Approximation and online algorithms: CS60023: Autumn 2018

Instructor: Sudebkumar Prasant Pal
TA: Soumya Sarkar

IIT Kharagpur

email: spp@cse.iitkgp.ernet.in

January 29, 2020

- Establishing the novel lower bound

Preliminary Background

- Suppose we have an optimization problem (i.e. computing a minimum sized vertex cover or a maximum cardinality stable set) where a certain parameter must be minimized or maximised.
- Say OPT is the value of the optimal solution and we can compute a solution of value v in polynomial time using an algorithm A, then we say that $\frac{v}{OPT}$ is the *approximation ratio* for the algorithm A.
- In a minimization problem $\frac{v}{OPT} \geq 1$

Minimization problems

- OPT is generally not known but we may know a lower bound m, where $m \leq OPT$, yielding an upper bound for the approximation ratio $\frac{v}{OPT} \leq \frac{v}{m}$.
- Thus we can estimate an upper bound i.e. $\frac{v}{m}$ on the approximation ration $\frac{v}{OPT}$ for Algorithm A if we know the lower bound m; this bound is better if we have tighter value for m.
- Note that in maximization problems, we can in a similar manner define the approximation ratio as $\frac{v}{OPT} \leq 1$.
- The smaller the upper bound estimate for OPT, the better would the lower bound on the approximation ratio.

Large DAG subgraphs of directed graphs

- Let us consider the problem of computing a large directed acyclic subgraph in a given *directed graph* $G(V, E)$.
- If OPT is number of edges in the maximum size DAG, then $OPT \leq e$.
- If we partition the set E of edges into two sets so that each set induces a DAG, then we can choose the bigger one, ensuring at least $\frac{e}{2} \geq \frac{OPT}{2}$ edges are selected in the large DAG.
- Naturally each set in the partition must induce a DAG. To achieve this requirement, we use the total ordering of integers after arbitrarily numbering the vertices from 1 through $n = |V|$.

Large DAG subgraphs of directed graphs (cont.)

- Then we take the *forward edges* ($i < j$) in one set and the *backward edges* ($i > j$) in the other set. Observe that both these sets of edges constitutes DAGS; so we can pick the larger one.[1]

[1]See Problem 1.9 in page 7 in [2].

Large cuts for undirected graphs

- The size of a cut in a graph $G(V, E)$ cannot exceed $e = |E|$. If OPT is the size of the max sized cut then $OPT \leq e$
- We can start with just about any cut and then keep improving it and stop this incremental step when we cannot increase the cut size anymore.
- This incremental step could be moving one vertex across the cut only if it has a larger number of neighbours in its own current side of the cut compared to the number of neighbours on the other side of the cut.
- When we stop, we find that each vertex has more neighbours on the opposite side, that is, at least half its degree is *exhausted* across the cut.

Large cuts for undirected graphs (cont.)

- Therefore, we have total vertex degree across the cut at least half the sum of degrees of all vertices, which is at least $\frac{2e}{2} = e$.
- However, this count is just twice the number of edges across the cut as each edge counts once in the degree of its two vertices. So, we conclude that at least $\frac{e}{2}$ edges are across the cut, which is more than $\frac{OPT}{2}$.

Minimum maximal matchings

- See Problem 1.2 on page 8 in [2]
- Consider an undirected graph $G(V, E)$. Let M be a maximal matching of m edges and let OPT be the size of a maximum cardinality matching M'.
- We wish to show that $m \geq \frac{OPT}{2}$. To this effect we first observe that all the OPT edges of the maximum matching M' are incident on the $2m$ vertices of M; that is, the $2m$ vertices of M *hit* all edges in M'.
- Since no two edges of M' can be incident on the same vertex, we have at most $2m$ edges in M', that is, $OPT \leq 2m$. The approximation ratio is therefore $\frac{|M|}{|M'|} = \frac{m}{OPT} \geq \frac{m}{2m} = \frac{1}{2}$.

Vertex cover using DFS tree: Ratio factor two

- Note that internal vertices of any DFS tree of a connected undirected graph G form a vertex cover of G. Why?
- So, this vertex cover can be computed in polynomial time. See Problem 1.3 on page 8 in [2].
- If the number of internal vertices is m then we can show that there is a matching of size $\lceil \frac{m}{2} \rceil$, a lower bound on vertex cover size. Why?
- So, the size OPT of the minimum vertex cover is no lesser than this lower bound. So, $m \leq 2 \times OPT$.

Vertex cover from matching: Ratio factor two

- In the next section 7, we observe that the size of any maximal matching in an undirected graph is a lower bound for the size of the minimum vertex cover.

Vertex covering using a large cut

- The following problem is due to Vishnoi, given as Exercise 2.5 on page 23 of [2]. If we compute a large cutset H of cardinality at least half the size of the maximum cutset in an undirected graph G then $G \setminus H$ has maximum degree $\frac{\Delta}{2}$ if G has maximum degree Δ.
- Then, by induction we can show that a factor $\log \Delta$ algorithm for computing a vertex cover of G can be designed.
- The set of edges in H can be viewed as a bipartite graph on incident vertices across the large cut. We can therefore compute the exact vertex cover for this bipartite graph in polynomial time. How? (Recall the Konig-Egervary theorem for bipartite graphs.)

Vertex covering using a large cut (cont.)

- The vertex cover for $G \setminus H$ is computed recursively to within a factor $\log \frac{\Delta}{2}$ of the size of its minimum vertex cover as follows.
- So, a vertex cover for G of (i) size OPT_H for H, and (ii) (recursively) for $G \setminus H$ of size $(log \frac{\Delta}{2})OPT_{G \setminus H}$ gives a vertex cover of size at most $OPT_H + (\log \frac{\Delta}{2})OPT_{G \setminus H} \leq OPT_G + (log \Delta - 1)OPT_G = (\log \Delta)OPT_G$ for the whole of G.
- Section 2.4 Exercises 2.1, 2.2, 2.3 and 2.6 from pages 22-24 in [2]

The machines and jobs: A trivial lower bound

- We schedule n jobs in arbitrary arrival order on m identical machines.
- For the next arrival (in the arbitrarily selected sequence of arrivals of the n jobs), we assign the job to the least so far loaded machine.
- We know that the completion time or *makespan* can never be less than the processing time of the job with the largest processing time.
- So, OPT is at least $max_{i=1}^{n}\{p_i\}$, where p_i is the processing time of the job i, $1 \leq i \leq n$.

