James V. Salvan Homework 4 CSE 202:B00

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.

Key Idea:

If we know a Hamiltonian Path exists up for some subsets of vertices in V, then a Hamiltonian Path will necessarily exists if ∀ of these subsets with a Hamiltonian Path, we can reach another unaccounted for vertex from the last vertex selected in the subset. → Dynamic Programming

Definitions:

Let S be a subset of vertices in G not including the final node on the path

- $S \subseteq V - \{v_n\}$

Define $f(S, v)$ correspond to whether a Hamiltonian path exists to v_n from v

- More specifically $f(S,v)$ is TRUE if v, an unvisited vertex can get to v_n by visiting all nodes in $(V - S)$ no more than once. Otherwise $f(S,v)$ is FALSE, as we cannot get to v_n from $(V-S) \cup \{v\}$.

Algorithm:

Starting with an S of size $n-1$ (recall S has a size of at most $n-1$), check if for a vertex V, there is an edge from V to V_n. This corresponds to $f(\text{all vertices but } V_n \text{ \& } V, V)$. If this edge exists then this is TRUE, otherwise store FALSE. For all S of size $\geq n-2$, $f(S, V) = \text{OR } f(S \cup \{v\}, v) \; \forall \; v \in V - \{V_n\} - S$ vertices and all (V, v) edges. Conceptually this equates to checking if we can extend the path for a vertex V, by checking if any of the vertices to which it is connected has a Hamiltonian path. If so, since V is not accounted for, we can extend the Hamiltonian path. Finally as V_1 is defined to be the starting vertex, the existence of a Hamiltonian Path in G is $f(\{V_1\}, V_1)$.

Time and Space Complexity:

For each vertex in G, we must check up to $n \cdot 2^n$ other vertices in the worst case, leading to a time complexity of $n^2 \cdot 2^n$. As we only need to store a TRUE or FALSE for each vertex (for all 2^n instances), the space complexity is $O(n2^n)$

Proof of Correctness:

Base Case: Size $G \leq 2$

In the case of the # of vertices in $G = 0$, return TRUE or FALSE as desired, depending on corner case definition. If # of vertices in $G = 1$, then return TRUE as $V_1 = V_n$ (another corner case definition). If the # of vertices in $G = 2$, then either V_1 & $V_{2=n}$ are connected, or they aren't. Thus $f(\{V_1\}, V_1)$ checks the existence of such an edge & returns TRUE or FALSE as desired.

Inductive Hypothesis: For size $|V|$ (the # of vertices in G) > 2, assume $\forall\ 1 < i < n$, $f(S, V_i)$ returns the existence of a Hamiltonian path in accordance with the definition of f.

Inductive Step:

In attempting to extend the path of size $n - 1 - |S|$ by 1 (i.e., including V_i), there must necessarily exist a path for at least one $V - S \to V_i$ that is Hamiltonian. Otherwise no Hamiltonian can exist. As $f(S, V)$ employs the OR operation and searches the existance of a TRUE in all of its neighbors, then necessarily as V is unvisited, then we can extend the path. If $\forall$ paths to which V_i is connected are non-Hamiltonian, then adding V_i is also necessarily FALSE.

Improving to Polynomial Space:

This approach can be modified using the Inclusion-Exclusion Principle to yield a solution in $O(2^n \cdot p(n))$ & Polynomial space.

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 $w(v)$,

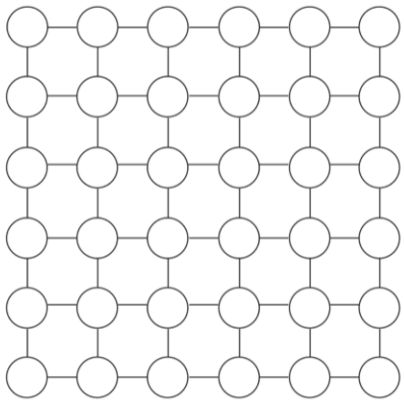

Figure 1: A grid graph

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.

Problem 2.1

Observe first the case when $v \in T$ & $v \notin S$. For this to be the case then v must be connected to some v' in S (i.e., an edge exists (v, v')), otherwise v would have been selected. Specifically the "heaviest-first" greedy algorithm keeps selecting nodes in G (selecting the maximum weight node remaining at each iteration) until G is empty. Thus the only way that $v \in T$ & $v \in S$ is if $\exists$ a $v' \in S$ for which there is an edge (v, v'), otherwise the above algorithm would have selected it.

