NP-completeness and Approximation Algorithms

1 Problems

Problem 1: Special case of 3-sat (KT 10.8)

Consider the class of 3-SAT instances in which each of the n variables occurs — counting positive and negated appearances combined — in exactly three clauses. Show that any such instance of 3-SAT is in fact satisfiable, and that a satisfying assignment can be found in polynomial time.

Problem 2: Graphical Steiner tree (KT 8.38)

Consider the following version of the Steiner Tree Problem, which well refer to as *Graphical Steiner Tree*. You are given an undirected graph $G = (V, E)$, a set $X \subseteq V$ of vertices, and a number k. You want to decide whether there is a set $F \subseteq E$ of at most k edges so that in the graph (V, F), X belongs to a single connected component. Show that Graphical Steiner Tree is NP-complete.

Problem 3: Hamiltonian path (KT 10.3)

Suppose we are given a directed graph $G = (V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$, and we want to decide whether G has a Hamiltonian path from v_1 to v_n. (That is, is there a path in G that goes from v_1 to v_n, passing through every other vertex exactly once?)

Since the Hamiltonian Path Problem is NP-complete, we do not expect that there is a polynomial-time solution for this problem. However, this does not mean that all nonpolynomial-time algorithms are equally "bad." For example, here's the simplest brute-force approach: For each permutation of the vertices, see if it forms a Hamiltonian path from v_1 to v_n. This takes time roughly proportional to $n!$, which is about 3×10^{17} when $n = 20$.

Show that the Hamiltonian Path Problem can in fact be solved in time $O(2^n \cdot p(n))$, where $p(n)$ is a polynomial function of n. This is a much better algorithm for moderate values of n; for example, 2^n is only about a million when $n = 20$.

In addition, show that the Hamiltonian Path problem can be solved in time $O(2^n \cdot p(n))$ and in *p*olynomial space.

Problem 4: MaxCut for Trees (KT 10.9)

Give a polynomial-time algorithm for the following problem. We are given a binary tree $T = (V, E)$ with an even number of nodes, and a nonnegative weight on each edge. We wish to find a partition of the nodes V into two sets of *equal* size so that the weight of the cut between the two sets is as large as possible (i.e., the total weight of edges with one end in each set is as large as possible). Note that the restriction that the graph is a tree is crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

Problem 5: Perfect assembly (KT 8.32)

The mapping of genomes involves a large array of difficult computational problems. At the most basic level, each of an organism's chromosomes can be viewed as an extremely long string (generally containing millions of symbols) over the four-letter alphabet $\{A, C, G, T\}$. One family of approaches to genome mapping is to

generate a large number of short, overlapping snippets from a chromosome, and then to infer the full long string representing the chromosome from this set of overlapping substrings.

While we won't go into these string assembly problems in full detail, here's a simplified problem that suggests some of the computational difficulty one encounters in this area. Suppose we have a set $S = \{s_1, s_2, \ldots, s_n\}$ of short DNA strings over a q-letter alphabet; and each string s_i has length 2ℓ, for some number $\ell \geq 1$. We also have a library of additional strings $T = \{t_1, t_2, \ldots, t_m\}$ over the same alphabet; each of these also has length 2ℓ. In trying to assess whether the string s_b might come directly after the string s_a in the chromosome, we will look to see whether the library T contains a string t_k so that the first ℓ symbols in t_k are equal to the last ℓ symbols in s_a, and the last ℓ symbols in t_k are equal to the first ℓ symbols in s_b. If this is possible, we will say that t_k *corroborates* the pair (s_a, s_b). (In other words, t_k could be a snippet of DNA that straddled the region in which s_b directly followed s_a.)

Now we'd like to concatenate all the strings in S in some order, one after the other with no overlaps, so that each consecutive pair is corroborated by some string in the library T. That is, we'd like to order the strings in S as $s_{i_1}, s_{i_2}, \ldots, s_{i_n}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $\{1, 2, \ldots, n\}$, so that for each $j = 1, 2, \ldots, n-1$, there is a string t_k that corroborates the pair $(s_{i_j}, s_{i_{j+1}})$. (The same string t_k can be used for more than one consecutive pair in the concatenation.) If this is possible, we will say that the set S has a *perfect assembly.*

Given sets S and T, the *Perfect Assembly Problem* asks: Does S have a perfect assembly with respect to T? Prove that Perfect Assembly is NP- complete.

