Course: Algorithm Analysis
Module: NP-Complete Problems

Gunnar Miller

March 8, 2025

Contents

1 NP-Complete Resource Reservation

A system consists of m resources, along with n processes which each request a subset of the resources. Each resource may be assigned to no more than one process. For a given assignment, a process is active in that assignment if it is allocated all the resources it requested, otherwise it is blocked. The Resource Reservation problem is do determine if, given an integer $k > 0$, m and a set of requested resources for each of the n process, a resource assignment exists that allows k total processes to be active. For each of the following resource reservation problems, give a polynomial time algorithm or prove it is NP-complete.

1.1 General Case

The general Resource Reservation Problem as described above:

Proving NP-ness: Given a list of k processes, need simply check that there are no resource conflicts. We proceed by combining sorted version of all the resource lists: if we encounter a duplicate entry during this process, the solution is invalid, otherwise it is valid. Each resource list is of length at most m each process, so we can sort a list in ($O(m \log m)$ time) and then merge the sorted lists (in the same fashion as mergesort) in $O(m)$ time, taking $O(nm \log m)$ time to combine all lists, so checking the solution is polynomial in n and m.

Proving Completeness:

We will proceed by demonstrating that Maximum Independent Set can be reduced to this problem. Actually, it is more convenient to prove Maximum Independent Set reduces to the special case found in Part C, but clearly if the special case is NP-Complete, so is the general problem.

Let $G(V, E)$ be a graph with $|V| = n$, For each $v_i \in V$ we define a process $p_i \in P$ where P is our set of processes. Then we define $f : P \to v$ as $f(p_i) = v_i$, noting that it has inverse $f^{-1}(v_i) = p_i$. For each edge $(v_i, v_j) \in E$ we define a resource $r_{i,j}$ whose resource list is p_i, p_j.

A solution for this problem will consist of a list P of k processes that can be simultaneously active. Then $f(P)$ will be an independent set of size k in V, because for any two processes p_a and p_a we can have $p_a, p_b \in P$ only if $(f(p_a), f(p_b)) \notin E$. Conversely, for any independent set $S \subseteq V$, we can be sure $f^{-1}(S)$ consists only of processes that share no resources and thus can be simultaneously active. Thus a solver for this problem can be used to find an independent set of size k in G if one exists.

The time complexity of our reduction is straightforward: mapping all our graph vertices to processes is $O(n)$, mapping all our edges to resources is $O(|E|)$ which is bounded by $O(n^2)$ and transforming a solution of this problem into an Independent Set is $O(k)$ where $k \leq n$. Thus our maximum time complexity is $O(n^2)$ and this problem is NP-Complete.

1.2 Only Two Processes

The special case where $k = 2$:

A sufficient (though probably not optimal) solution to this case is merely to use the solution-checking procedure outlined in part a to check every pair of processes until a solution is found. With n total processes there are $n(n-1)/2$ possible pairs, and checking only two processes under this procedure is $O(2m \log m)$ (since we are combining only 2 lists not n lists). Thus the entire algorithm takes $O(n^2 m \log m)$ and is of polynomial time complexity in m and n.

1.3 Limited Resource Sharing

The special case where each resource is requested by at most two processes:

This special case is NP-Complete. The proof from part a) uses a reduction of the Maximum Independent Set problem that already assigns only two processes to each resource, as thus holds for this case just as well as for the general problem.

2 Monotone SAT with Few True Variables

A formula on Boolean variables $x_1, x_2, ...x_n$ is **monotone** if the negations system ¬is not applied to any term of the formula. Given a monotone formula F on $x_1, x_2, ...x_n$ in Conjunctive Normal Form, the Monotone Satisfiability with Few True Variables problem asks whether there is a satisfying assignment for F in which at most k of $x_1, x_2, ...x_n$ are set to True.

Proving NP-ness:
First we observe that solutions for monotone satisfiability can be checked by exactly the same algorithm that checks solutions for ordinary SAT problems. So we can conclude without further effort that this problem is NP.

Proving Completeness:
To prove completeness we will show that set cover $\leq_n$ monotone satisfiability. Suppose we have some set U of size m with a set S of n subsets $S_1, S_2, ..., S_n$. We define a set $B = \{x_1, x_2, ...x_n\}$ where each x_i is a boolean variable, and a function $f : S \to B$ by $f(S_i) = x_i$ which we note has inverse $f^{-1}(x_i) = S_i$. For each element $u_j \in U$ we define set $X_j = \{f(S_i) | u_j \in S_i\}$, noting that $X_j \subseteq B$. Then we define the function h mapping elements of U to disjunctions of variables in B as follows: $h(u_j) = \bigvee_{x \in X_j} x$, with the variables in each disjunction listed in increasing order of index to ensure h is well-defined. Finally we define $F = \bigwedge_{j=1}^{|U|} h(u_j)$, noting that F is a monotone boolean CNF formula.