Similarly, the "heaviest-first" nature of this algorithm will only omit v from consideration if $w(v') \geq w(v)$, otherwise it would have selected v and omitted v' (i.e., deleted all neighbors of v which necessarily includes v' as elucidated above).

Finally observe the only other case for a node in T: $v \in T$ & $v \in S$. That is a vertex in T can either not be in S & satisfy the conditions elaborated above, or a vertex is in both T & S.

Therefore for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ such that $w(v) \leq w(v')$ and (v, v') is an edge of G.

Problem 2.2

Key Idea:

Can we separate the vertices selected by the greedy approach and the Optimal solution into those in both and those that are disjoint? If so, we can leverage the properties elucidated in 2.1 and the grid structure of the graph (i.e., the fact that every vertex in G is connected to at most 4 other vertices) to derive a ratio of $\frac{\text{Greedy}}{\text{Optimal}} \geq \frac{1}{4}$.

Definitions:

$w(S)$ → the total weights of all elements in an independent set S.

α → the approximation ratio. Define this as $\frac{\text{Optimal}}{\text{Greedy}}$ here as this is a maximization problem and want consistency in approaches (i.e., $\alpha \geq 1$).

Optimal → The maximum total weights of any independent set in the grid graph G.

Greedy → The total weights of the independent set selected by the "heaviest-first" greedy algorithm.

S → The independent set selected by the "heaviest-first" greedy algorithm

OPT → The independent set in G with the maximum total weight.

Approximation Ratio:

Define I to be the intersection of S and OPT. That is

$$I = S \cap OPT$$

Now we can rewrite S & OPT in terms of their shared and disjoint elements.

$S = I + S'$ ← S' is all remaining elements in S (i.e., those non-overlapping with OPT)

$OPT = I + OPT'$ ← OPT' is all remaining elements in OPT (i.e., those non-overlapping with S)

By definition $Greedy = w(S) = w(I) + w(S')$

Now observe that for each element/vertex is S', there are at most 4 vertices in OPT' not selected by the nature of the grid & independent set constraints. Note each of the 4 vertices in OPT' must have $\leq$ weight than the corresponding v' in S' based on the reasoning in 2.1 (in the case of distinct weights, as given by the problem $\leq$ can be replaced by $<$). Thus

$$w(S) \geq \frac{1}{4} w(OPT')$$

Plugging this in above gives:

$$Greedy = w(I) + w(S') \geq w(I) + \frac{1}{4} w(OPT')$$

Note now that if can get in the form of
$K \cdot (w(I) + w(OPT'))$ this will $= K \cdot$ Optimal

To do this we can multiply $w(I)$ by $\frac{1}{4}$ as

$$w(I) \geq \frac{1}{4} w(I)$$

Thus

$$w(I) + \frac{1}{4} w(OPT') \geq \frac{1}{4} w(I) + \frac{1}{4} w(OPT')$$

$$= \frac{1}{4}(w(I) + w(OPT'))$$

$$= \frac{1}{4} \cdot \text{Optimal}$$

Thus $\frac{\text{Greedy}}{\text{Optimal}} \geq \frac{1}{4}$ as desired

Putting this in terms of α (i.e., $\frac{\text{Optimal}}{\text{Greedy}}$) to ensure values ≥ 1 and an upper bound for the deviation ratio (as opposed to a lower bound when α is defined in a fractional manner) gives $\frac{\text{Optimal}}{\text{Greedy}} \leq 4$ as desired.

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.

Problem 3.1:

Consider the case when $n=5$ with the following (sorted in decreasing order) time requirements.

6, 5, 5, 3, 3

Iterating through a greedy approach that selects the longest time requirement job (thus the sorting of jobs in decreasing order) to the machine with the least total work so far would yield:

machine 1: 6 3 3

machine 2: 5 5

Thus the greedy approach here yields $\max(\text{machine 1}, \text{machine 2})$

$= \max(12, 10) = 12$

However observe that this greedy assignment is not optimal. Specifically observe the following assignment:

machine 1: 6 5

machine 2: 5 3 3

Which yields a total time to complete all jobs

$\max(\text{machine 1}, \text{machine 2}) = \max(11, 11) = 11$

Thus as $12 > 11$, we observe an example where this greedy heuristic is not optimal.