Example. Suppose the alphabet is $\{A, C, G, T\}$, the set $S = \{AG, TC, TA\}$, and the set $T = \{AC, CA, GC, GT\}$ (so each string has length $2\ell = 2$). Then the answer to this instance of Perfect Assembly is yes: We can concatenate the three strings in S in the order $TCAGTA$ (so $s_{i_1} = s_2$, $s_{i_2} = s_1$, and $s_{i_3} = s_3$). In this order, the pair (s_{i_1}, s_{i_2}) is corroborated by the string CA in the library T, and the pair (s_{i_2}, s_{i_3}) is corroborated by the string GT in the library T.

Problem 6: TSP in (1,2)-spaces

A (1,2)-metric space is a distance funciton $d(u, v)$ on a finite set V so that $d(u, u) = 0$ and for every $u \neq v$, $d(v, u) = d(u, v) \in \{1, 2\}$. (Note that such a function always obeys the triangle inequality, so is always a metric.)

1. Prove that the problem of deciding whether there is a travelling salesman path of length $\leq L$ remains **NP**-complete on (1,2)-metric spaces.

2. Give the best polynomial-time approximation algorithm you can for this problem.

Problem 7: Three-dimensional matching

Consider the following maximization version of the three-dimensional matching problem. Given disjoint sets X, Y, and Z, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subseteq T$ is a three-dimensional matching if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The maximum three-dimensional matching problem is to find a three-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

Give a polynomial time algorithm that finds a three-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

Problem 8: Bin packing

In the bin packing problem, we are given a collection of n items with weights $w_1, w_2, \ldots, w_n$. We are also given a collection of bins, each of which can hold a total of W units of weight. (We will assume that W is at least as large as each individual w_i.)

You want to pack each item in a bin; a bin can hold multiple items, as long as the total of weight of these items does not exceed W. The goal is to pack all the items using as few bins as possible. Doing this optimally turns out to be **NP**-complete, though you don't have to prove this.

Here's a merging heuristic for solving this problem: We start with each item in a separate bin and then repeatedly "merge" bins if we can do this without exceeding the weight limit. Specifically:
Merging Heuristic:

Algorithm 1: Merging heuristic

Start with each item in a different bin
while there exist two bins so that the union of their contents has total weight W **do**
  Empty the contents of both bins
  Place all these items in a single bin.
end while
Return the current packing of items in bins.

Notice that the merging heuristic sometimes has the freedom to choice several possible pairs of bins to merge. Thus, on a given instance, there are multiple possible executions of the heuristic.

Example. Suppose we have four items with weights 1, 2, 3, 4, and W = 7. Then in one possible execution of the merging heuristic, we start with the items in four different bins; then we merge the bins containing the first two items; then we merge the bins containing the latter two items. At this point we have a packing using two bins, which cannot be merged. (Since the total weight after merging would be 10, which exceeds $W = 7$.)

1. Let's declare the size of the input to this problem to be proportional to

$$n + \log W + \sum_{i=1}^{n} \log w_i$$

   (In other words, the number of items plus the number of bits in all the weights.) Prove that the merging heuristic always terminates in time polynomial in the size of the input. (In this question, as in **NP**-complete number problems from class, you should account for the time required to perform any arithmetic operations.)

2. Give an example of an instance of the problem, and an execution of the merging heuristic on this instance, where the packing returned by the heuristic does not use the minimum possible number of bins.

3. Prove that in any execution of the merging heuristic, on any instance, the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

Problem 9: Bin packing

Here's a way in which a different heuristic for bin packing can arise. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port. Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is **NP**-complete (you don't have to prove this). A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would over ow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck.

1. Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

2. Show that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

Problem 10: First-fit heuristic

Problem 35-1, p. 1050, parts e., f. (b-d are hints, p. 984 in old edition). Also give example where first-fit heuristic is NOT optimal. Can you prove a matching lower bound on the approximation ratio for this heuristic?

