Problem 2

Basic Idea

- *PERFECT ASSEMBLY* $\in \mathcal{NP}$: a permutation P of $s_i \in S$ is a certificate that can be checked in polynomial time by ensuring that $\bigcup P = S$, and $|P| = |S|$, and for every consequtive pair p_i, p_{i+1} in P, $p_i[l+1, 2l] \circ p_{i+1}[1, l] \in T$[1].
- *PERFECT ASSEMBLY* is $\mathcal{NP}$-hard: *HAMILTONIAN PATH* $\leq_P$ *PERFECT ASSEMBLY*
  - For every $v \in V$, generate a string vv in S.
  - For every $(u, v) \in E$, generate a string uv in T.
  - $G = (V, E)$ has a hamiltonian path iff S has a perfect assembly with respect to T.

Proof: *PERFECT ASSEMBLY* $\in \mathcal{NP}$

- A certificate *str* is a string permutation of all $s_i \in S$, such that every consequtive pair (s_i, s_{i+1}) is corroborated by a string in T.
- First, we check if *str* is in fact a permutation of S by checking if all consequtive substrings of lengths $2l$ are elements of S, if there are duplicates among these substrings, and ensure that the number of the substrings equals $|S|$. This can be done in polynomial time by comparing every such substring in *str* with each other (find duplicates) and finding every such string in S.
- Then, we check if every consequtive pair of substrings of length $2l$ in *str* is corraborated by a string $t_k \in T$. This is the case if, for two substrings s_i, s_{i+1}, $s_i[l+1, 2l] \circ s_{i+1}[1, l] \in T$.

Proof: *PERFECT ASSEMBLY* is $\mathcal{NP}$-hard

We show that *HAMILTONIAN PATH* $\leq_P$ *PERFECT ASSEMBLY*. We know that *HAMILTONIAN PATH* is $\mathcal{NP}$-hard. We show how to generate an instance of *PERFECT ASSEMBLY* and show that it has a solution if and only if there exists a hamiltonian path.

Generate instance of *PERFECT ASSEMBLY*

Given an instance of *HAMILTONIAN PATH* with $G = (V, E)$, we generate an instance of *PERFECT ASSEMBLY* as follows.

- $\Sigma = V$, where Σ is the alphabet.
- $S = \{vv \mid v \in V\}$
- $T = \{uv \mid (u, v) \in E\}$
- $l = 1$

[1] $\circ$ is string concatenation and $str[a, b]$ denotes the substring from index a to b (inclusive) of str.

HAMILTONIAN PATH $\Rightarrow$ *PERFECT ASSEMBLY*

- Let $G = (V, E)$ be a graph that contains a hamiltonian path with $|V| = n$.
- Then there is a sequence $(i_1, i_2, \ldots i_n)$, such that $(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ is a hamiltonian path.
- Then $str = v_{i_1}v_{i_1}v_{i_2}v_{i_2} \ldots v_{i_n}v_{i_n}$ is a certificate that proves that $S = \{v_{i_1}v_{i_1}, v_{i_2}v_{i_2}, \ldots, v_{i_n}v_{i_n}\}$ is a perfect assembly with respect to T, because *str* is a permutation of S (therefore, we visit every vertex exactly once) and, for every consequtive pair $v_iv_iv_{i+1}v_{i+1}$, there must be an edge $(v_i, v_{i+1}) \in E$ (otherwise, this would not be a path in G at all).

PERFECT ASSEMBLY $\Rightarrow$ *HAMILTONIAN PATH*

- Let $G = (V, E)$ be a graph with $|V| = n$ and let S be a perfect assembly with respect to T.
- Then, there must be a certificate *str* such that *str* is a permutation of S. Therefore, for $str = v_{i_1}v_{i_1}v_{i_2}v_{i_2} \ldots v_{i_n}v_{i_n}$, $P = (v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ is a permutation of V.
- P is a hamiltonian path because it visits every vertex exactly once and, for every consequtive pair of vertices (v_k, v_{k+1}), there exists an edge $(v_k, v_{k+1}) \in E$, because every consequtive pair in *str* is corroborated by a $t_j \in T$, and all $t_j \in T$ represent edges in G by definition.

Complexity of Reduction

$|\Sigma| = |S| = |V|$, $|T| = |E| = \mathcal{O}(|V|^2)$, $l = 1$, and $|s_i| = 2$ for every $s_i \in S$. Therefore, all quantities of the generated instance are polynomial in the number of vertices $|V|$ and the instance can be generated in polynomial time.

Problem 5

Subproblem a: Solve in $\mathcal{O}(2^n \cdot p(n))$ time

Basic Idea

- Solve with dynamic programming: Let $HAM[S, s, t]$ be true iff there is a hamiltonian s-t path in $S \subseteq V$.
- $HAM[S, s, t] = \bigvee_{k \in S} HAM[S - \{t\}, s, k] \wedge (k, t) \in E$
- Intuition: There is a hamiltonian path from s to t in S, if there is a hamiltonian path from s to some $k \in S$ in $S - \{t\}$ and $(k, t) \in E$.

