Approximation Algorithms & Linear Programming

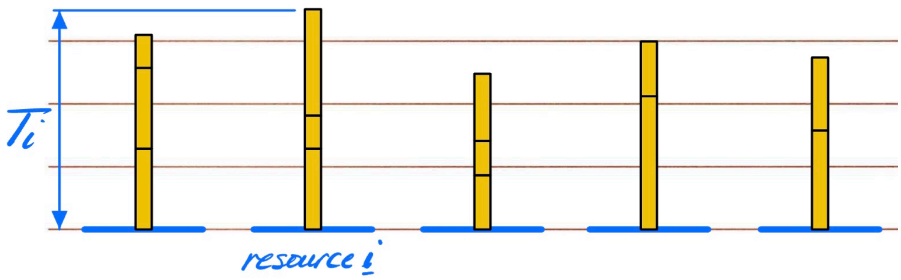
Julius April 17, 2024

1 Approximation Algorithms

Approximation Algorithms are Polynomial-time efficient algorithms that find approximate solution that is close enough to the optimal solution of Optimization problems (HP-Hard Problems).

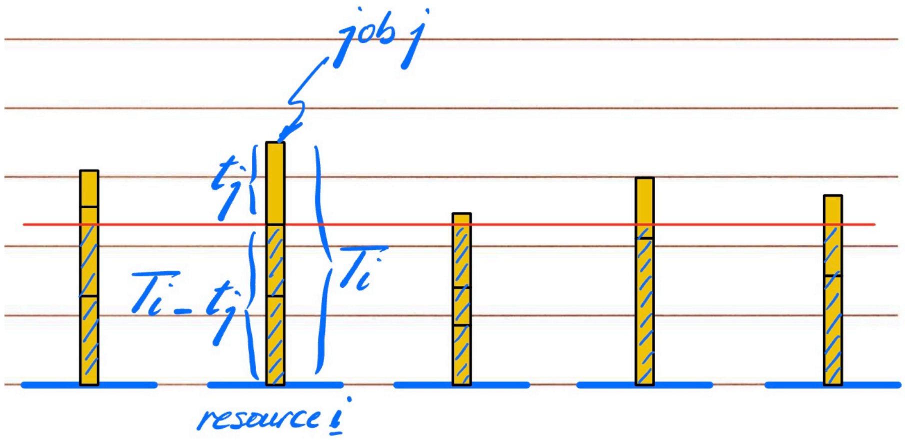
1.1 Load Balancing Problem

Given m resource with equal processing power and n jobs each takes t_j time to process.
The goal is to minimize the makespan, the time it takes to finish all jobs or the maximum load on any resource.
Notation: T_i: load on resource i, T^*: value of optimal solution, i.e. minimized makespan.



Greedy Balancing: Process requests in the order given, assign the next job to the resource with the smallest load. **Greedy Balancing is a 2-approximation algorithm.**

Proof. Let resource i ends up with the biggest load, and job j is the last job placed on resource i.



Consider load on resource i, we know that $t_j \leq T^*$ since job j can not be split across resources. And we know that $T_i - t_j \leq T^*$ since at time that job j was placed on resource i, all other resources must have a load of at least $T_i - t_j$. Therefore, $t_j + (T_i - t_j) \leq T^* + T^* \Rightarrow T_i \leq 2T^*$. □

Improved Greedy Balancing: Sort jobs in decreasing order of processing time, assign the next job to the resource with the smallest load. **Improved Greedy Balancing is a 1.5-approximation algorithm.**

Proof. Let resource i ends up with the biggest load, and job j is the last job placed on resource i.
If we have no more than m jobs, the algorithm is obviously optimal.
Otherwise, we will have at least two jobs on resource m. We get $T^* \geq t_m + t_{m+1}$, notice that $t_m \geq t_{m+1}$ we get $T^* \geq 2t_{m+1}$. Because jobs are sorted in decreasing order of processing time, $t_j = t_n \leq t_{m+1}$, so $T^* \geq 2t_j \Rightarrow t_j \leq 0.5T^*$. Besides, we still have $T_i - t_j \leq T^*$, so $T_i - t_j + t_j \leq 1.5T^* \Rightarrow T_i \leq 1.5T^*$. □

