Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Ki Min Kang Wisc id: 908 404 5062

Intractibility

1. *Kleinberg, Jon. Algorithm Design (p. 506, q. 4).* A system has a set of n processes and a set of m resources. Each process specifies a set of resources that it requests to use. Each resource might be requested by many processes; but it can only be used by a single process. If a process is allocated all the resources it requests, then it is active; otherwise it is blocked.

   Thus we phrase the Resource Reservation Problem as follows: Given a set of processes and resources, the set of requested resources for each process, and a number k, is it possible to allocate resources to processes so that at least k processes will be active?

   For the following problems, either give a polynomial-time algorithm or prove the problem is NP-complete.

   (a) The general Resource Reservation Problem defined above.

1) Checking if it's a doable problem (in NP)
We can quickly check a potential solution to see if it works.
For each resource, count how many of the K processes need it.
If the count is more than 1 for any resource, the solution won't work because each resource can only be used by one process at a time.

2) Proving it's a tough problem (NP-Hard)
- We compare it to a known tough problem, like the Independent Set problem, which is about finding a set of nodes in a graph that are not connected by edges.
- We transform the Independent Set problem into our Resource Reservation problem by making each process a node and each resource an edge.
- If we find an independent set of size K, we can also allocate resources for K active processes.

Since we can compare it to a known hard problem and since we can check a solution quickly, the Resource Reservation Problem is both NP and NP-Hard.

Therefore, it's very unlikely that there's a simple method to always find a solution quickly

(b) The special case of the problem when $k = 2$.

Let's look at a special case where we just want to make sure that two tasks can be completed

1. Look at every pair of tasks to see if there are any tools they both need.
2. If we find even one pair of tasks that don't need any of the same tools, it is in very luck! We can set those two tasks up and have them both going at the same time.
3. But if every pair of tasks needs at least one tool in common, it's a no-go. We can't have two tasks running simultaneously because they'd be fighting at least one tool.

(c) The special case of the problem when there are two types of resources–say, people and equipment–and each process requests at most one resource of each type (In other words, each process requests at most one person and at most one piece of equipment.)

1. List all the tasks that need just one person and one tool, and ignore tasks that don't fit this pattern.
2. Draw two groups (bipartite graph), one for people and one for tools.
3. Connect a person to a tool with a line if a task requires both that specific person and that specific tool.
4. Now, find a matching where each person and each tool is connected to at most one task.
5. If we can find as many matches as there are tasks, then we can start all tasks at once.

(d) The special case of the problem when each resource is requested by at most two processes.

The special case of the Resource Reservation Problem where each resource is requested by at most two processes is NP-complete.

This is because it can be reduced from the Independent Set problem, which is also NP-Complete, meaning there's no known fast way to solve it for all possible scenarios.

2. *Kleinberg, Jon. Algorithm Design (p. 506, q. 7).* The 3-Dimensional Matching Problem is an NP-complete problem defined as follows:

   Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T such that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

   Since 3-Dimensional Matching is NP-complete, it is natural to expect that the 4-Dimensional Problem is at least as hard.

   Let us define 4-Dimensional Matching as follows. Given sets W, X, Y, and Z, each of size n, and a collection C of ordered 4-tuples of the form $(w, x, y, z) \in W \times X \times Y \times Z$, do there exist n 4-tuples from C such that each element of $W \cup X \cup Y \cup Z$ appears in exactly one of these 4-tuples?

   Prove that 4-Dimensional Matching is NP-complete. Hint: use a reduction from 3-Dimensional Matching.

1. Verifying in Polynomial Time (NP):
We can check if each element from the four sets appear exactly once in polynomial time, confirming 4-dimensional matching is in NP.

2. Hard as the 3-Dimensional Problem (NP-Hard)
- Reduction Process: Transform a 3-Dimensional Matching Problem into a 4-Dimensional problem into a 4-Dimensional one.
- Constructing 4-tuples: Create a new set W of n distince elements. Pair each 3-tuple from the 3-Dimensional Problem with a unique element from W, forming 4-tuples.
- Bijection Equivalence: This bijection ensures that solving the 4-Dimensional problem is equivalent to solving the 3-Dimensional problem.