The second (non-trivial) lower bound for OPT

- Additionally, let us assume for the sake of contradiction that OPT is strictly less than the average processing time $\frac{1}{m}\sum_{i=1}^{n} p_i$.
- Then, we can complete all the n jobs in a sequential simulation on only one machine, spending a total time of $OPT \times m < \sum_{i=1}^{n} p_i$, which is less than that required to complete all the jobs sequentially on a single machine, a contradiction.
- So, we conclude that $OPT \geq \frac{1}{m}\sum_{i=1}^{n} p_i$.
- Therefore, OPT is at least the larger of $\frac{1}{m}\sum_{i=1}^{n} p_i$ and $max_{i=1}^{n}\{p_i\}$.

Upperbounding the makespan by bounding the start time of the last ending job.

- We certainly use both these lower bound for OPT in our analysis of the factor two algorithm.
- We must focus on the job p_j that ends last but may have started quite early.
- However, when p_j was scheduled on (say) machine M_i, all the other $m - 1$ machines must have been busy.
- This is due to the assignment rule that we must assign the next arrival to a machine that is least loaded at the time $start_j$ of arrival (and assignment to machine M_i).
- So, as we mentioned already, $start_j$ must be quite small, and we now argue that it is indeed not too large.

The ratio factor two bound for makespan: Bounding the start time (contd.)

- Suppose, for the sake of contradiction, we assume that $start_j$ is strictly greater than the average processing time $\frac{1}{m}\sum_{i=1}^{n} p_i$.
- So, since all the machines were busy just until $start_j$, and therefore fully utilized, we have $m \times start_j > \sum_{i=1}^{n} p_i$, and so a simulation on a single machine would complete all jobs, a contradiction because we have a p_j-sized job yet to be processed, a contradiction.
- Therefore, $start_j \leq \frac{1}{m}\sum_{i=1}^{n} p_i \leq OPT$.
- We have $p_j \leq OPT$ as well.
- Therefore, the makespan computed by Algorithm 10.2 in [2] is at most $2.OPT$. Why?

The “local search” offline version of makespan

- Assume that we have all the n jobs’ data p_j, and that we have already made an arbitrary assignment of the jobs to the m processors.
- We can improve by reassigning jobs to processors carefully.
- One such reassignment strategy is to reassign the currently last ending job b to a machine M if that helps finish job b earlier than time $C_b = start_b + p_b$.
- This improvement is therefore possible if the “total duration” of the machine M in the current schedule is smaller than $start_b = C_b - p_b$.

Offline makespan (contd.)

- The two lower bounds for OPT hold just as they hold for the online version or the incremental version.
- Also, if no improvement is possible, then we take the current $C_b = start_b + p_b$ as our upper estimate for OPT, yielding again the same upper bound of at most $2.OPT$ for C_b.
- How can we improve this approximation ratio to $2 - \frac{1}{m}$?

The notion of prices in the primal integral solution

- Observe that using the linear programming relaxation of the integer program and the dual LP of the (primal) LP relaxation, we saw how the (primal) integral solution computed by the algorithm is *fully paid for* by the computed dual variables.
- The objective function value of the primal integral solution is matched by the objective function value of the dual variables computed.
- However, further in the analysis, we divide the dual variables by a suitable factor and show that the scaled down dual solution is feasible.

The dual solution objective function value as a lower bound

- The scaling factor is the approximation guarantee of the algorithm since the dual gives a lower bound on the optimal value of the linear primal and dual linear programs, thereby giving a lower bound on also the optimal objective function value of the integer linear program.
- Indeed, the greedy algorithm defines dual variable values $price(e)$, for each element e. Observe that the cost of the selected sets in the set cover picked by the algorithm is *fully paid* for (in this case exactly equalled) by the dual solution.
- However, this dual solution is not feasible. We therefore needed to shrink the values by a factor of $H(n)$, so that no set is *overpacked* (all constraints of the dual LP are satisfied).

- As in Section 15.1 of [2], we focus on the standard form primal and dual LPs, and define *relaxed complementary slackness* conditions with parameters α and β, leading to the crucial Proposition 15.1 of [2].
- Primal complementary slackness conditions: Let $\alpha \geq 1$. For each $1 \leq j \leq n$: either $x_j = 0$ or $\frac{c_j}{\alpha} \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j$.
- Dual complementary slackness conditions: Let $\beta \geq 1$. For each $1 \leq i \leq m$: either $y_i = 0$ or $b_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq \beta b_i$.
- The design of Algorithm 15.2 is based on Proposition 15.1 leading to Theorem 15.3.
- Proposition 15.1: If x and y are primal and dual feasible solutions satisfying the conditions stated above then $\sum_{j=1}^{n} c_j x_j \leq \alpha\beta \sum_{i=1}^{m} b_i y_i$.
- Algorithm 15.2 starts with a primal *infeasible* solution and a dual feasible solution; these are usually the trivial solutions $x = 0$ and $y = 0$.

- It iteratively *improves the feasibility of the primal solution*, and the *optimality of the dual solution*, ensuring that in the end *a primal feasible* solution is obtained and all conditions, with a suitable choice of α and β, are satisfied.
- The primal solution is *always extended integrally*, thus ensuring that the final solution is integral.
- The current primal solution is used to determine the improvement to the dual, and vice versa.
- Finally, the cost of the dual solution is used as a lower bound on OPT, and by Proposition 15.1, the approximation guarantee of the algorithm is $\alpha\beta$.
- For exercise 12.4 in the print version of [2] (absent in the e-version), we use the paying mechanism for showing the equality of the objective function values for the primal and dual LPs provided the solutions for the primal and dual LPs obey the complememtary slackness conditions.

- To prove Proposition 15.1, we show that given a sufficient amount of balance, the dual can pay enough from this balance through dual variables y_i for the primal variabes x_j, so that the total payment done by the y_i's, and collected by the x_j's, is sufficient to meet the objective function cost of the primal solution.
- The upper bound (balance) for the total payment by the y_i's is the r.h.s. of Proposition 15.1 and the collection made by the primal x_j's is at least the lower bound l.h.s. of Proposition 15.1. The details as are follows.
- The payment from y_i to x_j is $\alpha y_i a_{ij} x_j$.
- The total payment to all the primal variables from y_i is therefore $\alpha y_i \sum_{j=1}^{n} a_{ij} x_j \leq \alpha\beta b_i y_i$ (due to the upper bounds in the relaxed dual complementary slackness conditions).
- So, the total payment from all dual variables to all primal variables is at most the balance r.h.s. $\alpha\beta \sum_{i=1}^{m} b_i y_i$ of Proposition 15.1.

