Algorithms: CSE 202 — Homework 4 Solutions

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 polynomial space.

Solution: Hamiltonian path (KT 10.3)

We solve the Hamiltonian Path Problem using dynamic programming in $O(n^2 2^n)$ time and $O(n2^n)$ space.

For $S \subseteq V - \{v_n\}$ and $u \in S$, let $H(S, u)$ be the predicate which is true if there is a path from u to v_n in G containing each vertex in $(V - S) \cup \{u\}$ exactly once and no other vertices. We define $H(S, u)$ to be false otherwise. G has a hamiltonian path if and only if $H(\{v_1\}, v_1)$ is true.

We provide the following recursive formulation for H. For all $S \subseteq V - \{v_n\}$ of size $n - 1$ and $u \in S$, we set $H(S, u)$ to true if there is an edge from u to v_n. For $S \subseteq V - \{v_n\}$ of size $n - 2$ or smaller and $u \in S$, we define

$$H(S, u) = \bigvee_{(u,v) \in E \text{ and } v \in (V - \{v_n\}) - S} H(S \cup \{v\}, v)$$

We argue the correctness of the recursive formulation as follows. If there is a path from u to v_n which includes all the vertices in $(V - S) \cup \{u\}$ exactly once and no other vertices, let v be the next vertex after u in such a path. Clearly, $v \neq v_n$ since such a path must have at least three vertices and v is not the last vertex in the path. In addition there must be a path from v to v_n that goes through the vertices in $(V - (S \cup \{v\})) \cup \{v\} = V - S$ exactly once and no other vertices, which is exactly the condition captured by $H(S \cup \{v\}, v)$. By considering all potential next vertices, we ensure that $H(S, u)$ is correctly computed.

For each S and u, the computation of $H(S, u)$ takes at most linear time. Since there are at most $n2^n$ possible values for S and u, we get $n^2 2^n$ as the time complexity. We need $n2^n$ space to store the table H.

We ask the students to write the necessary dynamic programming definitions, recursive formulations, and the code to find and output a hamiltonian path if there is one.

Space-efficient algorithm: We use the inclusion-exclusion principle to obtain a space-efficient algorithm. For $S \subseteq V - \{v_1, v_n\}$, let P_S be the number of paths from v_1 to v_n of length $n - 1$ that avoid the vertices in S. The paths are required neither to contain a a vertex exactly once nor to contain all the vertices in $V - S$. The only requirement is that the paths do not contain any vertices in S. We show that P_S can be computed in polynomial time using dynamic programming.

Let $P_S(u,k)$ for $u \in V$ and $0 \le k \le n-1$ be the number of paths of length k from u to v_n that avoid S. P_S is exactly $P_S(v_1, n-1)$.

$P_S(u,0)$ is 1 if $u = v_n$ and 0 otherwise. For $u \in V$ and $1 \le k \le n-1$, we define

$$P_S(u,k) = \sum_{(u,v)\in E} P_S(v, k-1).$$

For the proof of correctness of the recursive formulation, consider the set of paths of length $k \ge 1$ from u to v_n that avoid the vertices in S. Partition the set based on the the vertex v that immediately follows the first vertex u in the path. There must be such a vertex in each path since the paths have length at least 1. The number of paths where v immediately follows u is exactly $P_S(v, k-1)$ which is the number of paths from v to v_n of length $k-1$ that avoid S. We get $P_S(u,k)$ by summing over all such vertices v.

Using the above algorithm, we compute P_S for every $S \subseteq V$. We then use the following inclusion-exclusion formula to compute number of hamiltonian paths from v_1 to v_n in G.

$$\sum_{S\subseteq V} (-1)^{|S|} P_S$$

This sum can be calculated in polynomial space. We claim that there is a hamiltonian path from v_1 to v_n if and only if the sum is positive.

To prove the claim, consider a path p of length $n-1$ from v_1 to v_n. Let T be the set of vertices that are not included in the path p. p is hamiltonian if and only if $T = \varnothing$. We claim that p is counted once in each of P_S where $S \subseteq T$. Therefore, the effective contribution of the path p to the sum is exactly

$$\sum_{A\subseteq T} (-1)^{|A|} = \sum_{i=1}^{|T|} \binom{|T|}{i} (-1)^i = (1 + (-1))^{|T|}$$

which is positive if and only if $|T| = \varnothing$.

Problem 2: 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 $w(v)$,

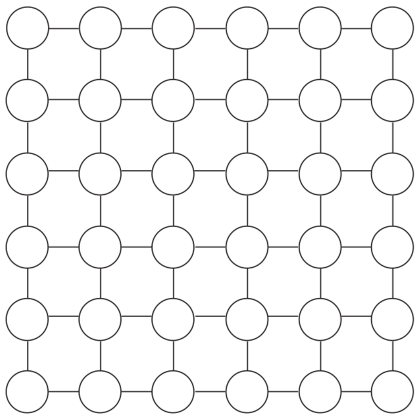

Figure 1: A grid graph

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)

For any set A of nodes, let $w(A) = \sum_{v\in A} w(v)$. Let S be the independent set returned by the *heaviest-first* greedy algorithm. Let T be any independent set of the graph.

1. Let $v \in T$. If $v \notin S$, there must be a neighbor v' of v in S whose selection in some iteration of the greedy algorithm must have removed v from further consideration. According to the selection criterion, the weight of v' is at least as much as the weight of v. Otherwise, greedy algorithm would have chosen v instead of v' during the iteration. We thus have that either $v \in S$ or it has a neighbor v' in S such that $w(v') \geq w(v)$.
2. Let $R = S \cap T$ be the set of nodes common between S and T. Let $S' = S - R$ and $T' = T - R$ be the sets of vertices that are exclusive to S and T respectively. S' and R are disjoint and $S = S' \cup R$. Similarly, T' and R are disjoint and $T = T' \cup R$.

