Algorithms: CSE 202 — Homework IV

Problem 1: 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.

Solution: Hamiltonian path (KT 10.3)

- Given a set S, let's first compute the number of paths from s to v with a length of $n-1$. We solve this using dynamic programming.

  Let $P_S(u, k)$ denotes the number of walks of length k from s to u which passes through the vertext in S. For $u \in S \cup \{t\}$,

$$P_S(u, 0) = \begin{cases} 1 & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

$$P_S(u, k) = \sum_{w \in S \text{ and } (u,w) \in E} P_S(w, k-1) \text{ where } k = 1, 2, .., n-2$$

$$P_S(t, n-1) = \sum_{w \in S \text{ and } (t,w) \in E} P_S(w, n-2)$$

  $P_S(t, n-1)$ is the number of walks of length $n-1$ from s to t which passes through the vertext in S. Note that if graph $G = \{V, E\} (s \in V, t \in V, |V| = n+1)$ has a *Hamiltonian Path*, then $P_{V-\{s,t\}}(t, n-1) \geq 1$.

- Let X denotes the number of *Hamiltonian paths* in $G = \{V, E\}$ where $V = \{v_1, v_2, ..., v_n\}$. For $S \subseteq \{v_2, v_3, ..., v_{n-1}\}$, let $N(S)$ denotes the number of walks of length $n-1$ from v_1 to v_n using vertex in set $\{v_2, v_3, ..., v_{n-1}\} - S$, then

$$X = \sum_S (-1)^{|S|} N(S)$$

  Let's explain this equation. Since vertex can repeat in the path, not all paths with a length of $n-1$ in V are *Hamiltonian Path*. If a path with the length $n-1$ in G is not a *Hamiltonian Path*, then this path must exists in one or more subgraphs of G(some vertex are not used). Therefore to compute the number of *Hamiltonian Path*, we need to substract the number of this kind of paths from the total.

However, the same path may be substracted multiple times since a subset may have multiple supersets. So we need to add back those paths. Use inclusion and exclusion, the value of X should be:

$$X = \sum_{S} (-1)^{|S|} N(S)$$

You can refer to the paper "DYNAMIC PROGRAMMING MEETS THE PRINCIPLE OF INCLUSION AND EXCLUSION" by Richard M.Karp for this idea.

- To find the actual path, we can use self reduction. Starting from v_1, for each neighbor of v_1, say w, we can delete the node w from the graph and ask if there is a *Hamiltonian Path* from w to v_n exists in the new graph. If we find one, we repeat the self reduction step, else we go to the next neighbor.
- For a fixed subset S, the time complexity for computing $P_S(v_n, n-1)$ is $O(n|E|)$, we have 2^n subsets, therefore the total time complexity is $O(n|E|2^n)$. For space complexity, we compute the value for one subset at one time. This means space can be reused. So we just need $O(n)$ intermediate space.

Problem 2: 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.

Solution: MaxCut for Trees (KT 10.9)

We root the tree at some node r and associate a subtree T_v with each node $v \in T$, and let n_v denote the number of nodes in the subtree T_v. Let $OPT(v, k)$ be the maximum cut of the subgraph T_v separating T_v into two sides where the side containing v has size k. The final answer is $OPT(r, n/2)$ if $n = n_r$ is the number of nodes of T.

We will use $c(u, v)$ as the cost or weight of the edge $e = (v, w)$ of the tree. For a leaf v, we have $k = 1$ as the only possible value and $OPT(v, 1) = 0$. Now consider $OPT(v, k)$ for some non-lead v. The side A containing v must contain $k - 1$ nodes in addition to node v. If v has only one child u, then we need to consider two cases depending on whether or not the child u is on the same side of the cut as v, so we get that in this case

$$OPT(v, k) = max(OPT(u, k-1), c(v, u) + OPT(u, n_u - k + 1)).$$

If v has two children u and w, then we will consider all possible ways of dividing these $k - 1 = l_1 + l_2$ nodes between the two subtrees rooted at the two children of v. For each such division, there are further case depending whether or not the root of these subtrees is on the same side of the cut as v. We get the following recurrence in this case

$$\begin{aligned} OPT(v, k) = \max_{0 \le l_1 \le k-1, l_2 = k-1-l_1} & (max(OPT(u, l_1), c(v, u) + OPT(u, n_u - l_1)) \\ & + max(OPT(w, l_2), c(v, w) + OPT(w, n_w - l_2))). \end{aligned}$$

There are $O(n^2)$ subproblems. We can compute the values of the subproblems starting at the leaves, and gradually considering bigger subtree. This recurrence allows us to compute the value of a subproblem in $O(k) = O(n)$ time given the values of the subproblems associated with smaller trees. So the total time required is $O(n^3)$.