Problem 11: Cardinality maximum cut

Given an undirected graph $G = (V, E)$, the cardinality maximum cut problem asks for a partition of V into sets S and $\bar{S} = V - S$ so that the number of edges running between these sets is maximized. Consider the following greedy algorithm for this problem. Here v_1 and v_2 are arbitrary vertices in G, and for $A \subseteq V$, $d(v, A)$ denotes the number of edges between vertex v and the set of vertices A.

```
1  Initialization:
2    A ← {v1}
3    B ← {v2}
4  for v ∈ V − {v1, v2} do:
5    if d(v, A) ≥ d(v, B) then B ← B ∪ {v}
6    else A ← A ∪ {v}
7  Output A and B
```

Show that this is a factor 1/2 approximation algorithm and give a tight example. What upper bound did you use for the optimal value? Give examples of graphs for which this upper bound is as bad as twice the optimal value. Generalize the problem and the algorithm to weighted graphs.

Problem 12: Cardinality vertex cover

Consider the following factor 2 approximation algorithm for the cardinality vertex cover problem. Find a depth-first search tree in the given graph, G, and output the set, say S, of all the nonleaf vertices of this tree. Show that S is indeed a vertex cover for G and $|S| \leq 2.\mathbf{O}PT$.

Problem 13: Independent set on graphs of degree 3

Give as good a polynomial-time approximation algorithm as you can to solve independent set for graphs where every node has at most 3 neighbors. (Good means in terms of the approximation ratio, i.e. the maximum ratio of the size of an independent set in the input graph to the size of the independent set your algorithm finds.)

Problem 14: Directed acyclic graph

Given a directed graph $G = (V, E)$, pick a maximum cardinality set of edges from E so that the resulting subgraph is acyclic. Give a factor 1/2 algorithm.

Problem 15: Greedy-balance algorithm

Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they have come to you for help in getting around an unfortunate piece of legacy code that cannot be changed.

Here is the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1, which provides an approximation to within a factor of 2. In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it is generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system's internals won't let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the

optimal solution. (Basically, this portion of the code has to interact with so many other parts of the system that it is not worth the risk of something going wrong if it is replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they are able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm *in an order that will cause it to allocate them optimally.* In other words, they are hoping to be able to reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is,

> For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

Problem 16: Heaviest first (KT 11.10)

Suppose you are given an $n \times n$ grid graph G, as in Figure 1 Associated with each node v is a weight $w(v)$,

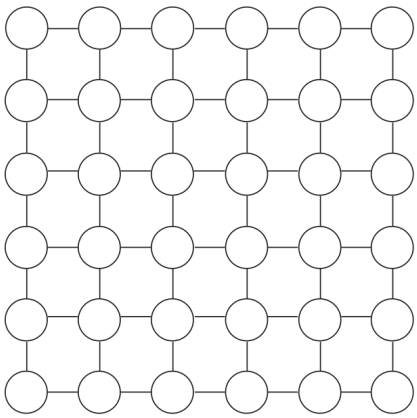

Figure 1: A grid graph

which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight.*)

Consider the following greedy algorithm for this problem.

Algorithm 2: The "heaviest-first" greedy algorithm

```
Start with S equal to the empty set
while some node remains in G do
  Pick a node v_i of maximum weight
  add v_i to S
  Delete v_i and its neighbors from G
end while
return S
```

1. Let S be the independent set returned by the "heaviest-first" greedy algorithm, and let T be any other independent set in G. Show that, for each node $v \in T$,either $v \in S$,or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G.

2. Show that the "heaviest-first" greedy algorithm returns an independent set of total weight at least $\frac{1}{4}$ times the maximum total weight of any independent set in the grid graph G.

Problem 17: Hitting set optimization (KT 11.4)

Consider an optimization version of the Hitting Set problem defined as follows. We are given a set $A = \{a_1, \ldots, a_\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. Also, each element a_i of A has weight $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in H, $\sum_{a_i \in H} w_i$ is as small as possible. (Recall from that H is a hitting set if $H \cap B_i$ is not empty for each i). Let $b = \max_i |B_i|$ denote the maximum size of any of the sets $B_1, B_2, \ldots, B_m$. Give a polynomial time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.

Problem 18: Approximating the independent set

The independent set problem: given an undirected graph, find the largest independent set S of vertices, i.e., a set so that no adjacent vertices u and v are both in S.

1. Give a linear programming relaxation for the independent set problem.

2. Give a polynomial time algorithm that, given a graph G with n vertices, either finds an independent set of size at least $n/3$ or certifies that no independent set in G has size greater than $2n/3$. (It may involve using an algorithm for linear programming.)

   Provide a clear description of your algorithm, analyze its time complexity, and argue that it either returns an independent set of size at least $n/3$ or concludes that there is no independent set of size greater than $2n/3$.

