TDA251 Algorithms, advanced course, 2011–12–13

Home exam

Simon Sigurdhsson* (900322–0291)

1 Is the Load Balancing algorithm FPTAS?

Recall that a polynomial-time approximation scheme is any algorithm for an optimization problem that approximates the optimal solution within a factor $1+\varepsilon$, while also being polynomial in n (but the running time may depend arbitrarily on $\frac{1}{\varepsilon}$. A *fully* polynomial-time approximation scheme, however, must be polynomial in both *n and* $\frac{1}{\varepsilon}$.

When analyzing the proposed algorithm, one could fairly easily prove an approximation ratio of $1+m\frac{L}{A}$. You'd be tempted to set $\varepsilon = m\frac{L}{A}$, but since m, L and A all depend directly on the problem, we can't really choose our ε freely. As such, the problem isn't FPTAS.

2 Selecting coloured points: An approximation ratio

Consider the algorithm's solution as the number of colours increase indefinitely. The absolute worst-case scenario in this case is that one point of colour c_1 resides at (0,0), while all other points are equidistibuted on the circle of radius r around it. This can be considered a worst-case scenario since we'd otherwise define r as the distance to whatever point lies the furthest away from c_1, and apply the same methodology to that setting. This means that the selection of points in this case will yield a perimeter of $2\pi r$, at least if $k \to \infty$.

Now, suppose we add points of colour $c \neq c_1$ outside the disc of radius r in such a way that the solution selected above still is gets selected by the algorithm; it is not important specifically how this is done. Since these points all are at least r away from the c_1 point, we can provide a very optimistic bound for the optimal solution: if the points in the optimal solution are all on a straight line, the perimiter of their convex hull is at least $2r$.

Using these bounds (the optimistic $T^* \geq 2r$ and the frankly pessimistic $T \leq 2\pi r$), we can compute the approximation ratio as $\frac{T}{T^*} \leq \frac{2\pi r}{2r} = \pi$, as claimed.

*ssimon@student.chalmers.se

3 An algorithm for Vertex Cover

By definition, any edge e selected by the algorithm will be covered by S, as that set contains both its endpoints. Now consider any edge e' that wasn't selected by the algorithm. Since it wasn't selected, it must have been adjacent to one of the edges that were. Since adjacent edges share a vertex, and both the endpoints of the selected neighbour of e' are in S, one of the endpoints of e' must also be in S. This holds for all e', and as such all e' are covered by S. This, in turn, means that all edges in the graph are covered by S and as such, S is a vertex cover.

4 An approximation ratio for the Vertex Cover algorithm

Consider any edge e in the graph given by the problem. As a lower bound of the optimal solution, this edge will have at least one adjacent vertex in the covering set S, by definition of the problem. If there are n edges in the graph, this means the optimal solution will include at least n vertices in the covering set; as such, we have a lower bound n for the optimal solution T^*.

Now, consider the same edge e in the setting of our algorithm. In the worst case, this edge is one that was selected by the algorithm, and both its endpoints are in S. As an absolute worst-case upper bound for the number of vertices included in S, we can assume that *every* edge was selected by the algorithm, and that S thus contains $2n$ vertices. We can then compute the approximation ratio as

$$\frac{T}{T^*} \leq \frac{2n}{n} = 2,$$

i.e. the resulting vertex cover S is at most twice the size of an optimal vertex cover.

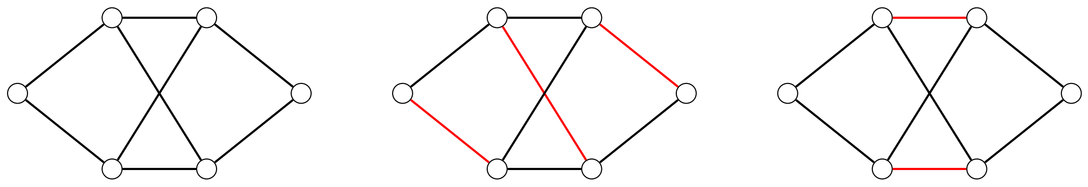
5 A matching by the Vertex Cover algorithm

As shown by Figure 1 on the following page, there is no guarantee that M is a maximum matching on the graph G. This depends on what e the algorithm chooses in every step. It is however true that M is always a *maximal* matching, i.e. that all edges $e \in G$ are either in M or share a vertex with some edge in M.

6 Reducing Weighted Hitting Set to Set Cover

Part 2 of the lecture notes provide an algorithm that solves the Weighted Set Cover problem within a factor $H(d)$ of optimality, where d is the size of the largest set. We can thus provide a reduction from Weighted Hitting Set to Weighted Set Cover that ensured $d \leq m$, and this will mean that we can solve the problem within a factor $H(m)$.

