TDA251 Algorithms, advanced course, 2011–12–09

Exercises 1–13

Simon Sigurdhsson* (900322–0291)

Exercise 1

Starting with the lower bound of the optimal solution T^* derived in the first lecture,

$$T^* \geq \frac{1}{m}\sum_i t_i = \frac{A}{m}$$

and the upper bound of the greedy solution T derived during the same lecture,

$$T \leq \frac{1}{m}\sum_i T_i + \max_i t_i = \frac{1}{m}\sum_i t_i + \max_i t_i = \frac{A}{m} + L$$

we can easily set up an expression for the approximation ratio of the greedy algorithm:

$$\frac{T}{T^*} \leq \frac{\frac{A}{m} + L}{\frac{A}{m}} = 1 + \frac{L}{A}m.$$

Trivially, we then have

$$\lim_{\frac{L}{A}\to 0} \frac{T}{T^*} \leq \lim_{\frac{L}{A}\to 0} 1 + \frac{L}{A}m = 1.$$

Exercise 2

Counterexample As a counterexample, consider bins with capacity $K = 7$ and the set of items $W = \{6,6,1,1\}$. The greedy algorithm will require three bins, creating a set of bins $\{\{6\},\{6,1\},\{1\}\}$. The optimal case clearly consists of two bins with the configuration $\{\{6,1\},\{6,1\}\}$.

*ssimon@student.chalmers.se

Approximation ratio Consider the optimal solution T^*. A trivial lower bound for this solution is given by

$$T^* \geq \frac{1}{K} \sum_{i=1}^{n} w_i = \frac{W}{K},$$

where we define W as the sum of all weights. Similarly we can define an upper bound for the greedy solution T, although this requires some motivation.

Consider two consecutive bins b_i and b_{i+1}. Start by making the observation that it is impossible for both these bins to be less than half full at the same time, since that would lead to a contradiction: any item w_j that doesn't fit in b_i must either be larger than $\frac{K}{2}$, meaning b_{i+1} is at least half full, or smaller than that, meaning meaning b_i is at least half full since w_j didn't fit. In the latter case bin b_{i+1} may of course also be more than half full since item w_{j+1} may still fit in there, but in the worst-case scenario $w_{j+1} > K - w(b_{j+1})$ and b_{j+1} is less than half full.

This means that in the worst-case scenario, exactly $\frac{T}{2}$ of the T bins given by the algorithm are more than half-full. These half-full bins cannot contain all items (since the other $\frac{T}{2}$ bins must have a combined weight of at least $\frac{T}{2}$), but they can have a maximum weight of K each. This means that the inequality

$$\sum_{i=1}^{n} w_i = W \geq \frac{T}{2} K$$

holds, which gives us an upper boundary on the algorithm's solution T:

$$W \geq \frac{T}{2} K \implies T \leq 2 \frac{W}{K}.$$

With this information, we can easily calculate the approximation ratio $\frac{T}{T^*}$:

$$\frac{T}{T^*} \leq \frac{2\frac{W}{K}}{\frac{W}{K}} = 2.$$

Exercise 3

Problem 3.1 It is not always true that $\Pr(A \wedge B) = \Pr(A) \cdot \Pr(B)$. This is only true under the assumption that the events A and B are *independent*.

Problem 3.2 This is always true. Additionally, if the events A and B are mutually exclusive, the inequality can be reduced to an equality ($\Pr(A \vee B) = \Pr(A) + \Pr(B)$).

Problem 3.3 A random variable, roughly speaking, is the result of some random process. It is almost always used in terms of its expected value, probability density function, cumulative density function or similar properties. Formally, it is a function $X : \Omega \to \mathbb{R}$ defined on a probability space (Ω,F,P).

Problem 3.4 The expected value of a random variable X is rougly speaking defined as the weighted average of the values it can take. For real-valued random variables, the formal definition is

$$\mathrm{E}[X] = \int_{\Omega} X \,\mathrm{d}P,$$

where P is a probability measure and Ω is the sample space. This can often (i.e. when the probability distribution of X has a probability density function $f(x)$) be simplified to

$$\mathrm{E}[X] = \int_{-\infty}^{\infty} x f(x) \,\mathrm{d}x.$$

For discrete random variables, the integral becomes a sum:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} x_i p_i.$$

The expected value tells us what value X most likely will have.