- The total collection in x_j is $\alpha x_j \sum_{i=1}^{m} a_{ij} y_i \geq c_j x_j$ (due to the lower bounds in the relaxed primal complementary slackness conditions.)
- So, the total collection at all the primal variables is at least the l.h.s $\sum_{i=1}^{n} c_j x_j$ of Proposition 15.1.
- Note that the total amount $\alpha y^T A x$ sent by the dual and the total amount $\alpha x^T A^T y$ received by the primal is the same quantity.
- The interesting example when $\alpha = 1$ and $\beta = f$ is that of the weighted set cover problem where each element is in at most f sets.
- We are guaranteed an f-factor algorithm if the post-condition satisfied by the execution of the algorithm satisfies the relaxed complementary slackness conditions, due to Proposition 15.1.
- The algorithm picks sets in the set cover one by one by satisfying the equality constraint due to the unit value of α in the dual inequalities.

The approximation algorithm with ratio factor two for vertex covering

- For an undirected *simple* graph $G(V, E)$, a set $W \subseteq V$ is a *vertex cover* if, for every edge $\{u, v\} \in E$, either u or v or both are in W.
- We wish to find a *minimum cardinality vertex cover* for the graph $G(V, E)$.
- This being an NP-hard problem, it is worthwhile searching for polynomial time approximation algorithms.
- We can try several heuristics. We may select (and delete) an arbitrary vertex $v \in V$ for inclusion in vertex cover C and drop all edges incident on v.

The approximation algorithm with ratio factor two for vertex covering (cont.)

- This step can be repeated until the graph becomes empty (of edges).
- Alternatively, we may use another rule, where we select an arbitrary edge $\{u, v\} \in E$ and include both u and v in C; we drop all edges incident on u and v and repeat the process until the graph becomes empty.
- For a *straight line graph* (that is, a simple path of n vertices and $n-1$ edges), the first method finds a vertex cover of size $n-1$, which is within twice the size of the optimal vertex cover of size $\lfloor n/2 \rfloor$. Why?
- Does it work well also for other classes of graphs like trees, planar graphs, and general graphs?

The approximation algorithm with ratio factor two for vertex covering (cont.)

- The second one chooses both vertices of all edges in a *maximal matching* S to be included in the computed vertex cover C.
- *A matching is a set* S *of edges where no two edges in* M *share any vertex. A matching* S *is called a maximal matching if we cannot add an additional edge to* M *to get a larger matching.*
- We analyze the second heuristic, following the exposition in [1].
- How well does this second heuristic work for straight line graphs?
- If C^* is any minimum vertex cover then $|S| \leq |C^*|$, where S is any maximal matching. Why?

The approximation algorithm with ratio factor two for vertex covering (cont.)

- Vertices in C^* have to cover each edge in the *(maximal) matching* S. So, C^* must include at least one vertex from each of the S edges.
- The $2|S|$ vertices comprise the computed approximate vertex cover C. So, $|C| = 2|S| \leq 2|C^*|$, since $|S| \leq |C^*|$.
- This gives a polynomial time algorithm yielding a vertex cover that is certainly at most twice the size of the minimum vertex cover.
- It is interesting to note that any matching in a graph would force at least as many vertices in the vertex cover as twice the number of edges in the matching.

The approximation algorithm with ratio factor two for vertex covering (cont.)

- So, the cardinality of the maximum matching gives a lower bound on the cardinality of any vertex cover. Do these two cardinalities ever coincide for any classes of graphs?
- From the algorithmic angle, we have already noted that the vertices of edges forming a *maximal* matching cover all edges, and a maximal matching can be computed using the *greedy* approach as in the second heuristic stated above (see [2]).
- As mentioned earlier, the *maximal matching* is such that none of its supersets enjoys the same property.
- So, observe that a vertex cover generated by our approximation algorithm might as well be smaller than twice the cardinality of the *maximum matching*.

The approximation algorithm with ratio factor two for vertex covering (cont.)

- In such cases, where the discovered maximal matching is smaller than the maximum matching, we can indeed have some savings.

Improvement of the approximation guarantee

- A natural question about improving the approximation guarantee is whether a better analysis of the algorithm being considered, can improve the approximation guarantee any further. Essentially, we must show that the analysis already provided for the algorithm is tight.

Tight example for the vertex cover algorithm

- In the case of the factor two algorithm using maximal matchings for vertex covering (in Section 7), we note that on $K_{n,n}$ the algorithm produces a solution that is twice the optimal in cardinality.
- Since $K_{n,n}$ is the complete bipartite graph on $2n$ vertices with n^2 edges, our algorithm would certainly choose a matching of size n, and therefore a vertex cover of size $2n$, thereby showing that we cannot get a factor better than 2 for such graphs for any integer n (see [2]), for this algorithm.
- This is despite the fact that the lower bound of the size of a maximal matching is n as well as the size of an optimal vertex cover is n.

Tight example for the vertex cover algorithm (cont.)

- So, this family of infinite graphs provides what we call a *tight (asymptotic) example* for the specific algorithm. Tight examples often give critical insights into the functioning of an algorithm and often lead to ideas for the design of other algorithms that can achieve improved guarantees.
- However, we do have vertex covers of size n in $K_{n,n}$!!! A smarter algorithm might be able to tackle special cases where smaller than $2n$-sized vertex covers can be discovered.

Maximal matchings lower bound cannot yield better approximation guarantees for vertex covering

- Now consider another question: can a better approximation algorithm be designed that achieves a better guarantee but still uses the the same lower bounding scheme as our current algorithm of Section 7. For addressing this second question, consider the complete graph K_n of n vertices where n is an odd integer.
- Note that it has a minimum vertex cover of size $n-1$; dropping any two vertices would leave an edge uncovered. Also, n being odd, we observe the maximum matching has cardinality $\frac{n-1}{2}$. For this example, no algorithm can achieve a ratio factor of approximation better than 2 for any odd integer n (see [2]).