1.2 Vertex Cover & Set Cover Problem

Optimization Version: Find the smallest set of vertices such that every edge has at least one endpoint in the set.
It's been shown that a constant approximation algorithm less than 1.3606 cannot be achieved for Vertex Cover, unless P=NP.
2-approximation algorithm: Start with an empty set, select an edge e not covered by the current set and add both endpoints of e to the set. Repeat until all edges are covered.
At each step, we place both ends of edge e in S, but the optimal solution will need at least one of them, so we are within a factor of 2 of the optimal solution.
Since Independent Set $\leq_p$ Vertex Cover, can we use a 2-approximation algorithm for Vertex Cover to find a 1/2-approximation to Independent Set?
No! E.g. consider a square with 4 vertices and 4 edges, above algorithm will select all 4 vertices for Vertex Cover, leading to an Independent Set of size 0. But the optimal Independent Set is of size 2.
Since Vertex Cover $\leq_p$ Set Cover, can we use a 2-approximation algorihtm for Set Cover to find a 2-approximation to Vertex Cover?
Yes! Because any Set Cover instance can be converted to a corresponding Vertex Cover instance by creating a vertex for each set and an edge for each element in the universe. So a Set Cover Solution of size k will produce a same size k solution for the corresponding Vertex Cover problem. And the optimal solution size for Set Cover is also equal to the optimal solution size for the corresponding Vertex Cover problem.
Since Independent Set $\leq_p$ Set Packing, we can use a 2-approximation algorithm for Set Packing to find a 2-approximation to Independent Set because of the same reason as above.

1.3 Independent Set Problem

Theorem: Unless P=NP, there is no $1/n^{1-\epsilon}$ approximation algorithm for the Maximum Independent Set problem for any $\epsilon > 0$, where n is the number of nodes in graph.
For General Independent Set problem, no approximation algorithm can guarantee to stay within a constant factor of the optimal solution.
However, for IS on **Bounded Degree Graph with** $\max\{d(v) : v \in V\} = \Delta$, Greedy-IS is an Approximation Algorithm that can guarantee to stay within a constant factor $1/(\Delta + 1)$ of the optimal solution.
Start from an empty set S, in each iteration choose a vertex with the smallest degree in the current Graph G, add it to S and remove it and all its neighbors from G. Repeat until G is empty.

Proof. Let $K_i = \{v \cup Neighbors(v)\}$ be the set of vertices removed in the i-th iteration, and there are t iterations in total. Let S^* be the optimal solution. Then $K_1 \cup K_2 \cup \ldots \cup K_t = |V|, K_1 \cap K_2 \cap \ldots \cap K_t = \emptyset$, and $|K_i \cap S^*| \leq |K_i| = \Delta + 1$. Then we have $|S^*| = |V \cap S^*| = \sum_{i=1}^{t} |K_i \cap S^*| \leq t(\Delta + 1) = |S|(\Delta + 1)$, so $|S| \geq |S^*|/(\Delta + 1)$ □

1.4 Max-3SAT Problem

Given a set of clauses of length 3, find a truth assignment that satisfies the maximum number of clauses.
Find a 1/2-approximation algorithm for Max-3SAT is really easy. **Just set all variables to true**, if at least half of the clauses are satisfied, we are done. **Otherwise, set all variables to false.**
More sophisticated algorithms can get to within a factor of 8/9 of the optimal solution.

1.5 Max-DAG Problem

Given a directed graph G, remove some edges to turn G into a Directed Acyclic Graph of Maximum size.
Find a 1/2-approximation algorithm for Max-DAG is really easy. **Choose an arbitrary ordering of nodes**, remove all edges that does NOT follow the order, i.e. remove (i, j) iff i > j. **So the remaining graph is a DAG with the chosen order as its topological order.** If the remaining DAG is of size at least m/2, then return this DAG; Otherwise, **reverse the ordering** and remove all edges that does NOT meet the reversed ordering, which will end up with the complement DAG of the previous one with size at least m/2.

Proof. For any arbitrary ordering of nodes, there will be k edges consistent with one ordering and m-k edges consistent with the reverse ordering. So either $k \geq m/2$ or $k - m \geq m/2$. The optimal DAG with p edges naturally satisfies $p \leq m$. So we are guaranteed that either $k/p \geq 1/2$ or $(m - k)/p \geq 1/2$. □

1.6 Bin Packing Problem

Given infinite supply of bins with maximum weight M and n objects with weight $w_i \leq M$. Partition objects into bins to minimize the number of bins used. This problem general is NP-Hard.
Given a 2-approximation algorithm for Bin Packing by just place objects in the bin in the order given until there is no more room in the current bin, then start a new bin.