Consider the following transformation of the Weighted Set Cover problem: we construct a bipartite graph G where all sets are represented by vertices on the left,



(a) *The original graph G.* (b) *A maximum matching on G.* (c) *A possible set of edges M.*

Figure 1: *An example showing that all sets M aren't maximum matchings.*

and the universe (i.e. all elements contained in the problem) on the right, and edges between every element in the universe on the right side and the sets it's contained in on the left side. An optimal solution to Weighted Set Cover then selects vertices from the left-hand side to cover all of those on the right side, while also minimizing the total weight of the vertices selected. Here, d (the maximum size of any set) is also the number of edges adjacent to the most connected element of the left side.

We can perform a similar transformation of Weighted Hitting Set: construct a bipartite graph G where all sets are represented by vertices on the left, and the universe on the right (as with Weighted Set Cover). Weighted Hitting Set now consists of selecting elements from the *right* side that cover the left side, while minimizing the total weight of the selected vertices. Here, m is clearly the number of sets on the left side of the graph.

It can easily be seen that by simply swapping the two sides of the bipartite graph in the Weighted Hitting Set problem, we have the bipartite graph seen in the Set Cover problem. If we define d as the number of edges adjacent to any set on the left side of this flipped graph, it is also clear that solving the Set Cover problem on this graph in the manner described above provides a solution at most $H(d)$ times larger. Finally, m is the number of sets on the right-hand side of the graph, and $d \leq m$. This means we can solve the problem within a factor $H(m)$.

7 Reducing Max-cut to Min-cut

Not true. Since both the max-cut and min-cut deal with the removal of *existing* edges in the graph, this means that transforming the graph in the way described actually changes the problem fundamentally. An example of this can be seen in figure 2 on the next page, where we have a max-cut, a min-cut and the graph created by the tranformation proposed. One can easily see that the maximum cut of the transformation cannot be equivalent to the minimum cut of the graph, since they do not share any edges.

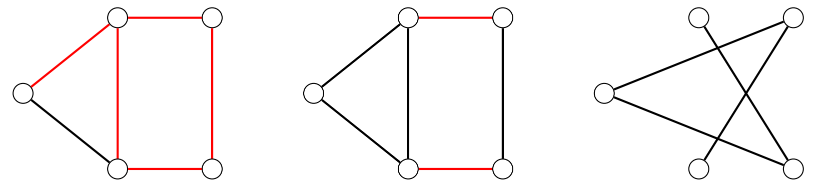

(a) *A max-cut on a graph G.* **(b)** *A min-cut on the graph G.* **(c)** *The transformation of the graph G*

Figure 2: *An example showing that the given reduction from* Min-cut *to* Max-cut *is incorrect.*

8 Testing a couple of bulbs

If the probability of a bulb being broken is p, one can easily conclude that the probability of that bulb functioning is $1-p$. We want all the bulbs of a group to be functioning, and such a group of bulbs will be lit with probability $(1-p)^m$, since there are m bulbs in the group. With k groups, we can thus expect to see $k(1-p)^m$ of the groups light up, and by extension one will see $n(1-(1-p)^m)$ bulbs that look broken.

9 A stochastic Max-cut algorithm

Consider the very simple randomized algorithm that assigns each vertex to one part with probability $\frac{1}{2}$. Now, look at a specific edge e. There is a probability $\frac{1}{2}$ that its endpoints are assigned to different parts. This goes for all edges, which means that an expected number of $\frac{m}{2}$ edges will be in the cut. This also implies that there always exists a solution with at least $\frac{m}{2}$ edges in the cut.

10 Expected time of the Max-cut algorithm

If we consider any random assignment of vertices as discussed above, the probability that $\frac{m}{2}$ or more edges are in the cut is given by $\left(\frac{1}{2}\right)^{\frac{m}{2}} = \sqrt{2}^{-m}$. This means that the expected number of assignments we have to make before finding one that has at least $\frac{m}{2}$ edges in the cut is $\sqrt{2}^m$, and as such the algorithm does not find a solution in polynomial time.

11 Arranging nodes in a directed graph

Proving that such a solution exists is fairly straight-forward. Simply consider any ordering of nodes $v_1,\ldots,v_n$. There are three possible scenarios of edge orientation:

1. Half the edges are correctly oriented, the other half is not.
2. More than $\frac{m}{2}$ edges are correctly oriented.

