Greedy Algorithm

Exchange Argument:

1. Define Greedy Solution as X to an optimal solution, X_{opt}.
2. Compare Solution that $X \neq X_{opt}$
3. Exchange Piece: How to transform X_{opt} into X by exchanging some piece of X_{opt} for some piece of X without increasing/decreasing cost of X_{opt}.
4. Iterate until $X = X_{opt}$.

Solution: Consider the following greedy algorithm. We will use x to denote the amount of value left. Initialize $x = n$; while $x > 0$, output the stamp with the largest value that is at most x and subtract its value from x.

We will prove correctness by an exchange argument. Let S be the sequence of stamps output by the greedy algorithm and S^* be an optimal set of stamps, sorted from largest value to smallest. If $S = S^*$, then S is optimal. Let v_i and v_i^* be the value of the i-th stamp in S and S^*, respectively. Consider the first index i such that the i-th stamp in S has a different value from the i-th stamp in S^*. Let x_i be the remaining value left after the first $i-1$ stamps. Since greedy chooses the stamp with the largest value that is at most x_i for its i-th stamp, it must be that $v_i > v_i^*$. Moreover, since each denomination can be divided evenly by every smaller denomination, there exists a subset T of S^* each with value less than v_i whose total value is exactly v_i. Such a subset must consist of at least two stamps. Thus, replacing T with a single stamp of value v_i results in a solution with fewer stamps than S^*. But this contradicts the optimality of S^*. Thus, we conclude that $S = S^*$ and that greedy is optimal.

The running time of this algorithm is $O(n)$. There are at most n iterations as x decreases by at least 1 per iteration. In each iteration, it takes $O(1)$ time to determine the value of the largest stamp at most x and to increment s by 1.

Solution: Suppose you have a sorted list of the white dot positions and the black dot positions. Then you should match the ith white dot with the ith black dot.

To prove that the algorithm is optimal consider an optimal solution X_{opt} and the solution X produced by the greedy algorithm. Again, if $X = X_{opt}$ then we are done. Otherwise there must exist a matched pair w_i, b_j with $i \neq j$ in X_{opt}. We call such a pair an inversion. To simplify the argument we assume that $i < j$ and that w_i is the leftmost white point that is part of an inversion.

Since w_i is matched to b_j, b_i must be matched to some w_k with $k > i$.

We argue that we can remove an inversion in the optimal solution X_{opt} and the cost of the matching will not increase. First consider the cost of matching these two pairs in the optimal solution:

$$C := C(w_i, b_j) + C(w_k, b_i) = |w_i - b_j| + |w_k - b_i|.$$

If we replace these two pairs with (w_i, b_i) and (w_k, b_j) we would get:

$$C' := C(w_i, b_i) + C(w_k, b_j) = |w_i - b_i| + |w_k - b_j|.$$

There are three cases depending on w_k's relative position between b_i and b_j.

1. When it's *smaller than both of them* the cost stays the same: indeed, then $w_i < w_k < b_i < b_j$, and so
$$C - C' = b_j - w_i + b_i - w_k - (b_i - w_i + b_j - w_k) = 0$$
2. When its *smaller than just* b_j, we have $b_i \leq w_k < b_j$ (and since $w_i < w_k$, we also have $w_i < b_j$), and
$$C - C' = b_j - w_i + w_k - b_i - (|w_i - b_i| + b_j - w_k) = 2w_k - (b_i + w_i + |w_i - b_i|)$$
Since $b_i + w_i + |w_i - b_i| = 2\max(w_i, b_i)$ (check it! $a + b + |a - b| = 2\max(a, b)$) and $w_k \geq \max(w_i, b_i)$ (since $w_k > w_i$ *and* $w_k \geq b_i$), we get
$$C - C' = 2w_k - 2\max(w_i, b_i) \geq 0.$$
3. When it's *bigger than both of them*, we have $b_i < b_j \leq w_k$, and so
$$C - C' = |w_i - b_j| + w_k - b_i - (|w_i - b_i| + w_k - b_j) = |w_i - b_j| - |w_i - b_i| + b_j - b_i$$
Now, by the triangle inequality, $|b_i - b_j| + |b_j - w_i| \geq |b_i - w_i|$, i.e., $|b_j - w_i| - |b_i - w_i| + |b_i - b_j| \geq 0$. and since $b_j - b_i = |b_i - b_j|$ as $b_j > b_i$, this last inequality is saying that
$$C - C' \geq 0$$
in this case as well.