Problem 3.5 The sum of two *independent* random variables X and Y is defined as the convolution of their probability density functions $f_X(x)$ and $f_Y(y)$:

$$f_Z(z) = (f * g)(z) = \int_{-\infty}^{\infty} f(z-y) g(y) \,\mathrm{d}y.$$

The sum of two dependent random variables X and Y is defined using their joint distribuion $f_{X,Y}(x,y)$:

$$f_Z(z) = \int_{-\infty}^{\infty} \int_{-\infty}^{z-x} f_{X,Y}(x,y) \,\mathrm{d}y \mathrm{d}x.$$

The product of the same (not nessecarily independent) random variables has been shown (Rohatgi 1976) to derive from the probability density function

$$f_Z(z) = \int_{-\infty}^{\infty} f_{X,Y}\left(x, \frac{z}{x}\right) \frac{1}{|x|} \,\mathrm{d}x,$$

where $f_{X,Y}(x,y)$ is the joint distribution of X and Y.

Problem 3.6 This is called linearity of expectation, and is always true.

Problem 3.7 This is only true for *independent* random variables X and Y.

Exercise 4

Suppose that we know the optimal radius r. We can then say that our optimal solution T^* is bounded by (or indeed equal to) r. Now let us analyze the approximative solution. One can easily argue that the worst possible selection of p is a point on the boundary of the disc covering the optimal solution. All other selections will either be better (i.e. inside the disc) or will not be selected by the algoithm (due to being in a group that is too "wide"). The algorithm will then (in the worst case) select all other points in the optimal solution — these can be at most $2r$ away from the initial point, since they are also covered by the disc.

This gives us a simple upper bound on the approximative solution: $T \leq 2r$. We use this, along with the optimal solution, to provide an approximation ratio:

$$\frac{T}{T^*} \leq \frac{2r}{r} = 2.$$

Exercise 5

Every set B_j will pay a price $p_j \geq 0$ for being hit. Consider any hitting set S, and the collection $T(a)$ of sets from S containing the element a. We say that the prices are fair if $\sum_{j \in T(a)} p_j \leq w(a)$, i.e. the payments of all sets containing the element a do not exceed the weight of $a \in S$. If prices are fair, we clearly have $\sum_{a \in S} \sum_{j \in T(a)} p_j \leq w(S)$. Since S is a hitting set every set B_j will appear at least once in that sum, and thus $\sum_j p_j \leq \sum_{a \in S} \sum_{j \in T(a)} p_j \leq w(S)$. In effect, the sum of any fair prices is a lower bound for the cost of any hitting set, in particular the opimal hitting set.

Instead of solving the hitting set problem directly, we attack the dual problem of constructing fair prices that maximize $\sum_j p_j$, and use these to construct a cheap hitting set. This is very easy: Call a set B_i tight if $\sum_{j \in T(a)} p_j = w(a)$ for some $a \in B_i$. Initially, we set all $p_j = 0$. We then proceed to take some j and raise p_j until one of the sets $B_k \in T(a)$ is tight, where $a \in B_j$. This is repeated for as long as possible, and the hitting set S is given by the set of tight sets.

Approximation ratio Now, consider the optimal solution S^*. As explained earlier, the optimal solution has a lower bound $w(S^*) \geq \sum_j p_j$. An upper bound for the algorithm's solution S is not hard to find; noting that for all $a \in S$ we have $\sum_{j \in T(a)} p_j = w(a)$. Summation over the elements $a \in S$ gives us $\sum_{a \in S} \sum_{j \in T(a)} p_j = w(S)$, and since every element appears at most as many times as its set B_j has members, we get $w(S) \leq b \sum_j p_j = b w(S^*)$ and an approximation ratio of b.

Exercise 6

First, we show that the problem is in NP (by providing a polynomial-time verifier), then we show that the NP-complete Set Cover problem can be reduced to our problem, and finally we show that any solution s to the Set Cover problem can be transformed to a solution to our problem.

The problem is in NP A simple, polynomial-time algorithm to verify a solution can for example consist of a breadth-first search from r trying to find all $t \in T$ by following the edges in F. Breadth-first-search algorithms are $O(|V| + |E|)$ and thus the verifier is polynomial.

A reduction from the Set Cover problem As proposed in the exercise: Given an instance of Set Cover with m subsets S_i of U and unit weights, consider the graph G that has nodes $\{r\} \cup U$ and where every S_i is represented by a "set node". An element node is adjacent to a set node if the element belongs to the set. All set nodes are adjacent to r, and the element nodes are terminal nodes, i.e. $T \equiv U$.