3. Less than $\frac{m}{2}$ edges are correctly oriented.

For cases 1 and 2, we trivially know that there exists a solution where at least $\frac{m}{2}$ edges are correctly oriented. For the third case, we can obtain such a solution by simply reversing the ordering; $v_n,\dots,v_1$ will trivially have at least $\frac{m}{2}$ edges that are correctly oriented. This implies that such a solution always exists. Note that this reasoning holds for both even and odd m.

12 Chernoff bounds on Quicksort

One can try to use Chernoff bounds on Quicksort to provide some kind information on how likely it is that the algorithm will reach worst-case performance, or deviate significantly from the average case. As stated in part 9 of the lecture notes, $O(\log n)$ subproblem types exist (i.e., there are $O(\log n)$ levels of recursion), which contributes to the overall average complexity since $O(n)$ time is spent on each type.

Chernoff bounds can give better bounds on the depth of recursion. We can consider the leaf of the longest branch in this recursion tree. Associate a random variable X_i with each split in this branch, and let it take the value 1 if the split is a "fair" split (more balanced than 25–75) and 0 otherwise. We will have a probability $p_i = \frac{1}{2}$ that $X_i = 1$, since the split will be in the middle half of the interval with that probability. To cover the worst case, we can assume that each fair split is exactly 25–75, and that the branch we're looking at got the larger part.

We need at least $\log_{\frac{3}{4}}(n)$ good splits before we reach the leaf of such a worst-case branch, and out of $d = 8\log_{\frac{3}{4}}(n)$ splits we expect $\frac{d}{2} = 4\log_{\frac{3}{4}}(n)$ splits to be good. We can now use Chernoff bounds to prove that there cannot be less than $\log_{\frac{3}{4}}(n)$ with high probability; let X be the sum of the d independent variables X_i. We then have $\mu = \mathrm{E}[X] = \frac{d}{2} = 4\log_{\frac{3}{4}}(n)$, and we set $\delta = \frac{3}{4}$:

$$\Pr\Big(X < \log_{\frac{3}{4}}(n)\Big) = \Pr\left(X < \left(1 - \frac{3}{4}\right)\mu\right) < e^{-\frac{\delta^2\mu}{2}} = e^{-\left(\frac{3}{4}\right)^2 4\log_{3/4}(n)} = n^{-\frac{9}{4\ln(4/3)}} < \frac{1}{n^3}.$$

We can thus conclude that with probability $1 - \frac{1}{n^3}$, we obtain $\log_{\frac{3}{4}}(n)$ good splits out of $d = 8\log_{\frac{3}{4}}(n)$ splits in total, which means that with probability $1 - \frac{1}{n^3}$, the recursion depth is bounded by $d = 8\log_{\frac{3}{4}}(n)$ and by extension the running time is bounded by $O(n \ln n)$.

13 An improvement of trivial 3-colouring

Consider a graph with n vertices. Such a graph has exactly $3 \cdot 2^{n-1}$ 3-colourings: if one colours any vertex of the tree with an arbitrary colour, every neighbour of a coloured vertex then has two possible colours. Since this holds for $n-1$ of the vertices, and there are three ways to colour the first vertex, there is a total of $3 \cdot 2^{n-1}$ different colourings.

We can use this to our advantage when finding a 3-colouring: simply pick an arbitrary tree that is a subgraph of G (such a tree can be found in polynomial time), and check if any of the 3-colourings of that tree is also a 3-colouring of G. This algorithm has running time $O(2^n)$, and will find a 3-colouring if one exists; any colouring of the graph must also be a colouring of the tree since removing edges only makes it easier to k-colour the graph (due to relaxation of constraints on the two endpoints of the removed edge).

14 Magnitute of the improvement in 3-colouring

If the $O(3^n)$ algorithm can solve the problem with size N, the size n of the problem that could be solved by the $O(2^n)$ algorithm will be given by $2^n = 3^N$, i.e. $n = N \log_2(3)$.

15 A Minimum Vertex Cover algorithm

As part 12 of the lecture notes describe, we can use the Nemhauser-Trotter kernelization to identify a set of $2k$ nodes that contains a vertex cover of size k. This is done by solving an LP, which can be done in polynomial time. If no such set exists, we can conclude that $2k < c$ and simply stop looking.

Once the set of nodes has been found, we can use the $O^*(1.47^k)$ algorithm described in part 11 of the lecture notes to search for a minimum vertex cover on the kernel problem. If one is found, simply return it if its size $c \leq k$ or report that no vertex cover with size $c \leq k$ exists.

16 Kernelization of Vertex Cover

The property that G' is *strictly* smaller than G is not reasonable. What would happen if we fed the output back into the algorithm as (G',k)? Other than that, it sounds almost exactly like kernelization and it's not unreasonable that FPT problems have kernelizations (in fact, they could be considered equivalent statements).