Maximal matchings lower bound cannot yield better approximation guarantees for vertex covering (cont.)

- So, we observe that by simply using the lower bounding scheme of maximal matchings, we cannot improve the approximation ratio.

Total weight of all edges cannot yield better approximation guarantees for weighted cut

- As in the case of vertex cover in Section 7, we can also make a similar observation for the *maximum weighted cut* problem.
- A polynomial time algorithm exists that ensures a cut of weighted capacity at least $\frac{1}{2}w(E)$, where $w(E)$ is the sum of weights of the edges. We now show here that for all n, we cannot have a better ratio factor for graphs K_{2n}, if we use the (obvious) upper bound of $w(E)$ for the maximum cut. The graph K_{2n} has exactly $n(2n-1)$ edges and a maximum cut of size n^2, giving an approximation ratio at most $\frac{1}{2}$ for such graphs for all integers n.

Total weight of all edges cannot yield better approximation guarantees for weighted cut (cont.)

- Observe that the cut is maximised when it separates any set of n vertices form the rest of the n vertices *So, we observe that by simply using the upper bounding scheme provided by* $w(E)$, *we cannot improve the approximation ratio.*

Tight example for the greedy weighted set cover algorithm

- Suppose, n sets each have a singleton element and the set weights are respectively, $\frac{1}{n}, \frac{1}{n-1}, \cdots, 1$, and the last set has all these n elements with set weight $1 + \epsilon$.
- The optimal cover has weight $1 + \epsilon$ but the greedy algorithm computes a set cover with weight $H(n)$; in each iteration, the cost effectiveness of the last set is higher than those of the previous sets and lower than those of the sets not yet selected. This is an example where the $H(n)$ upper bound is approached as ϵ approaches zero.
- In Section 13.1 of [2] we can see the Example 13.4 which reveals that the H_n bound is essentially tight, irrespective of the algorithm used. For this purpose, we must see the LP relaxation 13.2 on page 109 of [2].

Tight example for the greedy weighted set cover algorithm (cont.)

- See Sections 29.7 and 29.9 of [2] for seeing why the obvious greedy algorithm is the best one can hope for.

- Observe that using the linear programming relaxation of the integer program and the dual LP of the (primal) LP relaxation, we saw how the (primal) integral solution computed by the algorithm is *fully paid for* by the computed dual variables. The objective function value of the primal integral solution is matched by the objective function value of the dual variables computed. However, further in the analysis, we divide the dual variables by a suitable factor and show that the scaled down dual solution is feasible.
- The scaling factor is the approximation guarantee of the algorithm since the dual gives a lower bound on the optimal value of the linear primal and dual linear programs, thereby giving a lower bound on also the optimal objective function value of the integer linear program.

- Indeed, the greedy algorithm defines dual variable values $price(e)$, for each element e. Observe that the cost of the selected sets in the set cover picked by the algorithm is *fully payed* for (in this case exactly equalled) by the dual solution. However, this dual solution is not feasible. We therefore needed to shrink the values by a factor of $H(n)$, so that they fit into the given set cover instance, i.e., no set is *overpacked* (in other words, all constraints of the dual LP are satisfied).

Dual fitting for the constrained set multicover problem

- The discussion here on the *constrained set multicover* problem is from Section 13.2.1 in [2]. Here, each element e needs to be covered a specific integer number r_e of times. We also use the constraint that each set can be picked up at most once. A set S if picked up k times yields cost $k \times c(S)$. Such permissible picking of a set multiple times is allowed in the less constrained problem *set multicover*.

Integer program and the primal relaxation LP for the constrained problem

- Now we propose the integer programming formulation as min $\sum_{S \in \mathcal{S}} c(S) x_S$ subject to $\sum_{S : e \in S} x_S \geq r_e$, for all $e \in U$, given that $x_S \in \{0, 1\}$, for all $S \in \mathcal{S}$. Here, $r_e \in Z^+$.
- The LP-relaxation is tricky because we must now constrain each set to be selected at most once. So, we need to realize the constraint $x_S \leq 1$ as well.
- Therefore, replacing the integer program constraint on x_S taking on values 0 and 1 only, we now use the following constraints in the LP-relaxation: $-x_S \geq -1$ and $x_S \geq 0$, for all $S \in \mathcal{S}$.

The dual LP for the primal LP relaxation of the integer program

- The dual linear program for the LP-relaxation is therefore complex, with a few more variables because we do not have what we call *a primal covering linear program*.
- There are some negative elements in the matrices and vectors in the primal-dual linear program formulation).
- The additional constraints have new variables z_S in the dual. The dual LP is no more a packing program.
- The primal LP has one constraint for each element in U as well as a constraint for each set in $\mathcal{S}$; there are as many variables in the dual LP, the y_e variables as well as the z_S variables.

The dual LP for the primal LP relaxation of the integer program (cont.)

- Now, a set S can be overpacked with the y_e variables.
- This can be done only provided we raise z_S to ensure feasibility.
- The objective function value can then decrease. However, overall, overpacking may still be advantageous, since the y_e appear with coefficients of r_e in the objective function.
- max $\sum_{e \in U} r_e y_e - \sum_{S \in \mathcal{S}} z_S$
  subject to
  $(\sum_{e: e \in S} y_e) - z_S \leq c(S)$, for all $S \in S$
  $y_e \geq 0$, for all $e \in U$
  $z_S \geq 0$, for all $S \in \mathcal{S}$

- The greedy algorithm is as follows. We say that element e is *alive* if it occurs in less than r_e of the sets already selected.
- The algorithm picks a hitherto unpicked set which is the most *cost-effective set*; the *cost-effectiveness* of a set is defined as the average cost at which the set covers its currently alive elements.
- The algorithm halts when there are no more alive elements.

- On picking a set S, its cost $c(S)$ is distributed equally amongst the alive elements it covers. If S covers e for the jth time, $price(e, j)$ is set to the current cost-effectiveness of S as defined above.
- It is easy to see that the cost-effectiveness of sets picked is non-decreasing.
- Since cost-effectiveness is non-decreasing over iterations of selection of sets in the set cover, we have, for each element e, $price(e, 1) \leq price(e, 2) ... \leq price(e, r_e)$.