Problem 3.2:

Key Idea:

This problem can be conceptualized as the makespan problem with m, the number of machines here equal to 2.

Definitions:

T_G → The total time to complete all jobs, obtained by the greedy solution

T_{OPT} → The total time to complete all jobs, obtained by the optimal solution.

α → The approximation ratio $\frac{T_G}{T_{OPT}} \leq \alpha$, for which aim is to show $\alpha = \frac{7}{6}$

Approximation Ratio:

By definition know that the final job assigned to the machine with the maximum time to completion by the greedy solution contributed to the time. Define this as job j with $1 \leq j \leq n$. Thus by the greedy method know the following must hold true:

$$T_G - t_j \leq \frac{\sum_{i=1}^{j-1} t_i}{2}$$

⟸ After $j-1$ jobs the machine before j is assigned the load/time of the machine to which it is assigned must be $\leq$ the average current load.

We also know that T_{OPT} cannot exceed the average time & thus establishes a lower bound:

$$T_{OPT} \geq \frac{\sum_{i=1}^{n} t_i}{2}$$

⟸ Define $\frac{\sum_{i=1}^{n} t_i}{2}$ as L_{AVG}

Moreover because the jobs are sorted in decreasing order we know that t_j cannot exceed the average job time at point j. Specifically that is:

$$t_j \leq \frac{\sum_{i=1}^{j} t_i}{j}$$

Using these relationships we can formulate a relationship b/w T_G & T_{OPT}.

$$T_G - t_j \le \frac{\sum_{i=1}^{j-1} t_i}{2}$$ ⇐ add t_j to both sides, on the right side partition it into $\frac{t_j}{2} + \frac{t_j}{2}$

$$T_G \le \frac{\sum_{i=1}^{j} t_i}{2} + \frac{t_j}{2}$$

→ Note $\frac{\sum_{i=1}^{j} t_i}{2} \le L_{AVG} \le T_{OPT}$ so can substitute that above here

$$T_G \le T_{OPT} + \frac{t_j}{2}$$ ⇐ substitute t_j here w/ $\frac{\sum_{i=1}^{j} t_i}{j}$ as defined above

$$T_G \le T_{OPT} + \frac{1}{j} \cdot \frac{1}{2} \cdot \sum_{i=1}^{j} t_i$$

- Again note we have $\frac{\sum_{i=1}^{j} t_i}{2}$ and thus as did above can replace w/ L_{OPT}

$$T_G \le T_{OPT} + \frac{T_{OPT}}{j} = T_{OPT}\left(1 + \frac{1}{j}\right)$$

Thus we have $\frac{T_G}{T_{OPT}} \le 1 + \frac{1}{j}$ ← To derive a tighter numerical $\frac{7}{6}$ bound go case-by-case for values of $n = 1 \to 5$

$n=1$:

Observe that $n < m = 2$ here and thus with $n=1$, both T_{OPT} & $T_G = t_1$. Thus at $n=1$ greedy is optimal.

$n=2$:

Again observe that at $n=2$ greedy is optimal. Specifically by allocating 1 job to each machine ($n=m$) then $T_G = \max(t_1, t_2) = t_1$. Similarly the optimal solution can never achieve a $T_{OPT} < t_1$ because t_1 is the longest (i.e., $t_1 \geq t_2$) time job. Thus T_{OPT} here is also bounded by t_1. Given this @ $n=2$ greedy is optimal.

$n=3$:

Note in the instance when $n=3$ the greedy approach will yield the following allocation:

machine 1: t_1

machine 2: t_2, t_3

This occurs as $t_2 \leq t_1$ and thus the greedy approach is bounded by $\max(t_1, t_2+t_3)$ which is the same as the optimal solution here. To observe this, first recognize the case when all 3 jobs are assigned to 1 machine is non-optimal as we can always reduce the load by moving the maximal load job (i.e., t_1) to the unused machine.

Thus this leaves two cases for OPT for 2 jobs to one machine.

$$OPT = \begin{cases} t_1 + t_3 \\ t_2 + t_3 \end{cases}$$

However note $t_1 \geq t_2$, so we can effectively select $t_2 + t_3$ and t_1 as an optimal partitioning yielding $T_{OPT} = \max(t_1, t_2 + t_3)$ the same as the greedy approach.