Suppose we have a solver that can solve the monotone satisfiability with few true variable problem on F, determining whether there is a solution k true variables. If there is a solution, then some variable assignment b must satisfy F with a total of k of the x_i set to True. Let X^* be the set of x_i that are set to True under b. Then $f^{-1}(X^*)$ is a set of k subsets from S, which we claim covers U. To see this, consider some $u_i \in U$, and observe that, by construction, $h(u_i)$ is a clause of F, which must be satisfied by some $x_j \in X^*$. But if x_j is in $h(u_i)$ then $S_j = f^{-1}(x_j)$ must contain u_i, by definition. So we have found a subset $S_j \in S$ that contains u_i and S must cover U.

Conversely, suppose that we have some $S^* \subseteq S$ of size k that covers U. Then $f(S) \subseteq B$ with size k. Each clause q of F must be of the form $q = h(u_i)$ for some $u_i \in U$. We must have $u_i \in S_j$ for some $S_j \in S^*$ by our coverage assumption, which implies $f(S_j)$ is one of the disjunctives in q. So for each clause q of F there exists $f(S_j) \in f(S)$ that is a disjunctive of q, so setting the k elements of $f(S)$ to True will satisfy F.

Finally we must check time complexity. Defining B is $O(n)$. Defining each X_j is $O(mn)$ as we must check membership in each of n subsets of size at most m, which means defining the all the X_j is $O(m^2 n)$. Constructing each $h(u_j)$ is $O(n)$ its size is proportional to $|X_j|$ which is bounded by n. Constructing F requires the conjunction of all m of the $h(u_j)$, which is thus $O(mn)$. Recovering our solution from a solution of the Boolean satisfiability problem simply requires forming the image of our k-sized set X^* under the constant-time function f^{-1} and is thus $O(k)$. Thus the entire reduction is $O(m^2 n)$, which is polynomial time. Thus we conclude that Set Cover $\leq_p$ Monotone Satisfiability with Few True Variables.

3 4-D Matching

The 3-dimensional matching problem is a known NP-Complete problem defined as follows: given disjoint sets X, Y, 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 so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

The 4-dimensional matching problem is defined equivalently: Given disjoint sets W, X, Y, Z, each of size n, and given a set of $T \subseteq W \times X \times Y \times Z$ of ordered 4-tuples, does there exist a set of n 4-tuples in T so that each element of $W \cup X \cup Y \cup Z$ is contained in exactly one of these 4-tuples? Prove the 4-D matching problem is NP-complete.

Proving NP-ness:
The solution to a particular instance of the 4-D matching problem will look like a set T^* of n 4-tuples, which is a subset of some set T, which in turn is a subset of the Cartesian Product of size-n sets W, X, Y and Z. To verify that T^* is a solution, we need to do two things: check that $T^* \subseteq T$ and verify that T meets the condition of containing each element in $W \cup X \cup Y \cup Z$ in exactly one such 4-tuple. Checking $T^* \subset T$ can be done simply by checking element-wise if members of T^* are also in T. As T could be of size n^4 and there are n elements of T^* to check, this may take up to $O(n^5)$ time. To check the second condition, we simply create an instance of the set $W \cup X \cup Y \cup Z$ and start pulling elements out of them as we encounter them as coordinates in T^* until we run out of one or the other.

1. For a given element t of T^*, start by removing t from T^*.
2. For each coordinate of t in step 1, remove that coordinate from $W \cup X \cup Y \cup Z$. If it does not exist in $W \cup X \cup Y \cup Z$ (including if $W \cup X \cup Y \cup Z$ is empty), end immediately and conclude that T^* is not a valid solution.
3. Repeat steps 1 and 2 until T^* is empty. If $W \cup X \cup Y \cup Z$ is now also empty, T^* is a solution, otherwise it is not.

In step 2 we are removing four elements from a set of size (at most) $4n$, so each removal and each iteration of step 2 are $O(n)$. Step 1 will repeat n times, so this whole section is $O(n^2)$. Combined with the potentially $O(n^5)$ process of subset checking, our solution checking is polynomial time.