Therefore we can remove the inversion in X_{opt} and guarantee that we still have an optimal solution. This argument can be iterated until we have no more inversions in the solution and, hence, the greedy solution must be optimal since it has no inversions.

DnC

Master Theorem

num. subprob — size of subprob — Time complexity of Divide & Conquering.

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

1) if $f(n) = O(n^{\log_b(a) - \varepsilon})$ for $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$
2) if $f(n) = \Theta(n^{\log_b(a)} \log^k n)$ for $k \geq 0$, then $T(n) = \Theta(n^{\log_b(a)} \log^{k+1} n)$
3) if $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$ & $af(n/b) \leq \delta f(n)$ for $\varepsilon > 0, \delta < 1$ then $T(n) = \Theta(f(n))$

1. Divide: break into subproblems that are themselves smaller instances of same type of the original problem.
2. Delegate: recursively solving these variable.
3. Conquer: appropriately combing (merging) problem.

Use of Induction Proof.

Solution: The algorithm starts at the root and does the following recursively. If the current vertex is an internal vertex with smaller weight than both of its children, return the current vertex as a local minima; otherwise, recurse on one of the children with smaller weight. If the current vertex is a leaf, then return the current vertex as a local minima.

To see why this is $O(\log n)$ time, note that we take $O(1)$ time to decide whether to return the current vertex as a local minima or to recurse on one of the two children, and whenever we recurse, we go down one level of the binary tree. Since a complete binary tree with n vertices has at most $O(\log n)$ levels, the total time taken by this algorithm is $O(\log n)$.

The proof of correctness is easiest to see without using induction (see below for an induction proof). Suppose, towards a contradiction, that the algorithm returns a vertex v that is not a local minima. By definition of the algorithm, v cannot be an internal vertex because the algorithm only returns an internal vertex u after checking if its weight is indeed smaller than all of its neighbors. Thus, v must be a leaf. But then the algorithm reached v only because the v's parent has larger weight than v. Since v is a leaf, its parent is its only neighbor. Thus, v is a local minima.

Here's a proof of correctness via induction.
Base case: $(n = 1)$ The algorithm correctly returns the single vertex as a local minima.
Induction case: $(n > 1)$. Suppose the algorithm is correct for complete binary trees with less than n vertices. If the root r is a local minima, then the algorithm correctly returns the root. Otherwise, suppose the algorithm recursed on r's child u, and let v be the vertex returned by the recursive call. There are two cases: either $u = v$ is a child of the root r or not. In the latter case, the correctness follows from the induction hypothesis. For the former case, the induction hypothesis only tells us that v is smaller than both of v's children. We still need to show that v is smaller than its parent r. But this follows from the fact that we recursed on v precisely because v is smaller than r.

Some Post-Processing Algorithm

Merge Sort $O(n\log(n))$
Closest Pair of Point $O(n\log(n))$
Maximum sum Contiguous Subarray $O(n)$
LCS $O(nm)$ / RNA $O(n^3)$

Recurrence Formula

$T(n) = 2T(n/2) + O(n) \Rightarrow O(n\log(n))$
$T(n) = 2T(n/2) + O(\log(n)) \Rightarrow O(n)$
$T(n) = 2T(n/2) + O(1) \Rightarrow O(n)$
$T(n) = T(n/2) + O(n) \Rightarrow O(n)$
$T(n) = T(n/2) + O(1) \Rightarrow O(\log(n))$
$T(n) = T(n-1) + O(n) \Rightarrow O(n^2)$
$T(n) = T(n-1) + O(1) \Rightarrow O(n)$

LCS DP

LCS

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max(M[i-1,j-1] + 1) & \text{if } X[i] = Y[j] \\ \max(M[i,j-1], M[i-1,j]) & \text{if } X[i] \neq Y[j] \end{cases}$$

DP

Time Complexity: num. sub problems.