The dual solution

- The variables of the dual are set as follows at the end of the algorithm's execution.
- For each $e \in U$, we set (after scaling by H_n) $y_e = \frac{\alpha_e}{H_n} = \frac{1}{H_n}.price(e, r_e)$.
- For each $S \in \mathcal{S}$ picked up by the algorithm in the set cover, we set (after scaling down by H_n) $z_S = \frac{\beta_S}{H_n} = \frac{1}{H_n}.[\sum_{e-covered-by-S}(price(e, r_e) - price(e, j_e))]$, where j_e is the copy of e covered by S.
- Note that since $price(e, j_e) \leq price(e, r_e)$, so β_S is non-negative.
- If S is not picked by the algorithm, then β_S is defined to be 0.

The dual solution (cont.)

- Now observe that the objective value of the primal is $\sum_{e \in U} \sum_{j=1}^{r_e} price(e,j)$. Indeed, this is identical to the objective function value of the dual variables (α, β) since $\sum_{e \in U} r_e \alpha_e - \sum_{S \in \mathcal{S}} \beta_S = \sum_{e \in U} \sum_{j=1}^{r_e} price(e,j)$.
- After scaling down (α, β) by a factor of H_n we get the *scaled* dual LP feasible solution (y, z), where $y_e = \frac{\alpha_e}{H_n}$ and $z_S = \frac{\beta_S}{H_n}$.

Scaling and dual-fitting for satisfying the dual constraints

- For ascertaining that the (y, z) solution is a scaled but feasible dual solution, we need to look at each set S.
- Consider a set $S \in \mathcal{S}$ consisting of k elements. Order and enumerate its elements in the order in which their multiple occurance requirements were fulfilled.
- This is the order in which they stopped being alive.
- Let the ordered elements be $e_1, ..., e_k$.
- Suppose S is not picked by the algorithm. When the algorithm is about to cover the last copy of e_i, S contains at least $k - i + 1$ alive elements, so $price(e_i, r_{e_i}) \leq \frac{c(S)}{k-i+1}$.
- Since z_S is zero, we get $\sum_{i=1}^{k} y_{e_i} - z_S = \frac{1}{H_n} \sum_{i=1}^{k} price(e_i, r_{e_i})$ $\leq \frac{c(S)}{H_n}.(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{1}) \leq c(S)$.

Scaling and dual-fitting for satisfying the dual constraints (cont.)

- Next, we assume that S is picked by the algorithm.
- Also assume that just before S is picked up, $k' \geq 0$ elements of S are already completely covered.
- Then, $(\sum_{i=1}^{k} y_{e_i}) - z_S =$ $\frac{1}{H_n}[\sum_{i=1}^{k} price(e_i, r_{e_i}) - \sum_{i=k'+1}^{k}(price(e_i, r_{e_i}) - price(e_i, j_i))] =$ $\frac{1}{H_n}[\sum_{i=1}^{k'} price(e_i, r_{e_i}) + \sum_{i=k'+1}^{k} price(e_i, j_i)]$, where S covers the j_ith copy of e_i, for each $i \in \{k'+1, \ldots, k\}$.
- But $\sum_{i=k'+1}^{k} price(e_i, j_i) = c(S)$, since the cost of S is equally distributed among the copies it covers.
- Finally consider elements e_i, $i \in \{1, \ldots, k'\}$.

Scaling and dual-fitting for satisfying the dual constraints (cont.)

- When the last copy of e_i is being covered, S is not yet picked and covers at least $k - i + 1$ alive elements.
- Thus, $price(e_i, r_{e_i}) \leq \frac{c(S)}{k-i+1}$.
- Therefore, $(\sum_{i=1}^{k} y_{e_i}) - z_S \leq \frac{c(S)}{H_n}(\frac{1}{k} + \cdots + \frac{1}{k-k'+1} + 1) \leq c(S))$.

The final analysis of the factor H_k ratio bound

- The actual aprroximation ratio is as good as H_k, where k is the cardinaity of the largest set in $\mathcal{S}$. This fact is easily seen in the derivations above.

The k-centre problem

- The k-center problem is formally stated as follows. Let $G = (V, E)$ be a complete graph having a non-negative cost d_{ij} associated with each edge (v_i, v_j) of E. We assume that for every triple of vertices v_i, v_j , $v_l \in V$, the distances satisfy the triangle inequality, i.e., $d_{ij} \leq d_{il} + d_{lj}$.
- Given a positive integer k, (i) chose a set (called cluster centers) $S \subseteq V$ of $|S| = k$, and (ii) assign each of the remaining vertices $V \subseteq S$ to its nearest cluster center. The objective is to minimize the maximum distance of a vertex to its cluster center.
- Geometrically, the goal is to find k different *balls* covering all points so that the radius of the largest ball is as small as possible.

The k-centre problem (cont.)

- In other words, the goal is to find a set S of the centers of k different balls of the same radius that cover all points in $V \setminus S$ so that the radius is as small as possible.
- First, we define the distance of a vertex i from a set $S \subseteq V$ of vertices to be $d(i, S) = min_{j \in S} d_{ij}$. Then the corresponding radius for S is equal to $max_{i \in V} d(i, S)$, and the goal of the k-center problem is to find a set of size k of minimum radius.
- Again in this problem, we will use the triangle inequality. The n points of a set V of points with pairwise distances obeying the triangle inequality are given. We study the specific *clustering problem* of choosing a set S of k out of n points as *centres of clusters*, so that points closer to a centre in S than any other centres in S are grouped into a cluster.

The k-centre problem (cont.)

- The *cluster radius* is the radius of the smallest *ball* (circle) centred at each *cluster centre* and enclosing all points of that cluster. The maximum of the k cluster radii has to be minimised. This is an NP-hard problem; we present a factor two approximation algorithm as given in [3]
- The approximation algorithm is simple: it selects an arbitrary point initially as one cluster centre in the set S of cluster centres. Then, it repeatedly chooses cluster centres for newer clusters till all k centres are selected in S.
- Every subsequent cluster centre is chosen by selecting a point $i \in V \setminus S$ whose distance $d(i, S)$ to the points in S is maximized.

The k-centre problem (cont.)

- For proving the factor two approximation bound, we again choose an arbitrary optimal solution S^* with r denoting the radius of the largest cluster in the optimal solution S^*.
- Due to triangle inequality, the distance between any two vertices within any cluster of the optimal solution S^* is bounded by $2r$. The solution S of k cluster centres, as identified by our approximation algorithm may be different from S^*.
- Assume that the algorithm has chosen only one vertex $u(B) \in S$ from a cluster B with centre $v(B) \in S^*$ of the optimal solution S^*.
- Furthermore, suppose only one vertex is selected in S from each cluster of the optimal solution S^*.

