1.2, etc.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci*, 58(1):137–147, 1999.

[2] Avrim Blum. Approximation and Online Algorithm Class at CMU, Spring 2000. `http://www.cs.cmu.edu/~avrim/Approx00/index.html`.

[3] N. Buchbinder, K. Jain, and S. Naor. Online primal-dual algorithms for maximizing ad-auctions revenue. In *ESA*, 2007.

[4] Shuchi Chawla. Advanced Algorithms Class at University of Wisconsin, Fall 2007. `http://pages.cs.wisc.edu/~shuchi/courses/787-F07/`.

[5] Michel Goemans. Combinatorial Optimization Class at MIT, Spring 2004. `http://www-math.mit.edu/~goemans/18997-CO/topics-co.html`.

[6] Anupam Gupta and R. Ravi. Approximation Algorithms Class at CMU, Fall 2005. `http://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/`.

[7] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[8] R. M. Karp, S. Shanker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Systems*, 28(1):51–55, 2003.

[9] Robert Kleiberg. Learning, Games, and Electronic Markets Class at Cornell, Spring 2007. `http://www.cs.cornell.edu/courses/cs683/2007sp`.

[10] J. Kleinberg and É. Tardos. *Algorithm Design.* Addison-Wesley, 2005.

[11] D. C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, 1991.

[12] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Mathematical Software*, 11(1):37–57, 1985.