# Homework 4

Wenjun Zhang

A53218995

**Problem 1: Hamiltonian path (KT 10. 3)**

---

**Algorithm description:**

We are asked to show that the Hamiltonian Path Problem can be solved in time $O(2^n \cdot p(n))$. We can apply an algorithm based on Dynamic Programming.

We have a start node $v_1$ and we define $DP(S', v)$ to represent whether there is a path from $v_1$ to $v$ which passes the nodes in $S$ exactly once. If there is such path, $DP(S, v) = 1$. Otherwise, $DP(S', v) = 0$. The base case is $DP(\{v_1\}, v_1)$.Next, the recurrence can be expressed as:

$$DP(S', v) = \begin{cases} 1 & \text{if } DP(S' - v, u) = 1 \text{ and edge } (u, v) \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

At the end, if $DP(S, v_n) = 1$, then there is a Hamiltonian path from $v_1$ to $v_n$. Otherwise, there is not.

**Proof of correctness:**

We first define $DP(S', v)$ as whether there is a path from $v_1$ to $v$ which passes the nodes in $S$ exactly once. The base case is $DP(\{v_1\}, v_1) = 1$, which means $v_1$ can reach to $v_1$ by passing $v_1$.

Next, we are going to prove the recurrence

$$DP(S', v) = \begin{cases} 1 & \text{if } DP(S' - v, u) = 1 \text{ and edge } (u, v) \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

For each vertex set $S'$ and node $v$, a Hamiltonian path exists if and only if $v$ has a neighbor $u$ such that a Hamiltonian path exists for $S' - v$ and $u$. If $DP(S' - v, u) = 0$, there is no such Hamiltonian path from $v_1$ to $u$ that passes all the nodes once in $S' - v$, let alone a Hamiltonian path from $v_1$ to $v$ in vertex set $S'$. If $DP(S' - v, u) = 1$ and there is an edge $(u, v)$, then the Hamiltonian path from $v_1$ to $u$ plus the edge $(u, v)$ will be the Hamiltonian path from $v_1$ to $u$ in vertex set $S'$. Additionally, $DP(S' - v, u)$ can be looked up from the previous computation by Dynamic Programming.

At the end, if $DP(S, v_n) = 1$, it means there is a Hamiltonian path from $v_1$ to $v_n$ that passes all the nodes in $S$, where $v_1$ being the start and $v_n$ being the end.

**Time complexity:**

There are $C_{n-1}^{1} + C_{n-1}^{2} + \cdots + C_{n-1}^{n-1} = O(2^n)$ sets need to consider, and for each set, we need to consider $O(n)$ nodes as the end and to compute $DP(S', v)$ we need to traverse $O(n)$ neighbors of $v$. Therefore, the time complexity of the algorithm is

$$T(n) = (2^n n^2)$$

**Space complexity:**

To compute $DP(S', v)$, we need to consider $O(n)$ neighbors of $v$ under set $S' - v$. We compute a vertex set one at a time, so the space can be reused. Therefore, the space complexity is $O(n)$.

**Problem 2: MaxCut for trees (KT 10. 9)**

**Algorithm description:**

We are asked to find the maxcut of a binary tree in polynomial-time. We can design an algorithm based on Dynamic Programming method.

Suppose the $r$ is the root of the binary tree, and for a node $u$, $T_u$ represents the subtree whose root is $u$, and the number of nodes in this subtree is $n_u$. We also mark the weight of edge $(u, v)$ as $w_{uv}$. We define $DP(u, k)$ as the maxcut of the subtree $T_u$, where the cut set containing the root $u$ has $k$ nodes.

The base case we build are, for the nodes $u$ which don't have child, we define $DP(u, 1) = 0$. Then for the recurrence, there are two situations:

1. If the node $u$ has only one child $v$, the recurrence is

$$DP(u, k) = \max\left(DP(v, k-1), DP(v, n_v - k + 1) + w_{uv}\right)$$

2. If the node $u$ has two children $v$ and $d$, suppose the cut set containing $u$ has $m_v$ nodes from the subtree $T_v$ and $m_d$ nodes from the subtree $T_d$, the recurrence is

$$DP(v, k) = max_{0 \le m_v \le k-1, m_d = k-1-m_v}\{\max\left(DP(v, m_v), Dp(v, n_v - m_v) + w_{uv}\right) + \max(DP(d, m_d), DP(d, n_d - m_d) + w_{ud})\}$$

Finally $DP(r, n/2)$ is the result we want.

**Proof of correctness:**

We first define $DP(u, k)$ as the maxcut of the subtree $T_u$, where the separated set involving the root $u$ has $k$ nodes. The base cases are set for the leaves, since they don't have any child, the subtrees whose roots are leaves have only one node and they don't have any edges, so for leaves $DP(u, 1) = 0$.