Dynamic Programming Table

- Base case for subsets of size one: $\forall a_1 \in V, a_2 \in V, a_3 \in V : HAM[\{a_1\}, a_2, a_3] := a_1 = a_2 = a_3$
- $\forall s \in V, t \in V, S \subseteq V, |S| > 1 : HAM[S, s, t] = \bigvee_{k \in S} HAM[S - \{t\}, s, k] \wedge (k, t) \in E$
- We fill the table with increasing cardinalities of S, i.e. we first fill the table for all subsets S with $|S| = 2$, then for all subsets S with $|S| = 3$, and so on. Notice, that, in the recursive definition, we only use table entries with a subset S that has a smaller cardinality.

Full Algorithm

- Fill the dynamic programming table for all base cases.
- Fill the dynamic programming table according to the recursive definition as described in the previous section.
- Check all values $HAM[V, a, b]$ for all $a \in V, b \in V$. If at least one of them is true, then there is a hamiltonian path in the graph.

Space Complexity

The size of the table is $2^{|V|} \cdot |V| \cdot |V|$, so the space complexity is exponential in the number of vertices.

Runtime Complexity

- Filling the dynamic programming base cases: $|V|^3$ entries.
- Filling the rest of the table according to the recursive definition: there are $\mathcal{O}(2^{|V|} \cdot |V|^2)$ table entries and for every entry we iterate over all $k \in S$, where $|S| \leq |V|$. Therefore, this takes $\mathcal{O}(2^{|V|} \cdot |V|^3) = \mathcal{O}(2^{|V|} \cdot p(|V|))$ time.
- We scan the table for all $a \in V, b \in B$ at $HAM[S, a, b]$. This takes $\mathcal{O}(|V|^2)$ time.
- The overall runtime complexity is $\mathcal{O}(2^{|V|} \cdot |V|^3)$.

Proof

We prove by induction over the size of $|S|$ that algorithm is correct.

- *Induction Base:* Let $S \subseteq V$ with $|S| = 1$. Then, there can be hamiltonian s-t path if and only if $s = t$ and $S = \{s\}$, because we only take a look at the subgraph with this single vertex. Therefore, $HAM[\{a\}, s, t] := a = s = t$.
- *Induction Hypothesis:* Let $HAM[S, s, t] = \textit{true}$ if and only if there is a hamiltonian s-t path in $S \subseteq V$, for all $|S| \leq k$.
- *Induction Proof:* We want to decide whether there is a hamiltonian s-t path in $S \subseteq V$ with $|S| = k+1$. A hamiltonian path is a sequence of vetices $(v_1, v_2, \ldots, v_{k+1})$. Such a hamiltonian path exists if and only if a hamiltonian path $(v_1, v_2, \ldots, v_k)$ exists in $S - \{v_{k+1}\}$ and there is an edge $(v_k, v_{k+1}) \in E$ in G. Therefore, $HAM[S, s, t] = \bigvee_{k \in S} HAM[S - \{t\}, s, k] \wedge (k, t) \in E$ for $|S| > 1$. From the induction hypothesis, we know the correct value of $HAM[T, a, b]$ for $a \in V, b \in V, T \subseteq V, |T| = k$.
- A hamiltonian path can start and end at any vertex. Therefore, there is a hamiltonian path in G, iff $HAM[V, a, b] = \textit{true}$ for at least one combination of $a \in V, b \in V$.

Subproblem b: Solve in polynomial space

Basic Idea

- The basic idea relies on the concept of alternate addition/substraction from the lecture.
- We count the number of all paths (starting at v_1 and ending at v_n) of size $|V|$ (that use $|V|-1$ edges) in $G = (V, E)$.
- A path of length $|V|$ is not hamiltonian iff at least one vertex v was not visited. Therefore, for every $v \in V$, we substract the number of paths of length $|V|$ in $G' = (V - \{v\}, E')$, i.e. all paths that do not visit v.
- Notice, that we substracted paths that miss two vertices twice. We can make up for this by adding all pathes of size $|V|$ that miss two vertices. But now we added paths that miss three vertices too often. We keep adding and substracting in such a way.
- Counting all these pathes takes exponential time because we eventually have to examine all subsets $S \subseteq V$ for paths of size $|V|$. Notice, that we do not need a DP table here, resulting in polynomial space complexity.
- We repeat this process for all possible start and end vertices.

Full Algorithm

We run the following algorithm for every pair $s \in V, t \in V$ and return *true* if at least one run returns *true*.

$counter \leftarrow 0$
$sign \leftarrow 1$
for $k \leftarrow |V|$ **downto** 1 **do**
  for all $S \subseteq V \wedge |S| = k$ **do**
    $counter \leftarrow counter + sign \cdot countPaths((S, E'), s, t)$
  end for
  $sign \leftarrow -sign$
end for
  return *true* **iff** $counter > 0$

The function $countPaths((V, E), s, t)$ counts the number of all paths of length $|V|$ in (V, E) that start at s and end at t. This can be done with a modified version of BFS. Note, that it is allowed to visit vertices and edges multiple times.