This reduction is polynomial-time, as a simple algorithm for reducing the problem is as follows:

1. Create G and add r to the graph.
2. For every element $u \in U$, add u to T (and consequently, G).
3. For every set S_i, add an element g to G and create an edge from g to every vertex $t \in T$ that corresponds to an element in S_i. Additionally, create an edge from g to r.

This algorithm is $O(1 + |U| + |U||S_i|) = O(|U||S_i|)$.

A solution of Set Cover also solves our problem Consider a solution S to the set cover problem and the reduction given above. S will choose one or several subsets S_i so that all elements in U are represented in their union. In the reduction, selecting a subset S_i is equivalent to including the corresponding node in the graph F. Since all elements in U are represented in the solution S, all nodes in T will be connected to at least one selected set node. All set nodes are also connected to r, and thus the solution also solves our problem.

However, there is also the question of optimality. Suppose the solution S^* to the Set Cover problem is minimal. If that is the case, there are as few subsets S_i^* as possible (let's say there are n such subsets), which in turn means that as few nodes as possible will be included in G. Since the cardinality of the graph, i.e. the number of vertices, depends completely on n (the cardinality will be $n + |U| + 1$, where $|U|$ is constant for each instance of the problem), a minimal solution to the Set Cover problem will also minimize the cardinality of our problem.

Exercise 7

Let us set up an equivalent maximum flow problem: consider the bipartite graph $G = (X,Y,E)$, and create a new graph $G' = (X,Y,E')$, where E' are the edges E directed from $x \in X$ to $y \in Y$. Additionally, insert a source node s connected to all $x \in X$ and a sink node t connected to all $y \in Y$ into the graph. We can now add flow constraints (capacities) to all these edges.

For all edges in E', set a capacity of (0,1). This will indicate wether the edge is selected or not (if the flow is 1, the edge will be selected). For all edges from s to $x \in X$, set a capacity of (0,2), representing the number of edges that can be connected to $x \in X$ (0, 1 or 2). Lastly, set the capacity of all edges from $y \in Y$ to t to (0,1), indicating wether the node $y \in Y$ is served at all.

After finding a maximum flow, it should be clear that the solution is valid and selects as many $y \in Y$ as possible. Finding the maximum flow can be done using for instance the Ford-Fulkerson algorithm.

Exercise 8

Consider the graph $G = (Q,E)$, i.e. a graph consisting of all the squares of the grid connected by edges E so that each square is 4-connected to its neighbours. Our task now is to find a bipartite perfect matching, i.e. a matching that divides Q into two sets X and Y so that $Q = X \cup Y$. This bipartite matching will solve the problem, as it will connect each square to *one* (and only one) of its neighbours creating pairs of two that fill the set Q, if this is possible.

The division of the set Q can be made before trying to solve the problem; it is easy to see that if we colour the squares akin to a chessboard (formally, let all $q_{ij} \in Q$ with $i + j = 0 \mod 2$ be black, and all other squares be white), we can let all white squares be in X and all black squares be in Y. It is trivial to see that any domino must be placed so that one end is on a white square and one is on a black square. Thus, we have a graph $G = (X,Y,E)$ to perform a perfect bipartite matching on. This is easily solved by the Hopcroft-Karp algorithm (Hopcroft and Karp 1971), which runs in $O(m\sqrt{n})$.

Exercise 9

Start with the graph $G = (S,T,E)$, where S are the sources, T are the sinks and E contains all edges from S to T that are present in the original graph. Our task is then to find a perfect matching in the graph, i.e. a selection of edges $E^* \subseteq E$ that connects every source to one and only one sink, and vice versa.

For each $e \in E$ assign a capacity $c(e) = [0,1]$ and add a direction to the edge, from a

source $s \in S$ to a sink $t \in T$. If the edge connects two sources or two sinks, discard it as it cannot be part of the solution. Now, add a super source s_0 that with edges of capacity 1 to every source $s \in S$, as well as a super sink t_0 with edges of capacity 1 from every sink $t \in T$.

Using the Ford-Fulkerson algorithm on this problem yields a set of edges carrying the flow 1. We can interpret these as edge-disjoint paths, and since the paths from super source to actual source (and from actual sink to super sink) all carry the flow 1 as well, the flow will only connect one edge to each source (and sink). From these edges we can *construct* edge-disjoint paths by from the source adjacent to each edge to the sink on the other end. This will give us k paths of length 1 connecting each source to a sink, and these paths will be edge-disjoint.