3. Consider the special case of graphs of at most degree 3. Find $\alpha > 1/3$ and a polynomial time algorithm that either finds an independent set of size αn or certifies that no independent set has size $(1 - \alpha)n$. Obtain as good an α as you can.

   Provide a clear description of your algorithm, analyze its time complexity, determine α, and argue that your algorithm either returns an independent set of size at least αn or concludes that there is no independent set of size greater than $(1 - \alpha)n$.

Problem 19: Job scheduling

You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has a set of 10 machines, and each job can be processed on any of these 10 machines.

At the moment the business is running the simple Greedy-Balance algorithm we discussed in class. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: they have been running it each day for the last month, and they have not observed it to produce a makespan more than 20% above the average load, $\frac{1}{10} \sum_i t_i$.

To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, i.e., $1 \leq t_i \leq 50$ for all jobs i; and the total load $\sum_i t_i$ is quite high each day: it is always at least 3000.

Prove that on the type of inputs the company sees, the Greedy-Balance algorithm will always find a solution whose makespan is at most 20% above the average load.

Problem 20: Knapsack (KT 11.11)

Recall that in the Knapsack Problem, we have n items, each with a weight w_i and a value v_i. We also have a weight bound W, and the problem is to select a set of items S of highest possible value subject to

the condition that the total weight does not exceed W–that is, $\sum_{i \in S} w_i \leq W$. Here's one way to look at the approximation algorithm that we designed in this chapter. If we are told there exists a subset $\mathcal{O}$ whose total weight is $\sum_{i \in \mathcal{O} w_i \leq W}$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V, then our approximation algorithm can find a set $\mathcal{A}$ with total weight $\sum_{i \in \mathcal{A}} w_i \leq W$ and total value at least $\sum_{i \in \mathcal{A}} v_i \geq V/(1+\epsilon)$. Thus the algorithm approximates the best value, while keeping the weights strictly under W. (Of course, returning the set $\mathcal{O}$ is always a valid solution, but since the problem is NP-hard, we don't expect to always be able to find $\mathcal{O}$ itself; the approximation bound of $1+\epsilon$ means that other sets A, with slightly less value, can be valid answers as well.)

Now, as is well known, you can always pack a little bit more for a trip just by "sitting on your suitcase"-in other words, by slightly overflowing the allowed weight limit. This too suggests a way of formalizing the approximation question for the Knapsack Problem, but it's the following, different, formalization.

Suppose, as before, that you're given n items with weights and values, as well as parameters W and V; and you're told that there is a subset $\mathcal{O}$ whose total weight is $\sum_{i \in \mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V. For a given fixed $\epsilon > 0$, design a polynomial-time algorithm that finds a subset of items $\mathcal{A}$ such that $\sum_{i \in \mathcal{A}} w_i \leq (1+\epsilon)W$ and $\sum_{i \in \mathcal{A}} v_i \geq V$. In other words, you want $\mathcal{A}$ to achieve at least as high a total value as the given bound V, but you're allowed to exceed the weight limit W by a factor of $1+\epsilon$.

Example. Suppose you're given four items, with weights and values as follows:

$$(w_1, v_1) = (5, 3), (w_2, v_2) = (4, 6)$$

$$(w_3, v_3) = (1, 4), (w_4, v_4) = (6, 11)$$

You're also given $W = 10$ and$V = 13$ (since, indeed, the subset consisting of the first three items has total weight at most 10 and has value 13). Finally, you're given $\epsilon = .1$. This means you need to find (via your approximation algorithm) a subset of weight at most $(1 + .1) * 10 = 11$ and value at least 13. One valid solution would be the subset consisting of the first and fourth items, with value $14 \leq 13$. (Note that this is a case where you're able to achieve a value strictly greater than V, since you're allowed to slightly overfill the knapsack.)

Problem 21: Low degree MST

The low degree spanning tree problem is as follows: Given a graph G and an integer k, does G contain a spanning tree such that all vertices in the tree have degree at most k (obviously, only tree edges count towards the degree)?

Prove that the low degree spanning tree problem is NP-hard with a reduction Hamiltonian path.

Now consider the high degree spanning tree problem: Given a graph G and an integer k, does G contain a spanning tree whose highest degree vertex is has degree least k? Give an efficient algorithm to solve the high degree spanning tree problems and an analysis of its time complexity.