The k-centre problem (cont.)

- Then vertices of S^* are in any case within a distance r of some cluster centre in S ! Why?
- Now consider any vertex $u \in V \setminus S^*$.
- Any $u \in V \setminus S^*$, within the radius r cluster B centred at its cluster centre vertex $v(B) \in S^*$, is within a distance of $2r$ from the cluster centre vertex $u(B) \in S$, due to triangle inequality.
- Now consider the other situation where one vertex $u(B)$ inside the cluster B is already selected in S and the algorithm still chooses another vertex $w(B)$ inside the cluster B as a center point in S.

The k-centre problem (cont.)

- Again, the distance between $u(B)$ and $w(B)$ is bounded by $2r$. Moreover, $w(B)$ must have been the furthest point from all points in S at the time it was selected in S, including $u(B)$, by the choice of the algorithm, and therefore, all the points are within a distance of $2r$ of some center point already selected in S.
- This argument holds even if the algorithm adds more points of B to S subsequently.

Multiway cut

- Given a set $S = \{s_1, s_2, ..., s_k\}$ of *terminals* where $S \subseteq V$, a multiway cut is a set of edges whose removal disconnects the terminals from each other.
- The multiway cut problem asks for such a minimum weight cut. This presentation is from Section 4.1 of [2]. The problem of finding a minimum weight multiway cut is NP-hard for any fixed $k \geq 3$.
- Observe that the case $k = 2$ is precisely the minimum (s, t)-cut problem, solvable in polynomial time using network flows. We present a $2 - \frac{2}{k}$ approximation algorithm for this problem as follows for $k \geq 3$.
- For each $i = 1, ..., k$ do
  1. identify the terminals in $S \setminus \{s_i\}$ into a single vertex

Multiway cut (cont.)

2. compute a minimum weight cut C_i for $(s_i, S - \{s_i\})$ using a network flow algorithm, and
3. discard the heaviest of these cuts. The output is the union of the rest, say C.

- Let A be an optimal multiway cut in G. A can be viewed as the union of k cuts as follows. The removal of A from G creates k connected components, each having one terminal.
- Let $A_i \subseteq A$ be the cut separating the component containing s_i from the rest of the graph.
- So, $A = \cup_{i=1}^{k} A_i$. Since each edge of A is incident at two of these components, each edge belongs to two of the cuts. So, $\sum_{i=1}^{k} w(A_i) = 2w(A)$.

Computational lower bounds on the sizes of cuts in the optimal solution

- Now the main lower bound argument is that C_i being a minimum weight cut for s_i, we have $w(C_i) \leq w(A_i)$.
- A similar lower bound argument is used also in the much more complex proof of an approximation bound for the minimum weight k-cut problem in Section 4.2 of [2].
- Note that this already gives a 2-approximation algorithm, by taking the union of all k cuts C_i.
- This union step in the alorithm is reminiscent of the vertex cover algorithm where for each matching edge we include vertices at both ends of the edge.

Computational lower bounds on the sizes of cuts in the optimal solution (cont.)

- Finally, since C is obtained by discarding the heaviest of the cuts C_i, we have $w(C) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} w(C_i) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} w(A_i) = 2(1 - \frac{1}{k}) w(A)$.

The k-cut problem

- The k-cut problem is similar to the multiway cut problem but in this case we do not provide any set of k terminals. This exposition is based on Section 4.2 of [2]
- This k-cut problem is a more general problem. The nice approximation bound in this problem requires a complicated analysis using Gomory-Hu trees.
- A k-cut is a set of edges whose removal leaves k connected components for a connected graph. For positive edge weights, we wish to find a minimum weighted k-cut. We will address the well-known result about the factor $2 - \frac{2}{k}$ approximation algorithm.

The Gomory-Hu tree and minimum weight cuts

- We use the Gomory-Hu tree T defined on the same vertex set V as that of the graph $G(V, E)$ with positive edge weights for edges in E. The edges of T may not belong to E.
- Suppose the removal of an edge e_T of T gives two components of the vertex set namely, S and $V \setminus S$.
- Let $E' \subseteq E$ be the edges of G whose removal from G partitions vertex set of G into S and $V \setminus S$.
- In other words, E' is the cut-set for S and $V \setminus S$ in G. Assign the sum of weights on edges of E' on edge e_T of T. So, edges of T have weights corresponding to the minimum weight cut-sets in G.

The Gomory-Hu tree and minimum weight cuts (cont.)

- Thus, out of $\binom{n}{2}$ minimum weight $u - v$ cuts, only $n - 1$ minimum weight cut sets in G are used as weights on edges of T.
- Thus, the min-cut tree T (called the Gomory-Hu Tree) has the property that the minimum cut between any two nodes v_i and v_j in G is the smallest weight edge in the unique path that connects v_i and v_j in T.

Properties of any optimal k-cut A and the approximation algorithm for computing a k-cut

- Let S be the union of minimum weights cuts in G associated with l edges of T. Then, the removal of S from G leaves a graph with at least $l + 1$ components.
- The k-cut approximation algorithm we analyze is simple; the algorithm first constructs the Gomory-Hu tree T for G in polynomial time, and then constructs a k-cut set C by taking the union of cut edges of G corresponding to the lightest $k - 1$ edges in T.
- If more than k connected components result then we keep throwing back cut edges till there are exactly k components.

Properties of any optimal k-cut A and the approximation algorithm for computing a k-cut (cont.)

- Let A be an optimal k-cut in G, which can be viewed as the union of k cuts. Let the removal of A from G create k connected components, $V_1, V_2, \ldots, V_k$.
- Let $A_i \subseteq A$ be the cut separating V_i from the rest of the graph. Then, $A = \cup_{i=1}^{k} A_i$. Each edge of A is incident at two of these components. So, each edge of A is in two of the cuts. So, $\sum_{i=1}^{k} w(A_i) = 2w(A)$.