Proof. The **total weight of objects placed in every other bin pairs must be greater than M**, since we started placing objects in the second new bin when we run out of space in the first bin. So the optimal solution needs at least 1 bin for every 2 bins we used in above algorithm. So if above algorithm uses k bins, the optimal solution needs at least k/2 bins. Therefore, the above algorithm is a 2-approximation algorithm. □

1.7 Upper-Bounded Subset Sum Problem

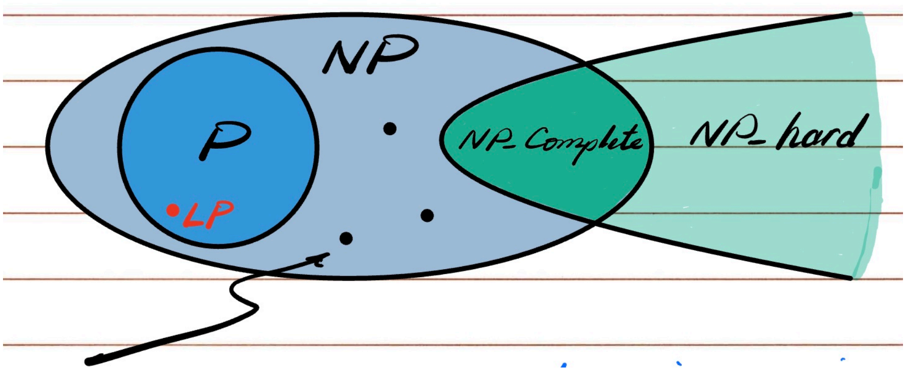
Given a set of positive integers $A = \{a_1, a_2, \cdots, a_n\}$ and a positive integer $B \geq a_i$, find the largest sum of subset of A on condition that the sum does not exceed B.
Obviously, this is a special case of the 0/1 Knapsack Problem with all item profit equal to its weight.
Given a **Linear Time Algorithm** to find a feasible subset of A that sums to at least half of the optimal solution. Start from a_1, place integers into S as long as their sum does not exceed B until we reach B/2. At any time, if the sum of S is greater than B/2, we are done. **If the sum of S is greater than B, remove all integers in S except the last added one, which must be greater than B/2**. Otherwise, sum of all integers in S can't exceed B.

1.8 0/1 knapsack Problem

Given a set of n items, each with a weight w_i and a profit p_i, and a maximum weight W, find the most valuable subset of items that fit into the knapsack. $p_i \geq 0, 0 \leq w_i \leq W$
Greedy Algorithm: Sort items in decreasing order of profit per unit weight p_i/w_i, add items to the knapsack in this order until the next item does not fit.
However, there is no NonZero Constant Approximation Ratio $0 < \rho < 1$ where we can guarantee above greedy algorithm will find a solution within ρ of the optimal solution.
Consider an item with profit 2 and weight 1, another item with profit W and weight W > 1. The greedy algorithm will select the first item, while the optimal solution is to select the second item. So the algorithm has approximation ratio 2/W, which can be arbitrarily close to 0 as W becomes large.
Actually, there is an Approximation Algorithm in Polynomial Time Approximation Scheme (PTAS), i.e. given any $\epsilon > 0$, the algorithm can find a solution within a factor of ϵ of the optimal solution.
Given $\epsilon > 0$, let $P = \max_i\{p_i\},\ K = \epsilon * P * n,\ p'_i = \lfloor p_i * K \rfloor$, then solve the 0/1 knapsack problem with weights w_i and profits p'_i via Dynamic Programming to get the solution S, where the state space is at most $n\lfloor P * K \rfloor$, so the time complexity $O(n^2 \lfloor P * K \rfloor) = O(n^2 \lfloor n\epsilon \rfloor)$ is polynomial. As long as K is small enough, the solution S will always be within a factor of ϵ of the optimal solution.

2 Linear Programming

Standard Linear Programming: Given a set of linear $\leq$ inequalities $A_{m*n}\vec{X}_n \leq \vec{B}_m$, where $\vec{X}_n \geq \vec{0}_n$, find the Maximum value of a Objective Linear Function $\vec{C}_n^T \vec{X}_n$ subject to the given constraints.
Feasible Region: the intersection of all half-spaces defined by the inequalities.
Simplex Algorithm: Start at a vertex of the feasible region, move to an adjacent vertex that improves the objective function. Repeat until no such move is possible. Because the intersection of a set of linear constraints must be a convex region. And because the objective function is also linear, the optimal solution will always be at a vertex of the feasible region. **Simplex Method has an exponential time worst case performance, while in practive normally runs in polynomial time.**
LP was in NP-Intermediate, until 1979 when Karmarkar introduced an interior-point method that runs in polynomial time to solve LP. Now LP is in P.