Since the edges conform to the requirements of the problem, so will the paths and as such they will solve the problem. The Ford-Fulkerson algorithm runs in polynomial time, and so does the construction of the flow problem (this is easy to see), which means the problem is polynomial.

Exercise 10

10.1: Deterministic algorithm The deterministic algorithm will, in the worst case, require n steps to complete. This is due to the fact that no matter what algorithm we use, since the array is not sorted we cannot use any clever divide-and-conquer algorithm but must instead check every element until x is found. In the worst case, we check all other elements before finding x.

10.2: Stochastic algorithm Consider a bin of n items of which one (x) is red and the rest are black. Clearly, finding x using a stochastic algorithm while not repicking any item is the same as removing items from this bin until we find a red one, and our random variable for which we want an expected value is the number of items picked before we get to the red one. It is well-known that this problem of selection without replacement lends itself to the hypergeometric distribution.

The hypergeometric distribution describes the probability of k successes in n draws from the bin, with no respect to the sequence of items (i.e. wether our red item was picked last or not). Thus, it does not accurately describe our problem. It can however use the distribution to reason that the probability of picking the red item on the ith query is exactly $\frac{1}{n}$ (Wikipedia 2011).

With that information, calculating the expected value is easy:

$$\mathrm{E}[X] = \sum_{i=0}^{n} i\frac{1}{n} = \frac{n+1}{2}.$$

10.3: (Not) disregarding chosen items The other method, which doesn't disregard items that are already chosen, is not preferrable. Everytime we pick an item, there is a probability $\frac{1}{n}$ that we will pick x. The number of trials needed until we pick x will be geometrically distributed, and as such will have an expected value $\mathrm{E}[X] = \frac{1}{p} = n$ (where $p = \frac{1}{n}$ is the probability of picking x), which is just as "bad" as the deterministic algorithm. Even worse, this means that half the time it will take *more* than n steps to find x, whereas the other method *never* takes more than n steps. Clearly, remembering what items we already picked is preferred.

Exercise 11

11.1: At most 3 literals If every clause has *at most* 3 literals, the initial deduction that at least $\frac{7}{8}$ of the clauses are satisfied with random assignment no longer holds. In fact, in the worst case a clause with only one literal is satisfied with the low probability of $\frac{1}{2}$, which is significantly less. Assuming the number of literals in a clause is uniformly distributed on {1,2,3}, the expected ratio of satisfied clauses will instead be

$$\mathrm{E}[X] = \frac{1}{3}\sum_{i=1}^{3} \Pr(X \text{ satisfied} \mid X \text{ has } i \text{ clauses}) = \frac{1}{3}\left(\frac{1}{2} + \frac{3}{4} + \frac{7}{8}\right) = \frac{17}{24} = 0.708\bar{3}.$$

Analogously to the case of exactly 3 literals, we now deduce that $\frac{17}{24}k$ is a *lower* bound for the number of clauses that can be satisfied, and that repeated random assignment eventually will satisfy this lower bound. We also analyze the expected number of iterations in the same manner:

$$\frac{17k}{24} = \sum_j jp_j = \sum_{j<\frac{17k}{24}} jp_j + \sum_{j\geq\frac{17k}{24}} jp_j,$$

$$\frac{17k}{24} \leq \sum_{j<\frac{17k}{24}} k'p_j + \sum_{j\geq\frac{17k}{24}} kp_j = k'(1-p) + kp \leq k' + kp.$$

This of course gives us that $kp \geq \frac{17k}{24} - k'$ which is at least $\frac{1}{24}$ (completely analogously to the case of exactly 3 literals). This means random assignment succeeds with probability $p \geq \frac{1}{24k}$ and as such the expected waiting time for success is $24k$ iterations.

This means that the algorithm is three times slower when the number of literals per clause is allowed to vary. Additionally, it has a slightly lower guarantee with only $0.708\bar{3}k$ clauses in every input.

11.1: Exactly k literals If every clause has exactly k literals, we can perform an analysis which is nearly identical to the one for 3 literals. We start by noting that the probability of a clause being satisfied no longer is $\frac{7}{8}$ but instead $\frac{2^k-1}{2^k}$. As such, following the analysis in the lecture notes almost to the letter, there exists a solution with at least that fraction of clauses satisfied, and we can of course construct an algorithm that takes on average $2^k n$ (where n is the number of clauses) iterations to find this solution.