Proof

- $countPaths((V, E), s, t)$ counts the number of s-t paths of size $|V| - 1$ in $G = (V, E)$.
- Let missing[i] be the number of s-t paths in all graphs with i vertices missing from V. E.g. $missing[1]$ is the number of s-t paths in all graphs $G = (V', E')$ with $V' = V - \{v\}$ for every $v \in V$.
- A graph $G = (V, E)$ contains a hamiltonian path if there is at least one s-t path for some $s \in V, t \in V$ where no vertex is missing in the path. We can determine the number of these paths by substracting the number of paths in all graphs with one vertex missing from the number of paths in the graph with no vertex missing. The number of these paths is $missing[0] - missing[1] + missing[2] - missing[3] + \ldots$. We have to use this alternating sum because we counted the number of paths with two vertices

missing in the graphs multiple times in *missing*[2]. Then, again, we counted some paths multiple times, yielding this alternating sum.

Finding all s-t paths of length $|V|-1$

- We maintain an array counter$[v_i, d]$ that counts how often we encounter the vertex v_i with a current distance of d.
- At the beginning, counter$[s, 0] = 1$ and all other array slots are initialized to 0.
- We run BFS and maintain tuples of *vertex* v and *current distance* d in the queue.
- For every tuple (v, d), we traverse all outgoing edges $(v, u) \in E$ and increase the counter for u at distance $d+1$ by the counter for the current tuple counter$[v, d]$.
- Note, that BFS traverses tuples (v, d) with increasing values of d, i.e. before a tuple (v, d') with $d' > d$ is processed, all tuples (u, d) have been processed. Therefore, we will never increase the counter for an already processed tuple.
- *counter*$[t, |V|-1]$ contains the number of s-t paths of length $|V|-1$.

```
counter[v_i, d] ← 0 ∀v_i ∈ V, 0 ≤ d ≤ |V|
counter[s, 0] ← 1
Q ← new Queue
Q.add((s, 0))
while |Q| > 0 do
    p = (v, d) ← Q.pop()
    for all (e = (v, u) ∈ E do
        if d + 1 < |V| ∧ (u, d + 1) ∉ Q then
            Q.push((u, d + 1))
        end if
        counter[u, d + 1] ← counter[u, d + 1] + counter[v, d]
    end for
end while
    return counter[t, |V| − 1]
```

This algorithm fills $|V| \cdot (|V|-1) = \mathcal{O}(|V|^2)$ array slots, and for every array slot (v, d), the algorithm traverses all of v's incident nodes u with $(v, u) \in E$. Therefore, the runtime complexity for this algorithm is $\mathcal{O}(|V|^2 \cdot |E|) = \mathcal{O}(|V|^4)$. The space complexity is $\mathcal{O}(|V|^2)$, i.e. the size of the array.

Runtime Complexity

- Generating all subsets $S \subseteq V$ in the for loop: $\mathcal{O}(2^{|V|})$ total iterations.
- One run of *countPaths* per subset: $\mathcal{O}(2^{|V|} \cdot |V|^4)$.
- Running the whole algorithm for all pairs of $s \in V, t \in V$: $\mathcal{O}(2^{|V|} \cdot |V|^6)$.

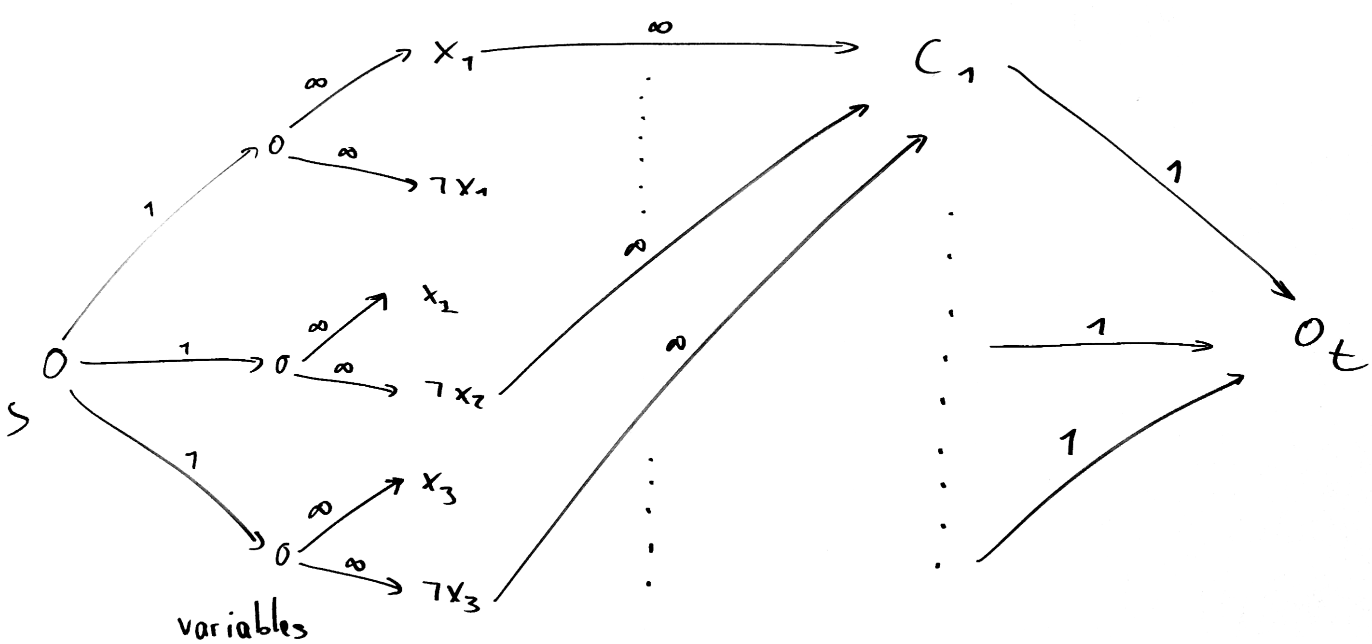
Space Complexity

- The space for the current subset S and the *counter* variable and the *sign* variable is constant.
- Every run of *countPaths* requires $\mathcal{O}(|V|^2)$ temporary space, i.e. this space is only required during one run of the function.
- The overall space complexity of the algorithm is $\mathcal{O}(|V|^2)$.