Establishing the novel lower bound

- Now we have to connect the properties of the output C of the approximation algorithm with the properties of the arbitrary minimum k-cut A, in order to establish the factor $2 - \frac{2}{k}$ ratio bound.
- The main idea is to *identify* (show the existence of) $k - 1$ cuts defined by the edges of T whose weights are *dominated* by the weight of the cuts $A_1, A_2, \ldots A_{k-1}$ of the optimal k-cut A. This lower bound argument is crucial. These $k - 1$ cuts are identified as follows.
- Let B be the set of edges of T that connect across two of the sets $V_1, V_2, \ldots, V_k$. Consider the graph on the vertex set V and the edge set B. We shrink each of the sets $V_1, V_2, \ldots, V_k$ to resective k single super-vertices.

Establishing the novel lower bound (cont.)

- So, we have essentially superimposed the tree T over the k connected components of an optimal k-cut A giving a possibly non-tree but connected graph with edge set B on k super-vertices.
- Observe that this graph must be connected since T is itself connected. Throw edges away from this graph until a tree T' survives. Let $B' \subseteq B$ be the leftover edges in T'. Clearly, $|B'| = k - 1$ as T' is a tree of k vertices.
- The edges of B' define the required $k - 1$ cuts which are dominated by $k - 1$ cuts from A.

Establishing the novel lower bound (cont.)

- Assuming that A_k is the heaviest cut amongst the cuts of A. Imagine rooting tree T' at V_k. We now define a correspondence between the edges in B and the sets $V_1, V_2, ..., V_{k-1}$: each edge corresponding to the set it comes out of in the rooted tree, going towards the parent.
- Suppose edge $(u, v) \in B'$ corresponds to a set V_i in this manner where V_j is the parent of V_i in T', $u \in V_j$ and $v \in V_i$. The weight of a minimum $u - v$ cut in G is $w'(u, v)$.
- Since A_i is also a $u - v$ cut in G (but may not be the minimum such cut!), we therefore have $w(A_i) \geq w'(u, v)$ for all i, $1 \leq i \leq k - 1$.

Establishing the novel lower bound (cont.)

- Since, the union of the lightest $k-1$ cuts defined by T is C in our approximation algorithm, we have $w(C) \leq \sum_{e \in B'} w'(e) \leq \sum_{i=1}^{k-1} w(A_i) \leq \sum_{i=1}^{k} (1 - \frac{1}{k}) A_i = 2(1 - \frac{1}{k}) w(A)$.

The K-server problem

- Online algorithms for the K-server problem are considered.
- K servers need to be moved around to service requests appearing online at points of a metric space.
- The total distance travelled by the K servers must be minimised, where any request arising at a point of the metric space must be serviced on site by moving a server to that site.
- $d(a_1, a_2)$ is defined as the distance between a_1 and a_2.
- M represents the metric space where d is the metric which satisfies the triangle inequality.
- M^K represents the set of configurations of the K points of M.
- Given configurations C_1 and C_2, $d(C_1, C_2)$ is the minimum possible distance travelled by K servers that change configuration from C_1 to C_2.

- $C_0 \in M^K$ is the initial configuration.
- Let $r = (r_1, r_2, ...r_m)$ be the sequence of request points in M.
- The solution $C_1, C_2, ..., C_m \in M^K$ is such that $r_t \in C_t, \forall t = 1....m$.
- Serving $r_1, r_2, ...r_m$ by moving through $C_1, C_2, ..., C_m$ entails solution cost $\sum_{t=1}^{m} d(C_{t-1}, C_t)$.
- The online algorithm uses only $r_1, r_2...r_t$ and $C_0, .., C_{t-1}$ to compute C_t.
- The offline algorithm uses also $r_{t+1}, r_{t+2}...r_m$.

- Given C_0, $r = (r_1, r_2, ... r_m)$, $cost_A(C_0, r)$ is the cost of the online algorithm A, and
- $opt(C_0, r)$ is the cost of the optimal algorithm.
- ρ is the competitive ratio.
- Competitive ratio is used as $cost_A(C_0, r) < \rho * opt(C_0, r) + \phi(C_0)$ for some ρ.
- $\phi(C_0)$ is independent of r.
- ρ_M may be used for metric space M.
- Conjecture: For every metric space with more than K distinct points the competetive ratio for the K-server problem is exactly K.
  $\rho = \inf_A \sup_r \frac{cost_A(C_0,r)}{opt(C_0,r)}$, modulo a constant term.

Theorem

In every metric space with at least $K + 1$ points, no online algorithm for the K-server problem can have competitive ratio less than K.

- We wish to show there are request sequences of arbitrary high cost for A for which the online algorithm A has cost K times that of the optimal offline algorithm. We prove this lower bound result later below.
- Now we consider the *double coverage* strategy, used for the online algorithm A to achieve the competitive ratio K.
- Let a_1, a_2, a_3 be ordered left to right on a horizontal line with a_2 closer to a_1 than a_2. Let servers s_1 and s_2 be at a_1 and a_3 respectively, initially, with no server at a_2.

- Let serving requests come repeatedly alternating between a_2 and a_1.
- If we move only the (closest) server s_1 (which was intitally stationed at a_1) up and down between a_1 and a_2 for the sequence $a_2, a_1, a_2, a_1, \cdots$, we incur unbounded competive ratio for asympopically large strings of requests a_2, a_1.
- This is so because the offline algorithm would place servers s_1 and s_2 at a_1 and a_2 respectively, permanently, instead of fixing s_2 at a_3.
- In the *double coverage* strategy instead, we move both s_1 and s_2 towards a_2 by amount $d(a_1, a_2)$ on serving request a_2, and then move s_1 back to a_1 on serving request a_1. So we use travel cost at most 3 times of that used by the optimal offline algorithm, which moves s_2 only once to a_2 on the first serving request for a_2.

- We continue to analyse the double coverage strategy whose suggested ratio is 3 for $K = 2$ servers.
- Note that consecutive configurations C_t, C_{t-1} differ only in r_t i.e., $C_t = C_{t-1} \cup r_t$.

- The scenario where servers are moved only to service requests directly is called *lazy*. The double coverage algorithm in that sense is not lazy.
- A non-lazy algorithm can however be *memory-less* like the double coverage algorithm since decisions are based only on the current configuration.
- Let us use potential $\Phi(C_t, C'_t)$ where C stands for the online algorithm and C' for the offline algorithm.
- Let $cost(t)$ and $opt(t)$ be the costs to service r_t by online and offline methods at the instant t.