Next, we are going to prove the recurrence. Firstly, when the node $u$ only has one child $v$, there are two cases: the child $v$ and root $u$ belong to the same set, or they are separated. When they are in the same set, we can just ignore the edge $(u, v)$ and the maxcut value is the same as $DP(v, k-1)$. When they are separated, we extract $k-1$ nodes from the subtree $T_v$, so the maxcut value is $DP(v, n_v - k + 1) + w_{uv}$. Therefore, if the node $u$ only has one child, the recurrence is

$$DP(u, k) = \max\left(DP(v, k-1), DP(v, n_v - k + 1) + w_{uv}\right)$$

Secondly, when the node $u$ has two children $v$ and $d$, we need to consider whether $v$ or $d$ is in the same set with $u$. The size of the set containing $u$ is $k$, so $m_v + m_d = k - 1$. Similarly, the maxcut value of the root $u$ and the subtree $T_v$ is

max $(DP(v, m_v), Dp(v, n_v - m_v) + w_{uv})$, and the maxcut value of the root $u$ and the subtree $T_d$ is $\max(DP(d, m_d), DP(d, n_d - m_d) + w_{ud})$. We need to consider all possible $(m_v, m_d)$ pairs, so combining the recurrence is

$$DP(v,k) = max_{0 \le m_v \le k-1, m_d = k-1-m_v}\{\max(DP(v, m_v), Dp(v, n_v - m_v) + w_{uv}) + \max(DP(d, m_d), DP(d, n_d - m_d) + w_{ud})\}$$

Since $DP(v,k)$ is a maximizer, and $DP(r, n/2)$ represents the maxcut value of the binary tree whose root is $r$ and the set containing $r$ is with size $k$, which corresponds to the problem's requirements.

**Time complexity:**

There are $n$ nodes and we set the higher bound of $k$ is $n/2$, here it's $O(n^2)$. For the recurrence, we need to compare $O(n)$ times. Hence, the time complexity of the algorithm is

$$T(n) = (n^3)$$

**Problem 3: Heaviest first (KT 11. 10)**

**Part 1:**

According to the "heaviest-first" greedy algorithm, a node can either be included in the set $S$ or it's deleted from the graph $G$. Therefore, for each node $v \in T$, either $v \in S$, or it's deleted from the G in the greedy algorithm. Since from the greedy algorithm, a node can only be deleted when a neighbor of it, whose weight is no smaller, is included in $S$. If $v$ is deleted, it means a node $v'$, which is a neighbor of $v$, must be included in $S$, and $w(v') \geq w(v)$.

**Part 2:**

Suppose the optimal set of the problem is $O$. According to part 1, 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 in $G$. Since $T$ is an arbitrary set, for each node $v \in O$, 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 in $G$. Since a node can at most have four neighbors, for a node $v' \in S$ and $v' \notin O$, at most four $v_i$ such that $(v_i, v')$ exists in $G$ are included in $O$, which can be expressed as

$$4w(v') \geq \sum_{i=1}^{4} v_i$$

Suppose there are two parts $S_1$ and $S_2$ where $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$ and $S_1 \subseteq O$. As mentioned above, we have

$$4w(S_2) \geq w(O - S_1)$$

If $S_1 = \emptyset$, we have

$$w(S) = w(S_2) \geq \frac{1}{4} w(O - S_1) = \frac{1}{4} w(O)$$

If $S_1 \neq \emptyset$, we have

$$w(S) = w(S_1) + w(S_2) \geq w(S_1) + \frac{1}{4} w(O - S_1) = \frac{1}{4} w(O) + \frac{3}{4} w(S_1)$$

Since the weights are non-negative, we have $w(S) > \frac{1}{4} w(O)$.

Hence, we prove that the "heaviest-first" greedy algorithm returns an independent set with weight at least $\frac{1}{4}$ times the optimal weight.

**Problem 4: Bin packing**

**Part 1:**

We are asked to prove that the merging heuristic always terminates in time polynomial in the size of the input.

We can compute the time complexity by clarifying several things below:

1. There are $n$ items in total, so the maximum times of merging is $n-1$.
2. We apply bit operation for summation and comparison. To add $w_i$ to $w_j$, we at most need $O(\log \max(w_i, w_j))$ bit operation. For each $w_i$ it appears at most $n-1$ times to sum with others.
3. After summation, we need to compare the sum with the bin volume W. There are at most $C_2^n = (n^2-n)/2$ combinations, where each will take $O(\log W)$ by applying bit comparison.

From what's discussed above, the time complexity of the algorithm is

$$O\left((n-1)\left((n-1)\sum_{i=1}^{n}\log w_i + \frac{n^2-n}{2}\log W\right)\right)$$

After simplification, the time complexity is

$$O(n^2\sum_{i=1}^{n}\log w_i + n^3\log W)$$

which is the polynomial in the size of the input $n + \log W + \sum_{i=1}^{n}\log w_i$.