2.1 Weighted Vertex Cover Problem

In $G = (V, E)$ $\forall i \in V$ $w_i \geq 0$, $S \subset V$ is a vertex cover if every edge in E has at least one endpoint in S. The total weight of S is $\sum_{i \in S} w_i$. The goal is to find a vertex cover S of minimum weight $\min\{w(S)\}$.
Decision Variables: $x_i = 1$ if vertex $i \in V$ is in the cover, 0 otherwise.
Then to make sure each edge $e = (i, j)$ is covered, we must have $x_i + x_j \geq 1$.
So the LP formulation is: $\forall(i, j) \in E,\ x_i + x_j \geq 1 \quad \forall i \in V,\ x_i \in \{0, 1\}$
Objective Function:

$$\min\{\sum_{i \in V} w_i x_i\}$$

.
The above LP formulation is not a standard LP, because the decision variables are not continuous.
Integer Linear Programming: The decision variables are Discrete, Objective and Constraints are linear. It's Optimization Version is considered NP-Hard, while Decision Version is NP-Complete.
Mixed Integer Linear Programming: The decision variables are a mix of continuous and discrete, Objective and Constraints are linear.
Nonlinear Programming: Nonlinear Constraints or Objective function. E.g. $\max\{x_i(1 - x_i)\}$ is quadratic.
To find an approximate solution to the above Integer LP problem, we can relax the constraints to $x_i \in [0, 1]$ to allow the decision variables to be continuous. Solve the relaxed LP in polynomial time to find $\{x_i^*\}$.
Eefine weight of the LP solution as $w_{LP} = \sum w_i x_i^*$. **How can we convert the LP solution to an ILP solution?**
Say $S_{\geq 1/2} = \{i \in V : x_i^* \geq 1/2\}$. Because in LP $\forall e = (i, j) \in E,\ x_i^* + x_j^* \geq 1$, it's impossible for both x_i^*, x_j^* to be less than 1/2. So $S_{\geq 1/2}$ is a vertex cover. Assume S' is the optimal vertex cover set and $w(S')$ is the weight of the optimal vertex cover set. Then $w_{LP} \leq w(S')$ and $w(S_{\geq 1/2}) \leq 2 * w_{LP}$ (because at most we round up all x_i^* from 1/2 to 1, see following example). So $w(S_{\geq 1/2}) \leq 2 * w(S')$
Suppose we have a triangle with 3 vertices and 3 edges, each vertex has a weight of 1. Then $w_{LP} = 0.5 + 0.5 + 0.5 = 1.5,\ w(S') = 2,\ w(S_{\geq 1/2}) = 1 + 1 + 1 = 3 \leq 2 * w(S')$

2.2 Max Flow Problem

Given a Directed graph G with a source S and sink T, capacity c_e on each edge e, find the maximum S-T Flow.
Decision Variables: $f(e)$ is the flow on edge e.
Constraints: $\forall v \in V \backslash \{S, T\}, \sum_{e\ into\ v} f(e) = \sum_{e\ out\ of v} f(e)$ and $\forall e \in E, 0 \leq f(e) \leq c_e$
Objective Function:

$$\max\{\sum_{e\ out\ of\ S} f(e)\}$$

Note that equalities can be easily represented as inequalities. $A = B \Leftrightarrow A \leq B, B \leq A$

2.3 Min Cut Problem

Given a Directed graph G with a source S and sink T, capacity c_e on each edge e, find the minimum S-T Cut.
Decision Variables: $x_i = \{0, 1\}$ is the set indicator for node $i \in V, x_i = 1 \Rightarrow i \in A, x_i = 0 \Rightarrow i \in B$
$y_{ij} = \{0, 1\}$ is the set indicator for edge $(i, j) \in E, y_{ij} = 1 \Rightarrow (i, j) \in Cut, y_{ij} = 0 \Rightarrow (i, j) \notin Cut$
Linear Objective Function:

$$\min\{\sum_{(i,j) \in E} c_{ij} y_{ij}\}$$

Nonlinear Objective Function:

$$\min\{\sum_{(i,j) \in E} c_{ij} x_i (1 - x_j)\}$$

Constraints: $\forall(i, j) \in E, y_{ij} \geq x_i - x_j \quad \forall(i, j) \in E, y_{ij} = \{0, 1\} \quad \forall i \in V \backslash \{s, t\}, x_i = \{0, 1\} \quad x_s = 1 \quad x_t = 0$
The first constraint is to ensure that when $x_i = 1, x_j = 0$, then $y_{ij} = 1$, which means edge (i, j) is in the cut.
Note that first constraint does not directly constrain the value of y_{ij} if $(x_i, x_j) = (0, 1), (1, 1), (0, 0)$, while minimized the total weight will try to set $y_{ij} = 0$ whenever possible.
The last constraint is to ensure that we get a valid cut as the solution. Otherwise, the LP-solver would try to put all the variables on the same side so that no edges need to be included, which can be avoided by forcing s and t to be on opposite sides.