Proving Completeness:
We proceed by showing that 3-D matching $\leq_p$ 4-D matching.

Consider a general instance of the 3-D matching problem, where X, Y and Z are sets of size n and let $T \subseteq X \times Y \times Z$.

As we know nothing about the elements of X, Y and Z, it is possible that some of them are natural numbers. Let us define $p = 0$ if $(X \cup Y \cup Z) \cap \mathbb{N} = \emptyset$ and $p = \max((X \cup Y \cup Z) \cap \mathbb{N})$ otherwise. Now we can define $W = \{a \in \mathbb{N} | p < a \leq p + n\}$, noting that $|W| = n$ and W, X, Y and Z are all disjoint. Consider the set $T' = W \times T$ and observe that $T' \subseteq W \times X \times Y \times Z$. Then T' is a valid candidate for the 4-D matching problem over W, X, Y and Z.

Now we define the projection function $f : T' \to T$ by $f(a, x, y, z) = (x, y, z)$. A solver for the 4-D matching problem will be able to determine if there exists $T^* \subseteq T'$ in which each element of $W \cup X \cup Y \cup Z$ appears as a coordinate exactly once. But if such a T^* exists, then the image of T^* in T under f must be a solution to the original 3-D matching problem on X, Y and Z. We can be certain of this because none of the W coordinates of elements in T^* appeared in any of X, Y or Z (since the sets were disjoint) and each element of X, Y and Z was required to appear exactly once in coordinates of elements in T^*, all of which were preserved by f. Conversely, suppose a solution $T^\#$ to the 3-D matching problem exists. Then for any ordering of the elements of $T^\# = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$, clearly $\{(p+1, x_1, y_1, z_1), (p+2, x_2, y_2, z_2), ..., (p+n, x_n, y_n, z_n)\}$ is a solution to the corresponding 4-D matching problem. So we are guaranteed to be able to find a solution to the 3-D matching problem in this fashion if one exists.

Finally we'll bound our time complexity to prove this reduction is polynomial. Finding p is linear in n as it only requires a single search through $X \cup Y \cup Z$. As T could have up to n^3 elements, forming T' is potentially $O(n^4)$. A solution for the 4-D problem is still always size n and applying the projection function takes a constant time per element, and is thus $O(n)$. So overall this reduction is $O(n^4)$. Thus a solver for the 4-D matching problem can be used to solve an arbitrary instance of the 3-D matching problem, and 3-D matching $\leq_p$ 4-D matching.

4 Evasive Path Problem

The Evasive Path Problem is defined as follows: given a graph $G(V, E)$ with source and destination nodes s and t, and subsets $Z_1, Z_2, ..., Z_k \subseteq V(G)$, decide whether there is a path from s to t that includes at most one node from each Z_i. Prove the Evasive Path Problem is NP-Complete.

Proving NP-ness:
Consider a solution to the evasive path problem in the form of a sequence of vertices $\{v_m\} = \{s, v_1, v_2, ..., v_p, t\}$ where $v_i \in V$ for all i (and thus $p + 2 \leq n$ where $n = |V|$). To verify this a solution, we need to prove that $\{v_m\}$ is an $s - t$ path that includes no more than 1 point from each of our subsets Z_j. For the first, we need to check each element in $\{v_m\}$ in order (except t) to verify that it has an edge to the next node in the path. We are examining $p + 1 < n$ nodes and some number of edges, so this it $O(|E| + n)$, which we can replace with the slightly weaker bound $O(n^2)$ since $|E| \leq n^2$. Next we need to check each subset S_j to ensure that none contains more than one element of $\{v_m\}$. Both sets we are comparing have sizes bounded by n, so we need make at most n^2 comparisons for each subset, thus checking all k subsets is $O(kn^2)$. So checking our solution is $O(kn^2) + O(n^2)$ which is just $O(kn^2)$, which is polynomial in both n and k.

Proving Completeness:
To prove completeness, we will show Independent Set $\leq_p$ Evasive Path. Consider an instance of the Maximum Independent set problem defined by a graph $G(E, V)$ with $(|V| = n)$ and an integer k where we wish to determine if there is an independent set of size k in G. We begin by constructing a graph $G'(E', V')$ from G as follows:

For each node $v_i \in V$ we define k nodes $v_{i,1}, v_{i,2}...v_{i,k}$ in V'. We will henceforth refer to a subset of these nodes for which the second coordinate is held constant as a layer of G' (with symbol L_x), so for example $L_j = \{v_{1,j}, v_{2,j}, ..., v_{n,j}\}$ is be the jth layer of G'. In addition to the kn nodes in these k layers, we add a source node s and a target node t which we will not consider to be part of any layer. For each layer we define a function $f_j : V \to L_j$ by $f_j(v_i) = v_{i,j}$. We also define a projection function $g : V' \to V$ by $g(v_{i,j}) = v_i$.