1. Define Subproblem: OPT[i] is what...
2. Define Recurrence: define cases. i.e) selects job i / does not select job i.
3. Define Base Case: solve base cases
4. Evaluate 1-3 order

1. Subproblems:
So now we know that we need to keep track of both the number of campsites (i) and the number of days left (ℓ) to define a recurrence.
We thus re-define our subproblem as follows:

Let $M[i, \ell]$ be the optimal minimum maximum distance walked in a day to get to campsite i in ℓ days.

Satisfying Property 1:
To ensure we only have a polynomial number of subproblems we must again ask ourselves which subproblems are related to the original problem. For example, calculating the optimal solution for getting to newcastle in $k+1$ days won't help us figure out how to get there in k days. So we can narrow down which subproblems we need to $M[i, \ell]$ for all $0 \leq i \leq n+1$, and $0 \leq \ell \leq k$. This gives us $O(nk)$ subproblems, which is polynomial.

Satisfying Property 2:
The subproblems satisfy the second property because the solution to our original problem is in itself a subproblem, namely $M[n+1, k]$.

Satisfying Property 3:
When we visualise the solutions to our subproblems in a table (see Figure 2), we can see that the subproblems satisfy a natural ordering as well.

Satisfying Property 4:
Again, even though the first three properties of a good subproblem are satisfied, we won't know if this definition of our subproblems will work until we have tried to define a recurrence.

2. Recurrence:

We use the same approach as before.

Assume we have an optimal solution $M(i, \ell)$ to reach campsite i in ℓ days. Let j be the last campsite used before reaching our destination i. By camping at campsite j we spend a whole day. Therefore, we can find the optimal campsites to stay at before campsite j by finding the optimal solution to get to campsite j in $\ell - 1$ days ($M(j, \ell-1)$). From here, the minimum maximum distance walked on any day to reach campsite i in ℓ days will be the maximum between *the minimum maximum distance walked to get to j* ($M(j, \ell - 1)$) and *the distance between j and i* (see Figure 1). Let us denote the distance from campsite i to campsite j as $D(i, j)$. We can express this as follows:

$$M(i, \ell) = \max\{M(j, \ell-1), D(j, i)\} \quad (1)$$

So far we assumed we knew what the optimal solution is. In reality, we still need to figure out what j is. Since the solution is optimal we know that j must be the campsite to the left of i that minimises the expression above. Thus:

$$M(i, \ell) = \min_{j<i}\left\{\max\{M(j, \ell-1), D(j, i)\}\right\} \quad (2)$$

3. Base case: Now that we have our recurrence the next step is to define our base case. Again, we recall that our base case has to satisfy 3 main properties:

1. Trivial to solve without the need for smaller subproblems.
2. Can be solved in *constant time*
3. In a top-down approach, ensure it *will always be reached*. In a bottom-up approach, ensure you *will never require a smaller subproblem* to build up your solution.

Let's consider $i = 1$ and $\ell = 1$. The optimal solution to get to campsite 1 in 1 day is simply $D(0, 1)$ so that is indeed trivial and can be solved in constant time. However, we must again ask ourselves if it will always be reached. From our recurrence we can determine that we will need to compute $M(j, \ell - 1)$ for every $j < i$. Imagine we are trying to compute $M(i, 1)$ where $i > 1$. To compute this we must compute, for all $0 < j < i$, $M(j, 0)$. However, from here we can never get back to our base case where $\ell = 1$ since we always reduce the value of ℓ in our recursive call.

But let's think for a moment about the solution to $M(i, 1)$. Getting to campsite i in only 1 day seems as trivial as getting to campsite 1 in 1 day. The optimal solution will still be the distance between campsite 0 and campsite i. Hence, we can conclude that our base case doesn't need a specific value for i. It is simply $M(i, 1) = D(0, i)$.

4. Order of Evaluation: We have defined an $O(nk)$-sized table to store the solution to our subproblems (to avoid repeating any computations). Next, in order to turn the above into an algorithm we need to decide on which order to evaluate the subproblems in.

Justify the recurrence, base cases, how optimal solution can be obtained from og prob.