Problem 22: Makespan

We are given a sequence of n jobs with durations $t_1, t_2, \cdots, t_n$. The jobs are to be allocated to k processors. An allocation is an assignment $f : \{1, 2, \cdots, n\} \to \{1, 2, \cdots, k\}$. Given f, let $A_f(i)$ denote the set of jobs allocated to processor i for $1 \leq i \leq k$. For a given allocation f, the load $L_f(i)$ of a processor i is defined as

$$L_f(i) = \sum_{j \in A(i)} t_j \tag{1}$$

The makespan of f, M_f, is defined as

$$M_f = \max_i L_f(i) \tag{2}$$

Find an allocation that minimizes the makespan.

Problem 23: Maximum bisection

Consider the Max Bisection problem: Given a weighted, undirected graph $G = (V, E)$ with an even number of nodes, find a partition of V into two sets S, T with $|S| = |T|$ that maximizes the total weights of edges in $Cut(S, T)$. Give a *deterministic* algorithm that is guaranteed to find solutions to this problem within a factor of 2 of optimal. (Hint: this is very close to the Max Cut problem. The algorithms might also be very close.)

Problem 24: Maximum coverage

Given a universal set U of n elements, with nonnegative weights specified, a collection of subsets of U, $S_1, \ldots, S_l$ and an integer k, pick k sets so as to maximize the wight of elements covered.

Show that the obvious algorithm, of greedily picking the best set in each iteration until k sets are picked, achieve an approximation factor of $1 - (1 - 1/k)^k > 1 - 1/e$.

Problem 25: Maximum directed cut

Given a directed graph $G = (V, E)$ with nonnegative edge weights, find a subset $S \subseteq V$ so as to maximize the total weight of edges out of S, i.e., $\sum_{uv \in E:\ u \in S \text{ and } v \in V - S} w_{uv}$, where w_{uv} is the weight of the directed edge uv. Give a greedy algorithm to achieve an approximation guaranteee of factor $1/4$.

Problem 26: Metric TSP variation

Consider the following variant of metric TSP: given vertices $u, v \in V$, find a minimum cost simple path from u to v that visits all vertices. First give a factor 2 approximation algorithm for this problem, and then improve it to factor $3/2$.

Give a factor 2 approximation algorithm for the following problem: Given a directed graph $G = (V, E)$ with nonnegative edge costs, and a partitioning of V into two sets Senders and Receivers, find a minimum cost subgraph such that every Receiver vertex has a path to a Sender vertex.

Problem 27: Minimum cardinality maximal matching

Design a factor 2 approximation algorithm for the problem of finding a minimum cardinality maximal matching in an undirected graph.

Problem 28: Multicover

Give a $H(n)$ facotr approximation algorithm for the minimum-cost set multicover. You are given a collection S_i for $1 \leq i \leq m$ of subsets of a universe U. Each set S_i has a nonnegative cost, $w_i > 0$. Each element $e \in U$ needs to be covered at least a specificed number r_e of times. A set multicover is a subcollection of the sets $\{S_i\}$ such that each element appears in at least r_e distinct sets in the subcollection. The cost of subcollection is the sum of the costs of the sets in the subcollection.

For $n \geq 1$, $H(n) = \sum_{i=1}^{n} \frac{1}{i}$.

Problem 29: Online algorithm

Consider the bin packing problem when restricted as follows: Bins are all size 3, and items are either size exactly 1 or 2. Call this problem $1, 2$ bin-packing.

Give an explicit formula for the size of the mimimum packing of t items of size 2 and u items of size 1.

An on-line algorithm for bin-packing must process the packages in (unsorted) order, and assign each to a bin as it is processed. Prove that no deterministic on-line algorithm for $1, 2$ bin packing has approximation ratio less than $3/2$.

Give and analyze a deterministic on-line $1, 2$ bin-packing algorithm with approximation ratio $3/2$.

Remember that an algorithm has *competitive ratio* R if there is a constant B with $Cost(Algorithm(x)) \leq R * Cost(Opt(x)) + B$ for every x. Give a deterministic on-line $1, 2$ bin packing algorithm with competitive ratio less than $3/2$.

A probabilistic algorithm has approximation ratio R if $E(Cost(Algorithm(x))) \leq RCost(Opt(x))$ for every instance x. Prove that no probabilistic on-line 1-2 bin-packing algorithm has approximation ratio less than 4/3.