$n = 4$:

For $n = 4$, examine the possibilities of OPT. First as in $n = 3$ observe that an optimal solution cannot assign all $n = 4$ jobs to one machine and remain optimal (i.e., we can always reduce T_{OPT} than by moving t_1 to the unused machine). Thus an optimal solution must have a 1-3 or 2-2 job allocation.

Examine the 1-3 allocation first. Observe that if this 1-3 solution is optimal, then t_1 will always be the lone job. This follows that if it weren't (i.e., t_2, t_3, or t_4 were on their own), then we would always be able to generate an improved or equivalent optimal solution by swapping t_1 with whatever $t_{i>1}$ was on its own and yield a reduction. Specifically the machine w 3 jobs and t_1 was lower bounded by t_1, so moving t_1 onto its own machine does not change the global max since t_1 needs to be on at least one machine. Now observe that the exchange reduces the runtime of the 3-job machine by $t_1 - t_{i>1}$.

Thus in this case the Optimal allocation is

machine 1: t_1

machine 2: t_2, t_3, t_4

yielding $T_{OPT} = \max(t_1, t_2+t_3+t_4)$. Now note that when an Optimal solution is a 1-3 distribution, greedy will always return the same thing. Specifically observe that greedy is optimal at $n=3$. For a 1-3 to be optimal, then necessarily $t_2+t_3 \leq t_1$ & thus greedy will assign t_4 to machine 2 (in the case of equivalence $t_1 = t_2+t_3$ observe that the maximum remains the same then as $t_1+t_4 = t_2+t_3+t_4$ & thus the max load would remain the same in either allocation). Thus we observe that for a 1-3 allocation greedy is Optimal.

Now consider a 2-2 allocation. Imagine OPT returns the following allocation

machine 1: t_1, t_2

machine 2: t_3, t_4

Observe that we can always improve the maximal time by swapping t_2 & t_3. Specifically observe that $t_1+t_2 \geq t_1+t_3$ and $t_1+t_2 \geq t_2+t_4$ by the ordering in decreasing time. Thus we now have the following allocation:

machine 1: t_1, t_3

machine 2: t_2, t_4

Now observe that as $t_1 \geq t_2$ and $t_3 \geq t_4$ we can exchange t_3 & t_4 and obtain an equivalent or better solution. This yields

machine 1: t_1, t_4

machine 2: t_2, t_3

Note that this equates to what is returned by the greedy solution. That is if $t_2 + t_3 \geq t_1$ the greedy approach will assign t_4 to machine 1. If $t_2 + t_3 < t_1$ then a 2-2 cannot be optimal as we could move t_4 to machine 2. Thus here we observe $T_{OPT} = T_G = \max(t_1 + t_4, t_2 + t_3)$.

Thus in both cases observe once again that the greedy approach returns the optimal solution.

n = 5:

Observe here that greedy cannot always be optimal (as illustrated by the example given in 3.1 above). The goal then focuses on whether we can find a tighter bound than $1 + \frac{1}{j=5} = \frac{6}{5}$

As before observe the optimal solution can never return all n jobs on one machine as we can always reduce the maximum time by moving t_1 to the unused machine.

Thus we have the instances of a 1-4 allocation and a 2-3 allocation. For the 1-4 allocation to be optimal observe that as was the case for $n=4$ above, t_1 must be the 1 (i.e., lone) job. If not we could always reduce the maximum load by moving it onto its own machine. Observe in this case, greedy will also return the same solution i.e.:

machine 1: t_1

machine 2: t_2, t_3, t_4, t_5

As $t_1 \geq t_2 + t_3 + t_4$. Thus in the 1-4 job distribution, greedy is optimal.

Now return to when greedy is not always optimal for a 2-3 job distribution. However, observe that in this case at least one machine has 3 jobs assigned to it. In such a case $t_j \leq \frac{T_{OPT}}{3}$. Specifically t_j cannot exceed the average load of the 3-job-assigned machine, regardless of whether machine 1 or machine 2 are the rate-limiting (i.e., max time) factor. This again follows by sorting in decreasing order of time. Thus we can provide a narrower bound by instead of replacing t_j with $\frac{\sum_{i=1}^{j} t_i}{j}$ by instead replacing it with $\frac{T_{OPT}}{3}$.