2.4 Shortest Path Problem

Given a directed graph G with a source S and sink T, find the shortest path from S to T.
Decision Variables: $d(v)$ is the shortest distance from S to v for each $v \in V$.
Suppose there are three vertices X, Y, Z can reach T in G, then for node T we have following constraints: $d(T) \leq$

$d(X) + c_{XT}, d(T) \leq d(Y) + c_{YT}, d(T) \leq d(Z) + c_{ZT}$
Constraints: $\forall (u,v) \in E, d(v) \leq d(u) + w(u,v) \quad d(S) = 0$
Objective Function:

$$Max\{d(T)\}$$

it's not to $\min\{d(T)\}$ where we would just make all d(v) = 0. We need to reach the upper bound of d(T) under all edge constraints.

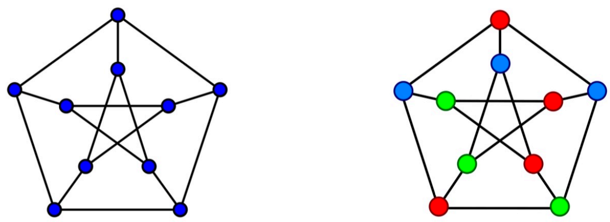
2.5 Max True-False SAT Problem

Given n boolean variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$ all of the form $x_i \wedge \bar{x}_j$, find the maximum number of clauses that can be satisfied.
Decision Variables: $y_i = \{0,1\}$ is the set indicator for clause c_i, then $y_i = 1 \Rightarrow x_i = 1, y_j = 0$.
Constraints: $\forall i \in [1,m], y_i = \{0,1\} \quad \forall i \in [1,m], y_i \leq x_i,\ y_i \leq (1 - x_j)$
Objective Function:

$$\max\{\sum_{i=1}^{m} y_i\}$$

2.6 Graph Coloring Problem

Given an undirected graph G = (V, E), find the minimum number of colors needed to color the vertices such that no two adjacent vertices have the same color. Let $|V| = n$ then use at most n colors.



Decision Variables: $\forall i = [1,n], v \in V, x_{vi} = \{0,1\}$ is the set indicator for coloring vertex v with color i.
$\forall i = [1,n], c_i = \{0,1\}$ is the color indicator for color i if any vertex is colored with color i.
Constraints: $\forall v \in V, \sum_{i=1}^{n} x_{vi} = 1 \quad \forall (u,v) \in E, \forall i \in [1,n], x_{ui} + x_{vi} \leq 1 \quad \forall v \in V, i \in [1,n], x_{vi} \leq c_i$
The second constrain is to ensure that no two adjacent vertices have the same color. The third one is to ensure that if a vertex is colored with color i, then color i is used.
Objective Function:

$$\min\{\sum_{i=1}^{n} c_i\}$$

2.7 Cargo Loading Problem

A Cargo plane can carry at most W weight of tons and V volume of cubic meters. There are n materials to be transported, each has a density of d_i tons per cubic meter, maximum available amount C_i cubic meters and revenue R_i per cubic meter. The goal is to maximize the total revenue within constraints.
Decision Variables: x_i is the amount of material i to be transported.
Constraints: $\sum_{i=1}^{n} d_i x_i \leq W \quad \sum_{i=1}^{n} x_i \leq V \quad 0 \leq x_i \leq C_i$
Objective Function:

$$\max\{\sum_{i=1}^{n} R_i x_i\}$$

2.8 Job Assignment Problem

Given a cost matrix C_{n*n} where C_{ij} is the cost of assigning worker i to do job j. Each worker i has a subset of jobs S_i that he/she is interested in. The job assignment is one on one. The goal is to minimize the total cost.
Actually, we are looking for a minimum cost perfect matching in a bipartite graph.
Decision Variables: $x_{ij} = 1$ if worker i is assigned to job j, 0 otherwise.
Constraints: $\forall i \in [1,n], \sum_{j \notin S_i} x_{ij} < 1 \quad \forall i \in [1,n], \sum_j x_{ij} = 1 \quad \forall j \in [1,n], \sum_i x_{ij} = 1 \quad x_{ij} \in \{0,1\}$
Objective Function:

$$\min\{\sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij} x_{ij}\}$$