Problem 3

Basic Idea

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$



- The number of clauses must be equal to number of variables, because every variable must appear in exactly 3 clauses.
- We build a flow network as illustrated above. Every variable can send a flow of 1 to its positive or negative literal and every literal can satisfy clauses it appears in. We only allow it to satisfy one clause. In fact, a literal could sometimes satisfy more than one clause, but we can always find an assignment such that every variable satisfies exactly one clause[2]. The capacity constraints on the edges leading to the sink t ensure that every clauses uses only one literal for satisfiability.
- There is a valid assignment of variables iff the max flow value is n. A variable is set to *true* if its positive literal receives a flow of 1. Otherwise, it is *false*.

[2] According to Hall's theorem. See proof for details.

Flow Network Construction

- Add a source s and a sink t.
- For every variable x_i, add a node v_{x_i} and two literal nodes l_{x_i} and $l_{\neg x_i}$.
- Add edges (s, v_{x_i}) with a capacity of 1, and edges (v_{x_i}, l_{x_i}) and $(v_{x_i}, l_{\neg x_i})$ with capcities of ∞.
- For every clause C_i, add a node v_{C_i}.
- Add edges (v_{C_i}, t) with a capacity of 1.
- If a is a literal and $a \in C_i$, add an edge (l_a, v_{C_i}) with a capacity of ∞.

Algorithm

- Build the flow network for the problem instance.
- Run the Ford-Fulkerson algorithm to obtain a max flow.
- If a literal node l_{x_i} receives a flow of 1, set x_i to *true*, otherwise set it to *false*.

Proof

- *Integrality constraint:* All flow capacities are integers. Therefore, the Ford-Fulkerson algorithm generates an integer flow.
- *Valid assignment:* All variable nodes can send a flow of either 0 or 1 to one of the literals (due to integrality). In case the max flow value is $|\mathit{Variables}|$, i.e. there is a perfect matching[3], every variable sends a flow of 1 to either its positive or negative literal. Therefore, the assignment is valid in a sense that every variable has an assignment and there is no contradiction.
- *Perfect matching*
  - According to Hall's theorem, a bipartite graph $G = ((\mathit{Variables}, \mathit{Clauses}), E)$ has a matching of size *Variables*, if and only if, for every $S \subseteq \mathit{Variables}$, $|N(S)| \geq |S|$, where $N(S)$ is the set of vertices in *Clauses* that are reachable from S.
  - In the graph, we have additional nodes for positive and negative literals for every variable. When we think about a matching, we consider the variable nodes to be directly connected to the clause nodes. This is does not change anything in the following argument.
  - Let $S \subseteq \mathit{Variables}$. $N(S) \geq \max\{3, |S|\}$, because every variable appears in exacly 3 clauses. Therefore, 3 is a lower bound for $S \neq \emptyset$. For $|S|$ variables, we have $3|S|$ literals appearing in some clauses. Since every clause has exaclty 3 literals, we need at least $\frac{3|S|}{3} = |S|$ clauses. Therefore, S is connected to at least $|S|$ clauses.
  - Therefore, there is always a matching of size *Variables*, i.e. there is always a perfect matching. Therefore, this version of *3SAT* is always satisfiable.

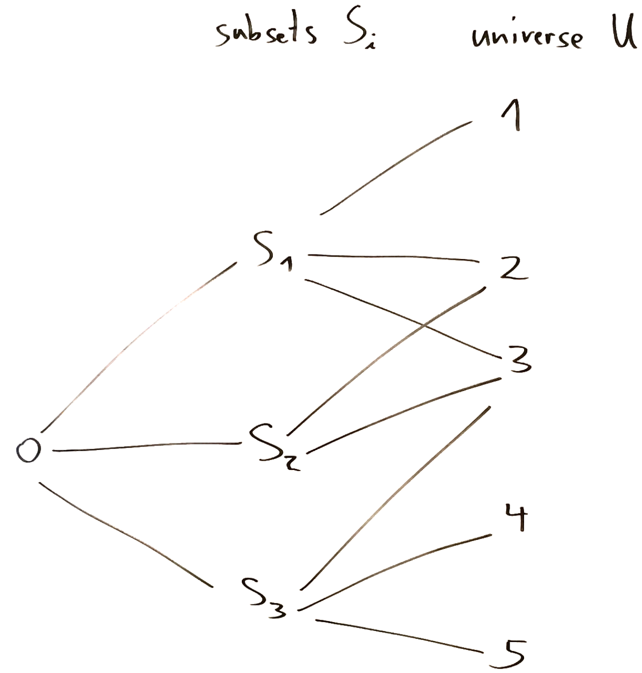
[3] We show below that this is always the case.