- We need to show that

$$cost(t) - K * opt(t) \leq \Phi(C_{t-1}, C'_{t-1}) - \Phi(C_t, C'_t) \quad (1)$$

- Adding for m steps we have

$$\sum_{t=1}^{m} cost(t) - K * \sum_{t=1}^{m} opt(t) \leq \Phi(C_0, C'_0) - \Phi(C_m, C'_m) \quad (2)$$

- We can drop $\Phi(C_m, C'_m)$ without disturbing upper bounding so that we have

$$\sum_{t=1}^{m} cost(t) - K * \sum_{t=1}^{m} opt(t) \leq \Phi(C_0, C'_0) \quad (3)$$

which gives the competitive ratio of at most K.

- Let us use the offline algorithm to respond to r_t first and then the online algorithm.
    1. $C'_{t-1} \implies C'_t$, whereas C_{t-1} is unchanged.
    2. $C_{t-1} \implies C_t$ where C'_t has already reached a server to location r_t.

  We define the potential function as

$$\Phi(C_t, C'_t) = K * d(C_t, C'_t) + \sum_{a_i, a_j \in C_t} d(a_i, a_j) \tag{4}$$

- $d(C_t, C'_t) \implies$ weight of the minimum weight bipartite matching in K_{C_t, C'_t}, the complete bipartite graph where servers of the offline and online algorithm form the two vertex sets C_t and C'_t.

- To prove inequality (1) we do the two transitions of [1], the offline algorithm, and then [2], the online algorithm. $cost(t) - K * opt(t) \leq \Phi(C_{t-1}, C'_{t-1}) - \Phi(C_t, C'_t)$
- Wherever $cost(t)$ is more than $K * opt(t)$, there is a *balancing payment* from fall in the potential function.

- Observe that $d(C_t, C'_t)$ is simply $\sum_{i=1}^{K} d(s_i, a_i)$ for the scenario of straightline geometry.
- **Offline algorithm movement** of servers for the request r_t.
- We have the following equations for potential functions for transition [1].

$$\Phi(C_{t-1}, C'_t) = K * d(C_{t-1}, C'_t) + \sum_{a_i, a_j \in C_{t-1}} d(a_i, a_j) \quad (5)$$

$$\Phi(C_{t-1}, C'_{t-1}) = K * d(C_{t-1}, C'_{t-1}) + \sum_{a_i, a_j \in C_{t-1}} d(a_i, a_j) \quad (6)$$

- By the definition of Φ in Equation 4 and from Equations 5 and 6 we deduce

$$\Phi(C_{t-1}, C'_t) - \Phi(C_{t-1}, C'_{t-1}) = K * [d(C_{t-1}, C'_t) - d(C_{t-1}, C'_{t-1})] \tag{7}$$

- Now by the triangle inequality $d(C_{t-1}, C'_t) \leq d(C_{t-1}, C'_{t-1}) + d(C'_{t-1}, C'_t)$ and inequality 7 we have

$$\Phi(C_{t-1}, C'_t) \leq \Phi(C_{t-1}, C'_{t-1}) + K * d(C'_{t-1}, C'_t) \tag{8}$$

- We will remember equation 8 for future use to prove inequality (1).

- Suppose we show for the online movement that

$$\Phi(C_t, C'_t) \leq \Phi(C_{t-1}, C'_t) - d(C_{t-1}, C_t) \tag{9}$$

- Combining Equation 8 and 9 we get
$\Phi(C_t, C'_t) + d(C_{t-1}, C_t) \leq \Phi(C_{t-1}, C'_t) \leq \Phi(C_{t-1}, C'_t)$ or
$d(C_{t-1}, C_t) - K * d(C'_{t-1}, C'_t) \leq \Phi(C_{t-1}, C'_{t-1}) - \Phi(C_t, C'_t)$
- But $d(C_{t-1}, C_t) = cost(t)$ and $d(C'_{t-1}, C'_t) = opt(t)$.
- So we have established inequality (1)
- Therefore, inequality (2) follows and we are done.

- Finally, to show Equation 9 for movement of online steps, we do as follows.
- Let us account the cost of the online algorithm for moving survers at the request r_t.
- Again, by the definition of Φ, we have $\Phi(C_t, C_t') = K \times d(C_t, C_t') + \sum_{a_i, a_j \in C_t} d(a_i, a_j)$ and $\Phi(C_{t-1}, C_t') = k \times d(C_{t-1}, C_t') + \sum_{a_i, a_j \in C_{t-1}} d(a_i, a_j)$

- Observe that if r_t is a point between two online servers s_i and s_{i+1}, then one of them moves towards its matching point of the offline configuration and the other server may move away from its matching offine server an equal distance.
- So, their total contribution does not increase the matching, i.e., $d(C_t, C'_t) - d(C_{t-1}, C'_t) \leq 0$.
- Without loss of generality assume that $d(s_i, r_t) \leq d(s_{i+1}, r_t)$.

- Since s_i and s_{i+1} move towards r_t by the same distance, $\sum_{a_i,a_j \in C_t} d(a_i, a_j) - \sum_{a_i,a_j \in C_{t-1}} d(a_i, a_j)$ is reduced by $2d(s_i, r_t)$.
- So, $\Phi(C_t, C'_t) - \Phi(C_{t-1}, C'_t) \leq 2d(s_i, r_t)$, or
- $\Phi(C_t, C'_t) \leq \Phi(C_{t-1}, C'_t) - 2d(s_i, r_t)$, or
- $\Phi(C_t, C'_t) \leq \Phi(C_{t-1}, C'_t) - d(C_{t-1}, C_t)$, where $d(C_{t-1}, C_t)$ represents the change in the distance between online servers.
- This is the very Inequality 9 for this case.

- Now consider the other case where r_t lies outside the interval of the K servers, and only one server (say, s_1) moves to r_t.
- Here, the first term of the potential decreases by $K \times d(s_1, r_t)$, because s_1 moves closer to its matching point a_1.
- The second term of the potential increases by $(K-1) \times d(s_1, r_t)$ as the distance to s_1 from s_2, s_3, ... , s_k increases by $d(s_1, r_t)$.
- The difference of these two terms is $d(s_1, r_t)$, which is equal to $d(C_{t-1}, C_t)$.
- So, in this second case too, $\Phi(C_t, C'_t) \leq \Phi(C_{t-1}, C-_t) - d(C_{t-1}, C_t)$, that is, Inequality 9
- This completes the proof.

References

- Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.
- Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.
- David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.