Give and analyze a probabilistic on-line 1-2 bin-packing algorithm with approximation ratio less than 1.5, i.e., better than any deterministic algorithm.

Problem 30: Almost optimal binary search tree problem

In the optimal binary search tree problem, you are given n sorted elements $x_1 < x_2 < \ldots, < x_n$, each with a non-negative probability p_i of being searched for, with $\sum_i p_i = 1$. A solution is a binary search tree containing the elements. The cost of such a tree is the expected number of comparisons required to find an element in the tree, if each element x_i is searched for with probability p_i. In other words, let element x_i be stored at a node of depth d_i in the tree. Then the cost of the tree is $\sum_i d_i p_i$.

There is an n^3 time dynammic programming algorithm for this problem. However, the following greedy heuristic does almost as well: Put the weighted median of the set at the root. Then recurse to create the left and right sub-trees.

Describe a fast way of computing the tree given by this greedy heuristic.

Problem 31: Representative set

At a lecture in a computational biology conference about a year ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set, and thereby learn (by inference) about all the proteins in the full collection.

To be useful, the representative set must have two properties.

- It should be relatively small, so that it will not be too expensive to study it.
- Every protein in the full collection should be "similar" to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by *distance function* d —- given two proteins p and q, it returns a number $d(p, q) \geq 0$. In fact, the function $d(.,.)$ most typically used is the edit distance, or sequence alignment measure. We'll assume this is the distance being used here. There is a pre-defined distance cut-off Δ that's is specified as part of the input to the problem; two proteins p and q are deemed to be "similar" to one another if and only if $d(p, q) \leq \Delta$. . We say that a subset of P is a representative set if for every protein p, there is a protein q in the subset that is similar to it, i.e., for which $d(p, q)\Delta$. Our goal is to find a representative set that is as small as possible.

1. Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: if the minimum possible size of a representative set is s^*, your algorithm shour return a representative set of size at most $O(s^* \log n$.
2. Note the close similarity between this problem and the Center Selection problem — a problem for which we considered approximation algorithms in the text. Why doesn't the algorithm described there solve the current problem?

Problem 32: Scheduling

Consider the following scheduling problem. You are given a set of n jobs, each of which has a time requirement t_i. Each job can be done on one of two identical machines. The objective is to minimize the total time to complete all jobs, i.e., the maximum over the two machines of the total time of all jobs scheduled on the machine. A greedy heuristic would be to go through the jobs and schedule each on the machine with the least total work so far.

1. Give an example (with the items in sorted order) where this heuristic is not optimal.

2. Assume the jobs are sorted in decreasing order of time required. Show as tight a bound as possible on the ratio between the optimal solution and that found by the greedy heuristic. A ratio of 7/6 or better would get full credit. A ratio worse than 7/6 might get partial credit.

Problem 33: Scheduling jobs for parallel machines

Say that you are given a set of n jobs to be scheduled on a parallel computer with P processors. Each job J_i has a (positive real-valued) duration D_i and an (integer-valued) processor requirement $1 \leq p_i \leq P$. You need to assign each job an interval of time of length D_i, $[s_i, f_i = s_i + D_i)$, $s_i \geq 0$, to be run on the machine. At any time t, the total processor requirements of the set of jobs going on at time t can be at most p, i.e. $\sum_{\{i|s_i \leq t < f_i\}} p_i \leq P$. You want to minimize the time that all jobs are finished, i.e., minimize $Cost(S)$, over all legal schedules S, where $cost(S) = max_i f_i$.

Give an *approximation algorithm* for this problem, guaranteed to be at most c times the cost of the optimum solution for a constant c. The algorithm itself may be very simple; almost all of the points will be for a good value of c and the proof that your algorithm achieves factor c approximation ratio.

Problem 34: Shortest first heuristic

Recall the Shortest-First greedy algorithm for the Interval Scheduling problem: Given a set of intervals, we repeatedly pick the shortest interval I, delete all the other intervals that intersect I, and iterate.

In class, we saw that this algorithm does not always produce a maximum-size set of non-overlapping intervals. However, it turns out to have the following approximation guarantee. If s^* is the maximum size of a set of non-overlapping intervals, and s is the size of the set produced by the Shortest-First algorithm, then $s \geq s^*/2$. (That is, Shortest-First is a 2-approximation.)

Prove this fact.