Runtime Complexity

- Building the flow network: We create $\mathcal{O}(n)$ nodes for variables/literals and $\mathcal{O}(n)$ nodes for clauses, since the number of clauses equals the number of variables. We add no more than $\mathcal{O}(n^2)$ edges.
- The max flow can be calculated with the Edmonds-Karp algorithm in $\mathcal{O}(|V||E|^2) = \mathcal{O}(n^5)$.
- The overall runtime is $\mathcal{O}(n^5)$. Therefore, the runtime complexity is polynomial in the number of variables.

Problem 1

Basic Idea



$k_{STEINER} = |U| + k_{SET\text{-}COVER}$

$S_1 = \{1, 2, 3\}$

$S_2 = \{2, 3\}$

$S_3 = \{3, 4, 5\}$

$S = \{S_1, S_2, S_3\}$

$U = \{1, 2, 3, 4, 5\}$

$k_{SET\text{-}COVER} = 2$

- *STEINER* $\in \mathcal{NP}$: a subset $F \subseteq E$ of edges is a certificate that can be checked in polynomial time by ensuring that $|F| \leq k$ and that, for an arbitrary edge $(u, v) \in F$, all nodes in V can be reached from u. This can be done with DFS.

- *STEINER* is $\mathcal{NP}$-hard: $SET\,COVER \leq_P STEINER$
  - For every element $u \in U$ and every subset $S_i \in S$, generate a vertex.
  - Connect all subset vertices to a common root vertex and all elements to the subsets they are included in.
  - $k_{STEINER} = k_{SET\,COVER} + |U|$
  - There is a set cover of at most $k_{SET\,COVER}$ subsets iff there is a Graphical Steiner Tree with at most $k_{STEINER}$ edges.

Proof: $STEINER \in \mathcal{NP}$

- A certificate is a subset $F \subseteq E$.
- First, we check if $|F| \leq k$.
- Then, we check if the graph induced by F is contains all vertices and is connected. We take an arbitrary edge $(u, v) \in F$ and start a DFS on u. There is a Graphical Steiner Tree of at most size k iff the DFS visits all $v \in V$. The DFS can maintain an array of already visited nodes and determine whether all vertices were visited. DFS takes $\mathcal{O}(|V| + |F|) = \mathcal{O}(|V| + |E|)$ time.

Proof: *STEINER* is $\mathcal{NP}$-hard

We know that *SET COVER* is $\mathcal{NP}$-hard and show how to reduce it *STEINER*.

Graph Construction

We show how to generate a graph for the Graphical Steiner Tree problem from an instance of *SET COVER*. Let $I = (U, S, k)$ be an instance of *SET COVER*.

- Add a root vertex v_r.
- For every element $u_i \in U$, add a vertex v_{u_i}.
- For every subset $S_i \in S$, add a vertex v_{S_i}.
- If $u_i \in S_i$, add an edge $\{v_{u_i}, v_{S_i}\}$.
- For every subset S_i, add an edge $\{v_r, v_{S_i}\}$.

Generate instance of *STEINER*

- Build the graph according to the previous section.
- Set $k_{STEINER} = k_{SET\,COVER} + |U|$.

SET COVER $\Rightarrow$ *STEINER*

- Let $I = (U, S, k)$ be an instance of *SET COVER* that has a set cover $C = \{S_{a_1}, S_{a_2}, \ldots, S_{a_k}\}$ of size k.
- Then, the according Graphical Steiner Tree graph has the following Steiner Tree with $k + |U|$ edges: select all edges $\{v_r, S_{a_i}\}$ for all $1 \leq i \leq k$ (k edges). In adddition, select $|U|$ edges $\{S_{a_i}, u\}$ such that all $u \in U$ are covered. There must always be such edges, because C is a set cover, i.e. all $u \in U$ are contained in some sets $S_{a_i} \in C$.

STEINER $\Rightarrow$ *SET COVER*

- Let $I = (U, S, k)$ be an instance of *SET COVER* and $F \subseteq E$ be a set of edges in the Graphical Steiner Tree graph with $|F| \leq k + |U|$.
- Then, all vertices v_{u_i} are connected to v_r in F. Since there are no connections among the vertices v_{u_i}, they must all be connected through some vertices v_{S_i}. There cannot be more than k edges $\{v_r, v_{S_i}\}$ in use, because there are $|U|$ edges needed for all v_{u_i} and we have at most $k + |U|$ edegs in F. $\{S_i \mid \{v_r, v_{S_i}\} \in F\}$ is a set cover of size k, because, according to the graph construction, there is an edge between a vertex v_{u_i} and a subset vertex v_{S_i} if and only if $u_i \in S_i$. Therefore, $\bigcup_{S_i \in S : \{v_r, v_{S_i}\} \in F} S_i = U$.

Complexity of Reduction

We generate $\mathcal{O}(|U| + |S|)$ nodes and no more than $\mathcal{O}((|U| + |S|)^2)$ edges. Therefore, the reduction is polynomial in the size of the *SET COVER* instance.