Problem 3: 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 $w(v)$,

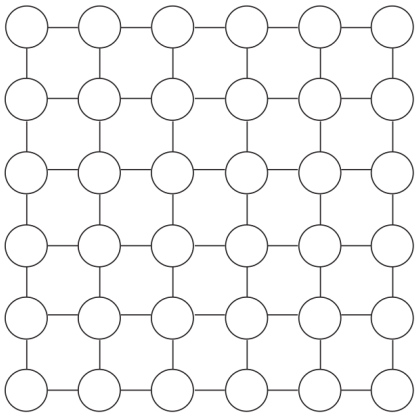

Figure 1: A grid graph

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 1: 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.

Solution: Heaviest first (KT 11.10)

1. If node $v \notin S$, it must have never been chosen by the greedy algorithm, in other word, it has been deleted in some iteration since it's neighbor v' has been chosen. According to our choosing rule, the weight of v' is at least as much as the weight of v, otherwise, in that iteration we would have chosen v instead of v'. Therefore, $w(v) \leq w(v')$.

2. - Define $S_1 = T_1 = \{v | v \in T, v \in S\}$ which denotes the nodes in both T and S
   - Define $T_2 = \{v | v \in T, v \notin S\}$ as the set of nodes only exist in T
   - Define $S_2 = \{v' | (v, v') \in E, v \in T_2\}$ which denotes the nodes in S that are neighbors of nodes in T(nodes only exist in S)

   According to the definitions and the fact that S and T are independent sets, we have the following assertions:

   - $T_1 \cap T_2 = \emptyset, T = T_1 \cup T_2$
   - $S_1 \cap S_2 = \emptyset, S = S_1 \cup S_2$

The first one is easy to prove. For the second assertion, let's prove $S_1 \cap S_2 = \emptyset$. Suppose there is a node v' exists in both S_1 and S_2, we know that there is a node v which is the neighbor of v' and $v \in T_2$, and since $v' \in S_1$, $v' \in T_1$. Therefore, $v' \in T_1, v \in T_2$ and $(v, v') \in E$, this contradicts the fact that T is an independent set. Thus $S_1 \cap S_2 = \emptyset$. The two assertions must hold.

From (a) we know that for any node v in T_2, there is a neighbor node v' of v exists in S_2 such that $w(v) \leq w(v')$. If we use v' to substitute v, since a node in S_2 can be used for at most 4 times(a node has at most 4 neighbors in a grid) and it's weight is at least equal to the weight of the node it substitutes, we can conclude that the total weight of S_2 is $\geq \frac{1}{4}$ the weight of T_2. Therefore, for any independent set T

$$\begin{aligned} weight(S) &= weight(S_1) + weight(S_2) && (1)\\ &\geq weight(T_1) + \frac{1}{4} weight(T_2) && (2)\\ &\geq \frac{1}{4}(weight(T_1) + weight(T_2)) && (3)\\ &= \frac{1}{4} weight(T) && (4) \end{aligned}$$

Therefore, 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 4: 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 2: 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.

Solution: Bin Packing

1.

We can first observe that by our assumption all weights are smaller than W, hence $\sum_{i=1}^{n} \log(w_i) \leq n \log W = \texttt{poly}(n + \log W)$. The question is therefore equivalent to showing that the time is polynomial in $n + \log W$.

We also observe that in each iteration, the number of bins decreases by 1. Therefore the number of iterations is bounded by n.

In each iteration, we need to check if there are two bins that can be merged. The number of pairs is bounded by $\binom{n}{2} = O(n^2)$ and for each pair we need to add up the two weights and then compare that number with W. Both addition and comparison are linear time operations in the bit-complexity of the numbers involved. Since at any point the weight of the bins in bounded by W, both the addition and the comparison take time $O(\log(W))$.