An interesting consequence of this is that while the probability of a solution existing increases with k (the probability of a clause being satisfied tends to one as k tends to infinity), so does the time needed to find a solution. In effect, the algorithm is exponential with respect to k.

Exercise 12

Consider a mixed algorithm in which we first perform $s-1$ steps of the deterministic algorithm followed by one step of the stochastic algorithm. In every such cycle, the stochastic algorithm sees $s-1$ steps "wasted", and as such it will take s times longer to reach a solution. Hence, the expected time to find a solution is $sf(n)$, where $s \geq 2$ and $s \in \mathbb{N}$.

In the worst case, the deterministic algorithm finished before the stochastic one, having spent a total of $O\left(O(g(n))\left(1+\frac{1}{s}\right)\right) = O(g(n))$ steps. The worst-case complexity of this hybrid algorithm is thus $O(g(n))$.

Exercise 13

13.1: A family of colorings What we wish to prove is that with a family of colorings F such that $|F| = O(ke^k \log(n))$, there is a non-zero probability that for every k-subset $S^{(k)}$ of the original set, there exists an $f \in F$ that colors the elements of said $S^{(k)}$ in such a way that all k colors are represented.

We can easily work out that the probability of a given $f \in F$ coloring a given $S^{(k)}$ is $\frac{k!}{k^k}$. Using the inequality $\frac{k!}{k^k} > e^{-k}$, we then have

$$\Pr(\text{A given } f \in F \text{ colors a given } S^{(k)}) = \frac{k!}{k^k} > e^{-k},$$

which in turn can be used to obtain the probability that some $f \in F$ colors the subset:

$$\Pr(\text{Some } f \in F \text{ colors a given } S^{(k)}) = \left(1 - \left(1 - e^{-k}\right)^{ke^k \log(n)}\right).$$

With this being true for all of the $\binom{n}{k}$ subsets $S^{(k)}$, we can also compute the probability that every $S^{(k)}$ is colored by some $f \in F$, using the fact that there are $\binom{n}{k}$ ways of generating a subset $S^{(k)}$ from S:

$$\begin{aligned}\Pr(\text{Every } S^{(k)} \text{ is colored by some } f \in F) = \\ = 1 - \Pr\left(\bigcup_{S^{(k)}} (\text{No } f \in F \text{ colors a given } S^{(k)})\right) \leq 1 - \sum_{S^{(k)}} \left(1 - e^{-k}\right)^{ke^k \log(n)} = \\ = 1 - \binom{n}{k}\left(\left(1 - e^{-k}\right)^{ke^k \log(n)}\right) \xrightarrow[k<n\to\infty]{} 1.\end{aligned}$$

Since the probability is strictly larger than a quantity that tends to one (and is larger than zero for $n > 1$), we must conclude that there exists at least one family of colorings F that satisfies the requirements.

13.2: Detecting paths Since we now know that the family F exists, we can safely assume that given enough randomly generated colorings, we will find one that colors a specific k-subset of the graph with one of each colour. Suppose this subset $S^{(k)}$ is a path of length k in the graph. The probability that this path is k-colored is, as outlined above, larger than e^{-k}. As such, if we randomly select $O\left(e^{-k}\left(ke^k \log(()n)\right)\right) = O(k \log(()n))$ colorings from the family F, we can assume that one of these k-colors k-length path of the graph G. This can be tested by simply checking if any of the subsets that are k-colored by the selected coloring are in fact paths in G. As such, given a family F of colorings, we can find a path of length k in the graph in $O(k \log(()n))$ expected time.

13.3: Hashing scheme If one saves the particular $f \in F$ that k-colors the subset $S^{(k)}$ containing the k elements you want to save, that specific coloring f could be used as a hash. But since the size of F is so large in k, this is impractical for larger numbers of elements. One could select a $k \approx \log(()n)$ to gain bins of size $\frac{n}{\log(()n)}$ in which to store elements (using e.g. binary search to sort through them), resulting in $O(k \log(()k) \log(()n))$ colorings to test. This is more reasonable.

References

Hopcroft, J. E. and Karp, R. M. (1971) A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*; oct. 1971 , pp. 122–125.

Rohatgi, V. K. (1976) *An Introduction to Probability Theory Mathematical Statistics*. New York: Wiley.

Wikipedia (2011) Hypergeometric distribution — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Hypergeometric_distribution&oldid=461825433.