Problem 4

Basic Idea

- Dynamic programming: $CUT[T, d]$ is the maximum achievable cut value of subtree T, where the difference between the partition set sizes is exactly d.
- $P = \{T, T_1, T_2\}$, where T_1 is T's left child and T_2 is T's right child[4].
- $CUT[T, d] = \max_{S \subseteq P} \{\sum_{(a,b) \in S \times P - S} w_{a,b} + \max_{-n \leq a_1, a_2 \leq n \wedge a_1 + a_2 = d - ||P - S| - |S||} \{CUT[T_1, a_1] + CUT[T_2, a_2]\}\}$
- For all leaves $L_i \in V$: $CUT[L_i, 1] = 0$ and for $d \neq 1 : CUT[L_i, d] = -\infty$.
- For every subtree T, it does not matter in which partition set T is. By swapping all nodes in set A and set B, CUT stays the same. Therefore, $\forall T \in V, d \in \mathbb{Z} : CUT[T, d] = CUT[T, -d]$.
- For every tree, we decide which nodes P are in the same partition set. This decision contributes to the partition size difference with a value of $||P| - |S||$, i.e. the difference of the number of nodes of P in A and in B. We must accumulate a difference in partition sizes in the child nodes T_1 and T_2 of $d - ||P| - |S||$, i.e. the difference in partition sizes in T_1 and T_2 can be arbitrary as long as the total difference in partition size is exactly d.

[4] As stated in the problem description, we only have binary trees.

Full Algorithm

- The following code describes the function *calculateCut*(T) that fills the dynamic programming table for all nodes in subtree T. This function fills the table in a DFS style. It calculates the values for both children first and then calculates the value according to the formula.

  if $T_1 = \emptyset \wedge T_2 = \emptyset$ **then**
      $CUT[T, 1] \leftarrow 0$
      $CUT[T, d] \leftarrow -\infty \forall d \in \mathbb{Z}\{1\} : -|V| \leq d \leq |V|$
  else
      if $T_1 \neq \emptyset$ **then**
          $calculateCut(T_1)$
      else
          $T_1 \leftarrow T_\emptyset$
      end if
      if $T_2 \neq \emptyset$ **then**
          $calculateCut(T_2)$
      else
          $T_2 \leftarrow T_\emptyset$
      end if
      for all $d \in \mathbb{Z} : -|V| \leq d \leq |V|$ **do**
          $CUT[T, d] \leftarrow -\infty$
          for all $S \subseteq \{T, T_1, T_2\}$ **do**
              $r \leftarrow \sum_{(a,b) \in S \times (\{T, T_1, T_2\} - S)} w_{a,b}$
              for all $a_1, a_2 \in \mathbb{Z} : -|V| \leq a_1, a_2 \leq |V| \wedge d = a_1 + a_2 + ||\{T, T_1, T_2\} - S| - |S||$ **do**
                  $v \leftarrow CUT[T_1, a_1] + CUT[T_2, a_2] + r$
                  if $v > CUT[T, d]$ **then**
                      $CUT[T, d] \leftarrow v$
                      $SOL[T, d] \leftarrow (S, (T_1, a_1), (T_2, a_2))$
                  end if
              end for
          end for
      end for
  end if

- Then, we invoke the algorithm as follows (R is the root of the tree).

  $CUT[T_\emptyset, 0] \leftarrow 0$
  $CUT[T_\emptyset, d] \leftarrow -\infty \ \forall d \in \mathbb{Z} : -|V| \leq d \leq |V|$
  $calculateCut(R)$
  $output(R \in A)$
  $printSol(R, 0, A)$

- This function *printSol*$(T, d, side)$ prints the rest of the solution. It retrieves the path of decisions that was made during the actual algorithm from the SOL table.

  $s \leftarrow SOL[T, d]$
  $S \leftarrow s[1]$
  $T_1 \leftarrow s[2][1]$
  $a_1 \leftarrow s[2][2]$
  $T_2 \leftarrow s[3][1]$

$a_2 \leftarrow s[3][2]$
for all $(T_c, a_c) \in \{(T_1, a_1), (T_2, a_2)\}$ **do**
  if $T_c \neq T_\emptyset$ **then**
    if $T_s \in S = T \in S$ **then**
      $output(T_c \in side)$
      $printTable(T_c, a_c, side)$
    else
      $otherSide \leftarrow A$ **if** $side = B$ **else** B
      $output(T_c \in otherSide)$
      $printTable(T_c, a_c, otherSide)$
    end if
  end if
end for

Proof

- *Independent subproblems:* For two trees T_1 and T_2, where T_1 and T_2 are not subtrees of each other, an optimal solution for T_1 is independent from an optimal solution for T_2, since there are no edges between T_1 and T_2.

- *Optimal substructure:* Given an optimal solution for all child nodes, the algorithm finds an optimal solution for the whole subtree. We prove this by induction.