The body of the loop consists of one addition and eliminating one bin. The addition is again $O(\log W)$ and the eliminiation of a bin might be a data structure operation that will be upper bounded by $O(n)$ for any reasonable data structure.

Combining all steps the total time is upper bounded by

$$O(n \cdot (n^2 \cdot \log W + \log W + n)) = O((n + \log W)^4) = \texttt{poly}(n + \log W) \tag{5}$$

2.

Consider $W = 4$ and items with weights 2,2,3, and 1. The optimal solution uses two bins $(2 + 2, 3 + 1)$. However, the greedy solution might merge an item with weight two with the item of weight 1, resulting in a solution with 3 bins, namely $(2 + 1, 3, 2)$.

3.

Let O be the number of bins the optimal solution uses and let $o_1, o_2, \ldots, o_O$ be the weights of all nonempty bins in the optimal solution. Similarly, let G be the number of bins in the greedy solution and $g_1, \ldots, g_G$ be the weight of the bins.

Further let $T = \sum_{i=1}^{n} w_i$ be the sum of all the weights. We have

$$\sum_{i=1}^{O} o_i = T = \sum_{i=1}^{G} g_i \tag{6}$$

Furthermore, since $o_i \leq W$ for all i we have

$$T \leq O \cdot W \tag{7}$$

Let $l := \min_{i=1}^{G} g_i$ be the weight of the smallest bin in the greedy solution. We have two cases:

Case 1: ($l \geq W/2$) In this case we have $g_i \geq W/2$ for all i, hence

$$T \geq \frac{G \cdot W}{2} \tag{8}$$

Combining equations 7 with 8 we get

$$O \cdot W \geq \frac{G \cdot W}{2} \tag{9}$$

and therefore

$$\frac{O}{G} \geq \frac{1}{2} \tag{10}$$

Case 2: ($l \leq W/2$) We can assume that $G \geq 2$, as otherwise the greedy solution would only use one bin and is necessarily optimal. All bins other than the smallest have weight at least $W - l$, as otherwise we could still merge two bins. We therefore have

$$T \geq (G-1)(W-l) + l \tag{11}$$
$$= GW - Gl - W + 2l \tag{12}$$
$$\geq GW - \frac{GW}{2} - W + W \tag{13}$$
$$= \frac{GW}{2} \tag{14}$$

using $G \geq 2$ and $l \leq W/2$ for the penultimate step.

As in case 1, we can then conclude

$$\frac{O}{G} \geq \frac{1}{2} \tag{15}$$

Problem 5: 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 $1 - (1 - 1/k)^k > 1 - 1/e$.

Solution: Maximum coverage

For a set $S \subseteq U$, let weight(S) denote the total weight of the elements in the set S.

Let $O \subseteq U$ be the set of elements covered by the optimal solution, for i such that $0 \leq i \leq k$, let $G_i \subseteq U$ be the set of elements covered by the greedy solution after i iteration. Note that $G_0 = \emptyset$ and G_k is the greedy solution.

Consider the absolute difference of weights $\Delta_i :=$ weight(O) $-$ weight(G_i).

Claim 0.1. *For all i such that $1 \leq i \leq k$*

$$\Delta_i \leq \Delta_{i-1} - \frac{\Delta_{i-1}}{k} = (1 - 1/k)\Delta_{i-1} \tag{16}$$

Proof. We consider the set of elements that are covered by the optimal solution but not by the greedy solution after $i-1$ iterations. We define $S_{i-1} := O - G_{i-1}$ and have weight(S_{i-1}) $\geq \Delta_{i-1}$. From the existence of the optimal solution we know that there are k sets that cover O and therefore also cover S. At least one of those k sets must cover at least a (weighted) fraction of $1/k$. Hence there is at least one set in the input that covers at least $\frac{\Delta_{i-1}}{k}$ weight not covered so far. The claim then follows immediately. □

To finish the proof of the approximation ratio, we can use our claim inductively to get

$$\text{weight}(O) - \text{weight}(G_k) = \Delta_k \leq (1 - 1/k)^k \Delta_0 = (1 - 1/k)^k \text{weight}(O) \tag{17}$$

and hence

$$\text{weight}(G_k) \geq (1 - (1 - 1/k)^k) \text{weight}(O) \tag{18}$$