Problem 35: Spread maximization (KT 11.7)

You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor i, where $i \in \{1, 2, \ldots, n\}$, the site has assigned a value v_i, representing the expected revenue that can be obtained from this customer.

Each visitor i is shown one of m possible ads $A_1, A_2, \ldots, A_m$ as they enter the site. The site wants a selection of one ad for each customer so that *each* ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the *spread* of this selection to be the minimum, over $j = 1, 2, \ldots, m$, of the total weight of all customers who were shown ad A_j.

Example Suppose there are six customers with values 3, 4, 12, 2, 4, 6, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of 9 by showing ad A_1 to customers 1, 2, 4, ad A_2 to customer 3, and ad A_3 to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don't have to prove this). So instead, we will try to approximate it.

1. Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s, then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\overline{v} = \sum_{i=1}^{n} v_i$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\overline{v}/(2m)$.

2. Give an example of an instance on which the algorithm you designed in part (1) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.

Problem 36: Subset sum (KT 11.3)

Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called feasible if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B$$

The sum of the numbers in S will be called the total sum of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

1. Here is an algorithm for this problem.

   Algorithm 3: A "Naive" greedy algorithm

   ```
   Initially S = ∅
   Define T = 0
   for i = 1, 2, ..., n do
     if T + a_i ≤ B then
       S ← S ∪ {a_i}
       T ← T + a_i
     end if
   end for
   ```

   Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

2. Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$. Less efficient algorithms will get partial credit.

   Provide a clear description of your algorithm, analyze its time complexity, argue that it returns a feasible set, and prove its performance guarantee.

Problem 37: TSP approximation

A variant on the TSP (Traveling Salesperson Problem) permits a node to be visited more than once if it results in a better solution. Show that if the cost function satisfies the triangle inequality, then there is always an optimum solution which visits each node exactly once. Show that if the triangle inequality does not hold, this problem can be solved by solving a TSP for which the triangle inequality does hold.

A clarification: Let TSP' be the following problem: Given a complete graph with weighted edges, determine the optimal tour that visits every vertex *at least* once.

Let TSP be the following problem: Given a complete graph with weighted edges, determine the optimal tour that visits every vertex *exactly* once.

Restatement of the problem:

Given a complete graph that satisfies the triangle inequality for its edge weights, prove that there is an optimal tour (according to TSP problem) which is also optimal according to TSP' problem.

Given a complete graph G which does not necessarily satisfy the triangle inequality, efficiently transform G to G' on the same vertex set but with different weights such that G' satisfies triangle inequality and such that the optimal tour in G' (according to TSP) can be efficiently transformed to an optimal tour in G (according to TSP').

Problem 38: Vertex coloring

Given an undirected graph $G = (V, E)$, color its vertices with the minimum number of colors so that the two end points of each edge receive distinct colors.

1. Give a greedy algorithm for coloring G with $\Delta + 1$ colors, where Δ is the maximum degree of a vertex in G.

2. Give an algorithm for coloring a 3-colorable graph with $O(\sqrt{n})$ colors.

2 Solved Problems

Problem 39: Spread maximization (KT 11.7)

You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor i, where $i \in \{1, 2, \ldots, n\}$, the site has assigned a value v_i, representing the expected revenue that can be obtained from this customer.

Each visitor i is shown one of m possible ads $A_1, A_2, \ldots, A_m$ as they enter the site. The site wants a selection of one ad for each customer so that *each* ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the *spread* of this selection to be the minimum, over $j = 1, 2, \ldots, m$, of the total weight of all customers who were shown ad A_j.

Example Suppose there are six customers with values 3, 4, 12, 2, 4, 6, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of 9 by showing ad A_1 to customers 1, 2, 4, ad A_2 to customer 3, and ad A_3 to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don't have to prove this). So instead, we will try to approximate it.

1. Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s, then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\overline{v} = \sum_{i=1}^{n} v_i$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\overline{v}/(2m)$.

2. Give an example of an instance on which the algorithm you designed in part (1) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.

Solution: Spread maximization (KT 11.7)

2.1 1

This is the same as the load balancing problem from pg 600 except that the objective is to maximize the minimum load instead of minimize the maximum load. Let $v_1, \ldots, v_n \geq 0$ be the loads to be spread across the m processors. Let $\bar{v} = \sum_{i=1}^{n} v_i$. The problem lets us assume that $\forall i \in [n]\ v_i \leq \frac{\bar{v}}{2m}$. We use the same algorithm as is on pg 601: for i going from 1 to n, assign job i to the least loaded machine (where the *load* of machine j is the sum of v_k such that k is assigned to j).