  - *Induction Base:* If T is a leaf, i.e. it has no child nodes, then there are no edges involved at all. Therefore, the cut value is always 0 and the difference between the sizes of the partition sets must be 1, i.e. $d = 1$, because we only have one vertex that must be in one set. For other values of d, the problem is infeasible. We denote this by a cut value of $-\infty$, i.e. the algorithm will never choose it unless the whole problem instance is infeasible.
  - *Induction Hypothesis:* Assume that for a vertex T, we know the optimal solution for all $-|V| \leq d \leq |V|$ for both the left and the right child vertex T_1 and T_2.
  - *Induction Proof:* We find the optimal solution for a given value of d by evaluting all possible solutions: we have to decide in which set we put T, T_1 and T_2[5] and think about the subtrees. In fact, it is not important, in what exact set we put the vertices, since we are only interested in the difference of size of the partition sets. To achieve a value of d, the difference in the left subtree plus the difference (of partition set sizes) in the right subtree plus the difference of the vertices T, T_1 and T_2 must equal d[6]. By evaluating all these possibilities, we can be sure to find the optimal solution. We only have to consider values $-|V| \leq d \leq |V|$, the difference of partition size sets cannot be greater than the number of vertices in total[7].

Runtime Complexity

- We fill the DP table bottom up using DFS. Therefore, we visit each vertex and edge once: $\mathcal{O}(|V|+|E|)$.

[5] In the algorithm, we consider all possibilities to put these vertices in the set S or not. Actually, this is not necessary, since the symmetric cases, where we switch the partition sets, have the same cut value and partition set size difference. But it doesn't hurt, either.

[6] Note, that it does not matter whether T_1 and/or T_2 put more vertices in set A or in set B, since both cases are completely symmetric. We can always swap both sets.

[7] In fact, for most cases we will have a value of $-\infty$ in the DP table for the lower vertices.

- For every vertex, we examine all values of $-|V| \leq d \leq |V|$: $\mathcal{O}(|V|)$. For every value of d, we examine all possibilities of taking subsets S of $\{T, T_1, T_2\}$: constant. For every subset S, we consider all possibilites to achieve a value of $d - ||P - S| - |S||$: $\mathcal{O}(|V|^2)$ possibilities of assigning a_1 and a_2. In total, we consider $\mathcal{O}(|V|^3)$ cases per vertex.

- For every case, we check two values in the DP table (constant time). The runtime complexity for filling the whole table is $\mathcal{O}(|V|^4)$.

- Printing the solution is done by visiting every vertex once with a DFS and examining one DP table entry per vertex. The overall runtime complexity for the algorithm is $\mathcal{O}(|E| + |V|^4) = \mathcal{O}(|V|^4)$.

Problem 6

Subproblem a: Approximation Algorithm

Basic Idea

- Sort all visitors V_i by their value v_i in a decreasing way.

- Greedy algorithm: Allocate the next visitor V_i to the the ad A_j that has currently the lowest sum.

Full Algorithm

sort V by value
for all $V_i \in V$ **do**
  $minAd \leftarrow \min_{1 \leq i \leq m} A_i$
  add V_i to $minAd$
end for

Proof of Approximation Factor

- We assume that $n \geq m$. Otherwise, at least one ad does not get any visitor. In that case, the maximum spread is zero for both the optimal algorithm and the greedy algorithm.

- Let T_{OPT} be the maximum spread that is generated by the optimal algorithm. $T_{OPT} \leq \frac{\sum_{i=1}^{n} v_i}{m}$, because in case all visitors are equally distributed, the minimum value sum is maximum.

- Let T_G be the spread that is generated by the greedy algorithm. Let A_l be the ad with the lowest sum. We denote the sum of the ad A_i with v_{A_i}.

  - A_l has the lowest sum: $\forall 1 \leq i \leq m : v_{A_l} \leq v_{A_i}$.
  - Let L_k be the last visitor allocated to A_k. $\forall 1 \leq i \leq m : v_{A_l} \geq v_{A_i} - v_{L_i}$. Otherwise, we would have allocated L_i to A_l.
  - In the formula above $v_{A_i} \geq \frac{\sum_{i=1}^{n} v_i}{m}$, because if this ad was below average, we would have selected A_l instead. $v_{L_i} \leq \frac{\sum_{k=1}^{n} v_k}{2m}$ according to the problem description.
  - Therefore, $T_G \geq \frac{\sum_{i=1}^{n} v_i}{m} - \frac{\sum_{i=1}^{n} v_i}{2m} = \frac{\sum_{i=1}^{n} v_i}{2m}$.

- Therefore, $T_G \geq \frac{T_{OPT}}{2}$.

Subproblem b

- $m = 2$
- Visitor values: {10, 8, 5, 4, 3}
- Optimal solution: 15 ($A_1 = \{10, 5\}$, $A_2 = \{4, 8, 3\}$)
- Greedy algorithm output: 14 ($A_1 = \{10, 4\}$, $A_2 = \{8, 5, 4\}$)

Problem 7

Subproblem a

- Let T be an arbitrary independent set in G and S be an independent set generated by the greedy algorithm in G.
- Let $v \in T$ be an arbitrary node in T's independent set.
  - First case: $v \in S$: v is also part of S's independent set. Nothing more to show.
  - Second case: $v \notin S$: v must have been eliminated during the run of the greedy algorithm. Otherwise, we would have chosen it eventually. A node is elimated if and only if it is chosen to be part of S (this cannot happen since $v \notin S$) or a neighbor u was chosen. In that case, all of u's neighbors including v were eliminated. Therefore, a neighbor u was chosen before v. Therefore, $w(u) \geq w(v)$, since the greedy algorithm selects vertices with decreasing weights, and $(v, u) \in E$, since u and v are neighbors. ■