   From the earlier argument, we know that for $v \in T'$, there is a neighbor v' of v such that $v' \in S'$ and $w(v) \leq w(v')$. We use v' in S' to *cover* for v in T'. Any such v' in S' need to cover at most four of its neighbors v in T' where $w(v) \leq w(v')$. From this covering argument, we get $w(T') \leq 4w(S')$. Therefore, for any independent set T, we have

$$\begin{aligned} w(S) &= w(R) + w(S') \\ &\geq w(R) + \frac{1}{4}w(T') \\ &\geq \frac{1}{4}(w(R) + w(T')) \\ &= \frac{1}{4}w(T) \end{aligned}$$

   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 3: 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 sorted in decreasing 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 approximation ratio for the greedy heuristic. A ratio of 7/6 or better would get full credit. A ratio worse than 7/6 might get partial credit.

Solution: Scheduling

a. Consider $3, 3, 2, 2, 2$. The optimal schedule is $(3, 3), (2, 2, 2)$, for a cost of 6, but the greedy schedule is $(3, 2, 2), (3, 2)$ for a cost of 7. This shows the greedy could be up to $7/6 * \mathit{optimal}$.

b. Let $a1 < a_2 < ... < a_n$ be the jobs in sorted order. Let $A = \sum a_i$ be the total work. Any schedule needs to have at least $A/2$ work on one side.

Let a_j be the last job scheduled on the larger side by the greedy algorithm. Let S be the total cost of that side, except for a_j, and let T be the total cost for the other side. Note that $S \leq T$, since at the time we scheduled a_j, we had a smaller amount of work on the side we scheduled it, and after that we only added jobs to the other side. So the cost of the greedy algorithm is $S + a_j \leq (1/2)(S + T) + a_j = 1/2(A - a_j) + a_j = 1/2(A + a_j)$. Note also that $a_j \leq A/j$, because there are j jobs at least as big. So we get that the ratio of greedy to optimal is at most $(1/2(A + A/j)/(1/2A) = 1 + 1/j$.

If j is large, this will give a good bound. However, if j is small, we can give a separate argument. If $j = 1$, that the larger side is just a_1, so the greedy cost is a_1. Since any schedule must perform a_1, this is also a lower bound for the cost of any schedule. So in this case, greedy must achieve the optimum.

It is not possible that $j = 2$, because we will either put a_3 on the same side as a_2, or there is no a_3, in which case a_2 is on the smaller side.

If $j = 3$, then again the greedy is optimal. That is because the greedy algorithm puts a_2 and a_3 together on the side not including a_1. So if a_3 is the last put on the larger side, the cost is $a_2 + a_3$. Now any schedule either puts a_2 and a_3 together or puts one of them with a_1, all of which mean one side costs at least $a_2 + a_3$. So the greedy schedule is optimal in this case.

So we are left with $j \geq 4$ as the only case when the greedy schedule might not be optimal. This gives us a maximum ratio of $1 + 1/j \leq 5/4$. This is not so far off from our lower bound of 7/6 from part a.

Problem 4: Maximum coverage

The maximum coverage problem is the following: Given a universe 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 weight of elements covered. Show that the obvious algorithm, of greedily picking the best set in each iteration until k sets are picked, achieves 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 $S_{o_1}, \ldots, S_{o_k}$ be an optimal collection of subsets that maximizes the sum of the weights of the covered elements. Let $O = \bigcup_{i=1}^{k} S_{o_i}$ be the set of elements covered by the optimal solution. For $1 \leq i \leq k$, let S_{g_i} be the set selected by the greedy algorithm during the i-th iteration. Let $G_0 = \varnothing$ and $G_i = \bigcup_{j=1}^{i} S_{g_j}$ be the set of elements covered by the greedy solution after the i-th iteration for $1 \leq i \leq k$. Note that G_k is the set of elements covered by the greedy solution.

Consider the difference of weights $\Delta_i :=$ weight(O) − weight(G_i). $\Delta_i \geq 0$ since O is the set of elements covered by the optimal solution. We argue that Δ_i is decreasing at a rate of $(1 - 1/k)$ as i increases.

Claim 1. *For $1 \leq i \leq k$*

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

Proof. We consider the set of elements that are covered by the optimal solution but not by the greedy solution after $i-1$ iterations and let $T_{i-1} := O - G_{i-1}$. We have weight$(T_{i-1}) \geq \Delta_{i-1}$. Since $T_{i-1} \subseteq O$ and O is the union of the k sets S_{o_j}, there must exist a set S_{o_j} for $1 \leq j \leq k$ such that weight$(S_{o_j} \cap T_{i-1}) \geq$ weight$(T_{i-1})/k$. During the i-the iteration, the greedy algorithm has the opportunity to select S_{o_j} to improve the coverage by at least weight$(T_{i-1})/k \geq \Delta_{i-1}/k$. Since the greedy algorithm in each iteration selects a set that improves the coverage as much as possible, we get that the weight of G_i is more than that of G_{i-1} by at least Δ_{i-1}/k. Hence, $\Delta_i \leq \Delta_{i-1} - \frac{\Delta_{i-1}}{k} = (1-1/k)\Delta_{i-1}$. □

To show we get the desired approximation ratio, we use our claim repeatedly to get

$$\text{weight}(O) - \text{weight}(G_k) = \Delta_k \leq (1-1/k)^k \Delta_0 = (1-1/k)^k \text{weight}(O) \tag{2}$$

and hence

$$\text{weight}(G_k) \geq (1 - (1-1/k)^k)\text{weight}(O) \geq (1 - 1/e)\text{weight}(O) \tag{3}$$