We claim that the maximum possible minimum load achievable L^* divided by the minimum load L achieved by the algorithm is $\leq 2 - \frac{2}{m+1}$, slightly better than the problem asks for. To see this, note that some $k \in [1, m-1]$ processors, say processors $1, \ldots, k$, get a load at least the average $\frac{\bar{v}}{m}$, and let w_j be the load of the jth one minus $\frac{\bar{v}}{m}$. w_j is in some sense the *waste* of j. We can also define w to be $\frac{\bar{v}}{m}$ minus the load of a least loaded processor, say processor $k+1$. w is in some sense the *waste* of $k+1$. For each $j \in [k]$, let a_j be the size of the last job allocated to j. Then $a_j \geq w_j + w$, since otherwise we would have allocated

it to $k+1$. Also, $w \le w_1 + \cdots + w_k$, and so $\min_{j\in[k]} w_j \ge \frac{w}{m-1}$. So $w \le \min_{j\in[k]} a_j - w_j \le \frac{\bar{v}}{2m} - \frac{w}{m-1}$. So $w(1+\frac{1}{m-1}) \le \frac{\bar{v}}{2m}$. So

$$\frac{L^*}{L} \le \frac{\frac{\bar{v}}{m}}{\frac{\bar{v}}{m} - w} \le \frac{\frac{\bar{v}}{m}}{\frac{\bar{v}}{m} - \frac{\frac{\bar{v}}{2m}}{1+\frac{1}{m-1}}} = 2 - \frac{2}{m+1}.$$

2.2 Even better

We claim that we can improve the approximation ratio to $\frac{3}{2}$ by first ordering the jobs from largest to smallest. To see this, consider some input on which the approximation ratio achieved is > 1. Let L_i be the load on processor i and S_i be the set of jobs on i. Let j be a least loaded processor and $w = \frac{\bar{v}}{m} - L_j$ be its waste.

Lemma 2.1. $\exists i \in [m](L_i \ge \frac{\bar{v}}{m} \wedge |S_i| \ge 3)$.

Proof. Suppose not so that $\forall i \in [m]$ $(L_i < \frac{\bar{v}}{m} \vee |S_i| \le 2)$. Let $k = |\{i \in [m] \mid L_i < \frac{\bar{v}}{m}\}|$. If $k = 0$, then $\forall i \in [m]$ $L_i = \frac{\bar{v}}{m}$, and so the approximation ratio is 1, a contradiction. So $k > 0$. So we have the contradiction

$$\bar{v} = \sum_{i\ni L_i < \frac{\bar{v}}{m}} L_i + \sum_{i\ni L_i \ge \frac{\bar{v}}{m}} L_i < k\frac{\bar{v}}{m} + (m-k)2\frac{\bar{v}}{2m} = \bar{v}.$$

□

Let i be as in the lemma and let a_i be the size of the last job allocated to i and $w_i = L_i - \frac{\bar{v}}{m}$ be its waste. Then $a_i \ge w_i + w$, otherwise the algorithm would have preferred allocating a_i to j. Since $|S_i| \ge 3$, $\frac{\bar{v}}{m} = L_i - w_i \ge 3a_i - w_i \ge 3(a_i - w_i) \ge 3w$. So the approximation ratio is

$$\le \frac{\frac{\bar{v}}{m}}{\frac{\bar{v}}{m} - w} \le \frac{\frac{\bar{v}}{m}}{\frac{\bar{v}}{m} - \frac{\bar{v}}{3m}} = \frac{3}{2}.$$

2.3 2

The analysis is tight for the first algorithm. For consider the example with $m(m+1)$ loads of value 1 and $m-1$ more of value m. Then $\frac{\bar{v}}{2m} = m$ and the algorithm will assign $m+1$ 1's and a single m to each of the first $m-1$ processors, and then $m+1$ 1's to the last processor, achieving a minimum load of $m+1$. But we could have assigned $2m$ 1's to the first processor and (a single m and m 1's to each of the rest), achieving a minumum load of $2m$. So the approximation ratio is $2 - \frac{2}{m+1}$.

I doubt the analysis is tight for the second algorithm.