**Part 2:**

Here is a counter example:

Suppose we have 4 items with weights 1,2,6,6 and the bin volume is 8. According to the merging heuristic algorithm, the first 2 items may be merged and the solution is 3,6,6, which takes 3 bins. But the optimal solution is to merge the first and the third bins and merge the second and the fourth bins, and the result is 7,8, which takes 2 bins.

Hence, the packing returned by the merging heuristic may not use the minimum number of bins.

**Part 3:**

Suppose the solution we obtain from merging heuristic is $S$ which uses $m$ bins, and the optimal solution is $O$, which uses $n$ bins. We are asked to prove that

$$m \leq 2n$$

Firstly, we can easily see that $n \geq \left\lceil \frac{\sum_{i=1}^{n} w_i}{W} \right\rceil$, because a bin can store weight no more than $W$, the minimum number of bins required is $\left\lceil \frac{\sum_{i=1}^{n} w_i}{W} \right\rceil$.

Secondly, we are going to prove that there is at most 1 bin in $S$ that store weights less than or equal to $W/2$. This is actually quite obvious since if there exist two such bins, according to the merging heuristic algorithm these two bins would have been merged.

Therefore, if there is no bin with weight less than or equal to $W/2$ in $S$,

$$m \leq \left\lceil \frac{\sum_{i=1}^{n} w_i}{W/2} \right\rceil \leq 2n$$

If there is one bin with weight less than or equal to $W/2$ in $S$, then we have

$$\sum_{i=1}^{n} w_i - (m-1)\frac{W}{2} > 0$$

Rewrite this equation, we have $\frac{\sum_{i=1}^{n} w_i}{W} > \frac{1}{2}(m-1)$. Therefore, $n \geq \left\lceil \frac{\sum_{i=1}^{n} w_i}{W} \right\rceil > \frac{1}{2}m$. Since $m$ and $n$ are integers, $2n \geq m$.

Hence, we prove that the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

**Problem 5: Maximum coverage**

Suppose the optimal collection covers a set of elements $O$, and its corresponding weights are $w(O)$. At each iteration, the set of covered elements from the Greedy algorithm is $G_i$ and its corresponding weight is $w(G_i)$. Specifically, $w(G_0) = 0$. Also, at each iteration, we define the weight we gain additionally to be $w_i$.

Firstly, we are going to prove that $w_i \geq \frac{1}{k}(w(O) - w(G_{i-1}))$. Since the optimal solution has $k$ sets, we can always choose up to $k$ subsets from the remaining subsets and plus and elements we already cover $G_i$ to make the total weight equal to $w(O)$. And this means there will always be a remaining subset that can add additional weight no less than $\frac{1}{k}(w(O) - w(G_{i-1}))$. According to greedy algorithm, we choose the best subset that can provide most additional weight, so $w_i \geq \frac{1}{k}(w(O) - w(G_{i-1}))$.

Therefore, we have

$$\begin{aligned} w(G_i) &= w(G_{i-1}) + w_i \\ &\geq w(G_{i-1}) + \frac{1}{k}\big(w(O) - w(G_{i-1})\big) \\ &= \frac{1}{k}w(O) + (1 - \frac{1}{k})w(G_i) \end{aligned}$$

By iteration, we have

$$w(G_k) \geq \frac{1}{k}w(O)\left\{1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \cdots + \left(1 - \frac{1}{k}\right)^{k-1}\right\}$$

$$= \frac{1}{k}w(O)\frac{1 - \left(1 - \frac{1}{k}\right)^k}{1 - \left(1 - \frac{1}{k}\right)}$$

$$= w(O)(1 - \left(1 - \frac{1}{k}\right)^k)$$

Rewrite this we get

$$\frac{w(G_k)}{w(O)} \geq 1 - (1 - \frac{1}{k})^k$$

Next, what's left to do to prove $1 - \left(1 - \frac{1}{k}\right)^k > 1 - 1/e$, or equivalently, $(1 - \frac{1}{k})^k < 1/e$. Taking a log form on both sides, we have $k\ln(1 - \frac{1}{k}) < -1$. We can rewrite this as $\ln(1 - 1/k) < -1/k$.

To prove this, we define a function $f(x) = \ln(1-x) + x$. It's easy to know that $f(0) = 0$ and $f'(x) = -\frac{1}{1-x} + 1$, where $f'(x) < 0$ for $0 < x < 1$. So $f(x)$ is monotonically decreasing for $0 < x < 1$, and $f(x) < 0$ for $x$ in this range. So when $k > 1$, $\ln(1 - 1/k) < -1/k$. Specifically, when $k = 1$, obviously

$$1 - \left(1 - \frac{1}{k}\right)^k = 1 > 1 - 1/e$$

Hence, we prove that the greedy algorithm achieves an approximation factor of $(1 - \left(1 - \frac{1}{k}\right)^k) > (1 - \frac{1}{e})$.