Subproblem b

- Let V be the set of vertices and $(v_1, v_2, \ldots, v_n)$ be the sorted sequence of vertices, i.e. $w(v_i) > w(v_j)$ if $i < j$.
- Let T_{OPT} be the value of the optimal solution. $T_{OPT} \leq \sum_{i=1}^{\frac{n}{2}} v_i$, because the optimal independent set cannot contain more than $\frac{n}{2}$ vertices. In the best case, we select every other vertex in every row, starting with a selected vertex in even/odd rows and with an unselcted vertex in an odd/even row. For example, for select the first, third, fifth, ... vertex in the first row, select the second, fourth, sixth, ... vertex in the second row, and so on. In the best case, we also select the first $\frac{n}{2}$ biggest vertices.
- Let T_G be the value of the solution of the greedy algorithm. $T_G \geq \sum_{i=1}^{\frac{n}{5}} v_{5i-4}$, because, in the worst case, every vertex that we choose eliminates four other vertices. Therefore, we only select one out of five vertices. In the worst case, we also eliminate the next four best vertices by selecting the currently best free vertex. I.e. we select the first best vertex, then the sixth best vertex (because we eliminated the second to fifth best vertex), then the eleventh best vertex, and so on.
- Need to show: $T_G \geq \frac{1}{4} T_{OPT}$, i.e. $\frac{T_G}{T_{OPT}} \geq \frac{1}{4}$

Problem 8

Basic Idea

- Replace all weights by w_i by rounded values $w'_i = \lfloor \frac{w_i}{w} \rfloor$ with $w = \frac{\epsilon W}{n}$, where W is the capacity of the bag and n is the number of elements.
- Solve *KNAPSACK* with dynamic programming for the rounded values.
  - $KNAPSACK[i, j]$ is the maximum achievable value for elements with index e with $1 \leq e \leq i$ and a bag capacity of j.
  - $KNAPSACK[i, j] = \max\{KNAPSACK[i-1, j-w'_i]+v_i, KNAPSACK[i-1, j]\}$. It is not allowed to exceed the weight constraint, see next section for detailed formula.
- For the rounded version of the problem, we are interested in a bag of capacity $W' = \lceil \frac{W}{w} \rceil = \lceil \frac{n}{\epsilon} \rceil$.
- For a fixed $\epsilon > 0$, the size of the DP table and therefore the runtime complexity is then independent of W and polynomial in the problem instance size. The DP table has a size of $\mathcal{O}(n \cdot \frac{n}{\epsilon})$.

Recursive DP Definition

- Let W' be the rounded capacity of the bag, v_i be the value of item i, and w'_i be the rounded weight of item i.
- $KNAPSACK[0, j] = 0$
- $KNAPSACK[i, j] = \max\{KNAPSACK[i-1, j-w'_i] + v_i \textbf{ if } j - w'_i \geq 0, KNAPSACK[i-1, j]\}$

Full Algorithm

- Let $w = \frac{\epsilon W}{n}$.
- Let $W' = \lceil \frac{W}{w} \rceil$.
- Let $w'_i = \lfloor \frac{w_i}{w} \rfloor$.
- Fill the DP table according to the recursive definition, starting with $i = 1$ and increasing values of i. For every pair (i, j), store the chosen path, i.e. whether to include item i or not, in a separate array *SOL* to be able to recover the solution later.
- $KNAPSACK[n][W']$ contains the maximum achievable value for a bag of size W' and all items. We trace back the solution through *SOL* and output all items that were taken.

Proof (attempt, not completed[8])

- *Capacity constraint:* We calculated *KNAPSACK* for rounded values and $W' = \lceil \frac{n}{\epsilon} \rceil$. In the original problem, this corresponds to a bag of size bigger than W, since we rounded W' up and w'_i down. Need to show that this rounding leads to a bag of size no larger than $(1 + \epsilon)W$, in comparison to the original bag size with the original item weights.

[8] I am not sure whether this approach turns out to be correct. Am I at least on the right way?

- *Value constraint:* Since we rounded weights down and the bag size up, it might now be possible to achieve a better value for the bag. We can be sure, that the solution will be at least as good as optimal solution for the original $KNAPSACK$, in case we take all items that we would have taken for the original $KANPSACK$. In that case, the value corresponds to the optimal value in the original problem, and the capacity constraint of the bag is satisfied, since we only made items lighter (due to rounding down) and the bag capacity bigger (due to rounding up). Therefore, $V_A \geq V_{OPT}$, where V_A is the value of the bag calculated by this algorithm and V_{OPT} is the value of the optimal solution in the original $KNAPSACK$ problem.

- This algorithm works for all given values of V and W: let W be the size of the bag. V cannot be better than the optimal solution for that bag size, i.e. $V \leq V_{OPT} \leq V_A$.

Runtime Complexity

The weights can be adapted in $\mathcal{O}(n)$, where n is the number of items. The DP table size is $\mathcal{O}(\frac{n^2}{\epsilon})$, and for every table entry we inspect two other table entries. Therefore, the runtime for caluclating the table is $\mathcal{O}(\frac{n^2}{\epsilon})$. The solution can be extracted by inspecting n entries in the SOL table. Therefore, the overall runtime complexity is $\mathcal{O}(\frac{n^2}{\epsilon})$.