As derived above here:

$$T_G \leq T_{OPT} + \frac{t_j}{2}$$ ← replace t_j here with narrower bound

$$T_G \leq T_{OPT} + \frac{1}{2} \cdot \frac{T_{OPT}}{3} = T_{OPT} + \frac{T_{OPT}}{6} = T_{OPT} \cdot \frac{7}{6}$$

Thus here $\frac{T_G}{T_{OPT}} \leq \frac{7}{6}$ as desired.

$j \geq 6$:

Note $j \geq 6$ can only occur when $n \geq 6$. Obj $j < 6$ we can fall back on the reasoning of $n < 6$ cases provided above. However when $n \geq 6 + j \geq 6$ note that the general ratio of α equal to $\frac{T_G}{T_{OPT}} \leq 1 + \frac{1}{j}$ gives an equivalent or narrower bound than $\frac{7}{6}$ by the fact $j \geq 6$.

Thus, given the above analysis can conclude an approximation ratio α of $\frac{7}{6}$ as desired.

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 $(1 - (1 - 1/k)^k) > (1 - 1/e)$.

Key Idea:

This problem can be conceptualized as how much gain the greedy solution gain on the optimal solution at each step from steps $1 \to K$.

Definitions:

- $\left(1 - \frac{1}{K}\right)^K \leq \frac{1}{e}$
- GR_j = The union of j subsets in U chosen by the greedy algorithm
  - Specifically $GR_j = \bigcup_{i=1}^{j} S_{g_i}$ where S_{g_i} is the i^{th} subset selected by the greedy algorithm
- OS = The optimal solution. That is the union of all K subsets selected by the optimal solution
  - More formally $OS = \bigcup_{i=1}^{K} S_{o_i}$ where S_{o_i} is the i^{th} subset selected by the optimal solution.
- $W(GR_j)$ = The weight of elements covered by the greedy solution of $j \leq K$ subsets
- $W(OS)$ = The weight of elements covered by the optimal solution
- $\triangle_j = W(OS) - W(GR_j)$

Approximation Ratio:

First observe that $\Delta_j \geq 0$ (i.e., the greedy solution can never select a collection of subsets with greater weight than an optimal solution). Moreover observe $\Delta_j \leq W(OS - GR_j)$, the weights of all elements unique to OS (i.e., those subsets in OS but not in GR_j).

Next note that $OS - GR_j$ is a subset of OS which is composed of k total subsets by definition. Thus at any step j, GR_{j+1} has the potential to select one of these OS subsets not in $OS - GR_j$. Observe then that one of these subsets must contribute at least the average weight remaining. Moreover OS is the selection of k-subsets, so one of the unselected OS subsets must have a weight contribution of:

$$W(S_{o_i} \cap OS - GR_j) \geq \frac{W(OS - GR_j)}{k}$$

$S_{o_i} \cap OS - GR_j$: Common elements in these 2 sets

Thus greedy can improve its weight relative to the optimal solution by at least $\frac{W(OS - GR_j)}{k} \geq \frac{\Delta_j}{k}$

Thus between j and $j+1$ observe then:

$$\Delta_{j+1} \leq \Delta_j - \frac{\Delta_j}{k} = \Delta_j\left(1 - \frac{1}{k}\right)$$

Finally we can extrapolate Δ_k from Δ_0 (when greedy has not selected any subsets).

$$\Delta_0 = W(OS)$$

$$\Delta_k \leq \left(1 - \frac{1}{k}\right)^k W(OS)$$

$$\Delta_k = W(OS) - W(GR_k)$$ by definition. Substitute this here

$$W(OS) - W(GR_k) \leq \left(1 - \frac{1}{k}\right)^k W(OS)$$

$$W(GR_k) \geq W(OS)\left[1 - \left(1 - \frac{1}{k}\right)^k\right]$$

$$\frac{W(GR_k)}{W(OS)} \geq 1 - \underbrace{\left(1 - \frac{1}{k}\right)^k}$$

by definition 1 this is $\leq \frac{1}{e}$ → substitute here

Thus $\frac{W(GR_k)}{W(OS)} \geq 1 - \frac{1}{e}$ as desired.