If there's a solution to the 3-Dimensional problem, the original 3-tuples are a solution to the 4-Dimensional problem. Conversely, if there's a solution to the 4-Dimensional problem, the original 3-tuples are a solution to the 3-Dimensional problem.

This reduction proves that 4-Dimensional Matching inherits the NP-completeness from the 3-Dimensional Matching.

3. *Kleinberg, Jon. Algorithm Design (p. 507, q. 6).* Consider an instance of the Satisfiability Problem, specified by clauses $C_1, ..., C_m$ over a set of Boolean variables $x_1, ..., x_n$. We say that the instance is monotone if each term in each clause consists of a nonnegated variable; that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$. Monotone instances of Satisfiability are very easy to solve: They are always satisfiable, by setting each variable equal to 1.

   For example, suppose we have the three clauses

   $$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

   This is monotone, and the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

   Given a monotone instance of Satisfiability, together with a number k, the problem of *Monotone Satisfiability with Few True Variables* asks: Is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

1. The problem is in NP:
Given a satisfying assigament with K true variables, we can verify it quickly by checking each clause.

2. Showing NP-Hardness:
Use a problem like Vertex Cover, which is NP-complete.

Create a clause for each edge in the Vertex Cover problem where each vertex becomes a variable in the clause. Since Vertex Cover is NP-complete, finding a vertex cover with K vertices corresponds to satisfying the clauses with K true variables.

If a vertex cover of size K exists, then the corresponding variables can be set to true to satisfy the Monotone Satisfiability instance. Conversely, if a Monotone Satisfiability instance is satisfiable with at most K true variables, the set of true variables corresponds to a vertex cover of the graph.

This reduction proves that if we can solve the Monotone Satisfiability with Few True Variables problem quickly for all cases, we could also solve the Vertex Cover problem quickly, which is known to be NP-complete.

4. *Kleinberg, Jon. Algorithm Design (p. 509, q. 10).* Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites and they've come across the following Strategic Advertising Problem.

   A company comes to them with the map of a Web site, which we'll model as a directed graph $G = (V, E)$. The company also provides a set of t trails typically followed by users of the site; we'll model these trails as directed paths $P_1, P_2, ..., P_t$ in the graph G (i.e., each P_i is a path in G).

   The company wants WebExodus to answer the following question for them: Given G, the paths $\{P_i\}$, and a number k, is it possible to place advertisements on at most k of the nodes in G, so that each path P_i includes at least one node containing an advertisement? We'll call this the Strategic Advertising Problem, with input $G, \{P_i : i = 1, ..., t\}$, and k. Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple.

   (a) Prove that Strategic Advertising is NP-Complete.

1. The problems in NP:
Given a set of nodes with ads, we can quickly check if each P_i has an ad.

2. NP-Hardness:
Reduce a known NP-complete problem to this one.

- For each edge in Vertex Cover, define a path in Strategic Advertising.
  A vertex cover that touches each edge corresponds to an ad placement touching each path.
- If there's a vertex cover of size k, then placing ads on the corresponding nodes will ensure each P_i has an ad.

Solving strategic advertising for any instance would solve Vertex Cover, and since Vertex Cover is NP-hard, so is Strategic Advertising. Thus, Strategic Advertising is NP-Complete.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising Problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input $G, \{P_i : i = 1, ..., t\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes.
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most polynomial number of steps, together with at most polynomial number of calls to the algorithm $\mathcal{S}$.

1. Call Algorithm S to check if a solution exists with k nodes.
2. Iterative Node Testing:
   - If S says no solution exists, then report the same.
   - If S says a solution exists, remove one node from the graph and re-run S to see if a solution still exists with one fewer node.
   - Repeat the process, trying to find the minimum set of nodes needed.

Each call to S takes polynomial time, and we make at most k calls, so the total time is polynomial.

This greedy-style approach iteratively improves the solution until it finds the minimum set or confirms none exists within k.

This approach uses S as a black box, leveraging its correctness to iteratively find a minimal solution or determine that no solution exists.