We will define the edges in E' as follows: there will be one edge from the source to each node $v_{i,1}$ in layer 1. For every $j < k$, there will be an edge from every node in layer j to every node in layer $j + 1$. The will be an edge from every node in layer k to t. We observe that this structure means an $s - t$ path must include one node in every layer (for $k + 2$ nodes at minimum). Further, because a node in layer j is adjacent to no other nodes in layer j but every node in layer $j+1$, a path that is longer than $k+2$ nodes must "backtrack", moving from a higher layer to a lower layer at some point. For any such path we can cut out the backtrack, replacing $\{...v_{a,j}, vb, j - 1, ...vc, j, vd, j + 1\}$ with $\{...v_{a,j}, vd, j + 1\}$, ending up with a shorter path that is a strict subset of the longer path, an operation that we can perform in time proportional to the length of the backtrack (and thus $O(kn)$ at most). We'll refer to this as a "shortest path reduction."

For each edge $(a, b) \in V$ (where (a, b) connects vertices v_a and v_b) we define the subset of V', $Z_{a,b} = \bigcup_{j=1}^{k} \{v_{a,j}, v_{b,j}\}$, We observe that taken together G' and our collection of subsets $Z_{a,b}$ constitute the necessary structure for the Evasive Path Problem.

Suppose we have a solver which can solve the Evasive Path Problem in polynomial time. Then it can determine whether there is an $s-t$ path in G' that includes no more than one node from each $Z_{a,b}$. Suppose that such a path exists. Then a shortest path reduction $P = \{s, p_1, p_2, ...p_k, t\}$ exists, which must also be a solution to the Evasive Path Problem as it is an $s - t$ path and a subset of our assumed solution. We claim that the set $Q = \{g(p_1), g(p_2), ..., g(p_k)\}$ is an independent set of size k in G. Clearly it is of size k. So see that the all the vertices are independent, suppose that some pair of vertices $g(p_c)$ and $g(p_d)$ shared an edge. Then by construction, our collection of $Z_{a,b}$ would include $Z_{g(p_c),g(p_d)}$, defined so that $p_c, p_d \in Z_{g(p_c),g(p_d)}$, contradicting our assumption that they are part of our Evasive Path solution. Thus no pair of vertices in Q may share an edge, and Q is an independent set.

Conversely, suppose there exists an independent set $Q = \{q_1, q_2, ..., q_k\}$ in G. Then we can clearly form an $s-t$ path in G' simply by applying our f_i functions to the elements of Q in order: $P = \{s, f_1(q_1), f_2(q_2), ..., f_k(q_k), t\}$. None of the elements of Q share an edge by assumption, so it's not possible for any two of them to be in the same subset Z, and thus P is a solution to the Evasive Path problem. Note that it is not a *unique* solution– at a minimum, any reordering of Q would produce another solution–but its existence guarantees that our solver for Evasive Path will find us *some* solution that maps to a particular solution to the Independent Set problem, provided the latter exists.

Finally, we'll show that our reduction is of polynomial time-complexity in n and k. Defining edges and vertices takes constant time, we define $nk + 2$ vertices and $nk(n - 1) + 2n$ vertices, for a time complexity of $O(n^2k)$. Each subset has $2k$ elements, and we define $|E|$ subsets (upper bounded by n^2) so defining our

subsets is also $O(n^2k)$. Transforming an Evasive Path solution into an Independent Set solution takes k constant-time calls of our projection function, and is thus $O(k)$. Thus we can reduce Independent Set to Evasive Path in polynomial time and Maximum Independent Set $\leq_P$ Evasive Path.

5 Geography on a Graph

The graph-theory version of the geography game can be stated as follows: given a digraph $G(V, E)$ and a starting node $s \in V$, the first player picks an edge leaving s and follows it, with the destination node becoming the current node. Thereafter each player takes turns picking an edge leaving the current node, following it to the new current node. Neither player may pick a node that has previously been the current node. A player wins the game if their opponent starts a turn with no legal moves.

Give a polynomial-time algorithm for determining which player has a winning move in the special case when the graph is a Directed Acyclic Digraph.

Idea:

Playing on a DAG makes solving the game much simpler, as it's no longer possible to re-visit nodes: the only way the game can end is to reach a node with no outgoing edges (which we'll call an "end node"). If we place our graph in topological order, all of our end nodes will be higher-numbered than any non-end node, allowing us to easily verify a winning state: if a player ever moves to a node with number equal to or greater than our lowest-numbered end-node, they win.

The key intuition for solving this problem is that realizing that each move can be considered the start of a new game on a smaller subgraph: specifically the graph of all nodes of equal or higher topological order to the current node. For brevity we will refer to a *node* having a "winning strategy" if a player taking the first move on that node has a winning strategy. A player never wants to jump to a node with a winning strategy (as their opponent will be the current player on that node), so a given node has a winning strategy if and only if it has an edge to at least one node that does *not* have a winning strategy. Since nodes can only have edges to nodes of higher order, we can solve the whole graph in a single pass by working from the highest order to the lowest, writing whether each node has a winning strategy by checking its successors.

Setup:

We'll assume we start with our n nodes numbered in topological order, with a 0-indexed, n-length array V holding adjacency lists for the nodes, arranged in the same order. Specifically, each entry $V[i]$ of V contains a list of nodes to which node i has an outgoing edge (with end nodes holding empty lists). We will assume without loss of generality that the starting node for the game is always node 0: any nodes of lower topological order than the starting node are unreachable and can have been dropped from the list while constructing V without impacting the final solution. We define a 0-indexed n-length array S, with all entries initialized to 0, to hold the solution information for our subgraphs. An entry of 0 will indicate that a player who is current on this node does not have a winning strategy, and entry of 1 will indicate that they do.

Algorithm:

```
for k = 1 to n:
    for e in V[n − k]:
        if S[e] = 0:
            S[n − k] = 1
        end if
    end for
end for
```

Analysis:
We prove correctness by induction on k. For the base case, let $k = 1$. Then the node $n - 1$ examined in this iteration of the outer for-loop is the node of highest topological order in the graph, as such it must be an end node. There is no winning strategy from an end node, and thus the proper label for the node is 0. As node $n - 1$ can have no outgoing edges, the edge list $V[n - 1]$ will be empty and the inner for-loop will exit immediately, leaving $S[n - 1]$ initialized to the correct label of 0.

For the inductive step, suppose $k \leq n$ and that all values of $S[n - i]$ are correctly labeled to indicate whether a node has a winning strategy for all $i < k$. We have two cases to examine:

Case 1: There exists some node p with $n-k < p \leq n$ for which $S[p] = 0$ and $p \in V[n-k]$

We know the label for $S[p]$ is correct, since all labels between $n-k$ and n are correct bv assumption. There is an edge from node k to node p, so starting on node k Player A has a winning strategy in jumping to node p. So the correct label for node k is 1. But having $p \in V[n-k]$ with $S[p] = 0$ are exactly the conditions for the algorithm to set $S[k] = 1$, which we have just shown is the correct label.

Case 2: No such p (as defined in Case 1) exists

In this case the current player must either jump to a higher-numbered 1-labeled node (the label being correct by assumption), giving their opponent a winning strategy, or they must be on an end node already. Either way, the correct label for node k is 0. As we initialized S with $S[k] = 0$, we need merely notice that with no such p as defined in Case 1, *if* statement in the inner loop of the algorithm will never trigger, so the value of $S[k]$ will never be changed from its correct starting value.

As Case 1 and Case 2 cover all possibilities and both show the algorithm will produce correct labels, we conclude that the algorithm will always produce the correct value for $S[n-k]$, provided the values of $S[n-i]$ are correct for all $i < k$, proving our inductive step.

Having proved our Base Case and our Inductive Step, we conclude that the algorithm will correctly determine if the player starting at node i has a winning strategy for all $1 \leq ik$. In particular, it will correctly determine whether the player starting the game at node 1 has a winning strategy or if instead their opponent does.

To prove efficiency we note that the inner for-loop executes in constant time (it only does a single comparison and assignment) and that neither loop can execute more times than there are nodes in the graph (that is, each loop is $O(n)$). So the whole for loop structure is $O(n^2)$. Our setup requires the node list to be in topological order and a (not necessarily ordered) outward edge list for each node, but both of those things can be computed from other standard graph representations in $O(n^2)$ time, so we can still conclude that the entire algorithm is $O(n^2)$