Solution: Let $S[n]$ denote the minimum number of stamps needed to make postage for n cents. We clearly then have

$S[0] = 0$

These cases above should be considered our "base cases" and we then work from these to compute values of $S[n]$ for higher values of n. The main idea, then is to consider what happens if we use a stamp of a particular value. For example, if we want postage of n cent, then we can clearly get it by taking one stamp of 1c, and $S[n-1]$ stamps (of appropriate values) to make up the remaining postage of $(n-1)$c. Or if we take a 7c stamp, then we need $S[n-7]$ stamps (of the right values) to make up the rest, and similarly if we take a 10c stamp. This idea is essentially the heart of the dynamic programming algorithm that we use.

add one more coin $S[j-1]$ → add one 1c

Set up the array S starting as above (with the values $S[i]$ for $i = 0, \ldots, 10$).
If $n > 10$ then $S[j] = 1 + \min\{S[j-1], S[j-7], S[j-10]\}$.
Thus, finding $S[n]$ takes $O(n)$. Now we can find $S[n]$ easily as above, but knowing this does not tell us the exact nature of the stamps we need. It would be nice to know that we can make 45c postage with six stamps, and that we need one 1c, two 7c, and three 10c stamps to do so. (For some values of n there could certainly be more than one way to do it. For example, we can get 63c with nine 7c stamps, or with six 10c and three 1c stamps.) Well, we can easily do this too with another array called, say, P. Then $P[n]$ will be a vector of length 3 that will tell us what stamps we need to make postage for nc. For example, we would have $P[45] = (1, 2, 3)$, meaning that we need one 1c, two 7c, and three 10c stamps. In general, if we have $P[n] = (a, b, c)$, then we take a 1c stamps, b 7c stamps, and c 10c stamps to make nc postage.
The calculation for $S[n]$ doesn't change. All we do is determine which denomination of postage we use and add it to the appropriate value of $P[n-1]$, $P[n-7]$, or $P[n-10]$. The running time is still $O(n)$.

Flow Network

MAX FLOW ≤ MIN CUT

s-t flow:

- Capacity Constraint: $e \in E$: $0 \leq f(e) \leq c(e)$
  e is saturated if $f(e) = c(e)$
- Conservation Constraint: $v \in V \setminus \{s, t\}$:
  $$\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

s-t cut is a partition (A, B) of V with $s \in A$, $t \in B$.

capacity of cut $(A, B) = \sum_{e \text{ out } A} c(e)$

Augmenting Path Algorithm.

→ s-t path in G_f called an augmenting path

b = bottleneck(P, F) ⇒ min capacity in residual graph

Minimum Cut is from original network

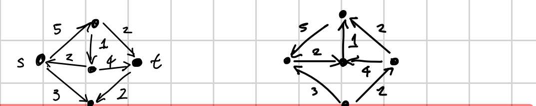
↳ Vertices reachable from s in the residual network for max flow.

- Ford-Fulkerson $O(nm^2)$ (choose any aug path time F)
- Ford-Fulkerson w shortest augmenting path will run in the order of $O((n+m)(nm^2))$
- Orlin's Algorithm: $O((m+n)(nm))$
- Edmonds-Karp: $O(m^2 \log(F))$ / General Augmenting Path $O(mn)$

$m\log(F)$ iter

↳ identical to Ford-Fulkerson, BUT defined aug path. (bfs)

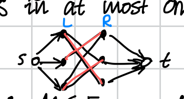
G (V, E) — Residual Network



Instance of X → Transform X to Y → Algo for MAX FLOW → Transform solution for Y to X → solution I

Application of MAX FLOW

- **Bipartite Matching:** $M \subseteq E$ is matching if each node appears in at most one edge.
  i.e) find max M (use Edmonds Karp)
  $G = (L \cup R, E)$

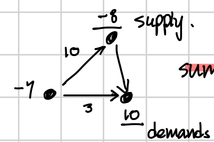


- **Perfect Matching:** $M \subseteq E$ is perfect matching if each node appears in exactly one edges M.
  $|L| = |R|$ capacity of flow $= |L| = |R|$

Let S be a subset of nodes, and let $N(S)$ be the set of nodes ~~adjacent~~ to nodes in S.

$|N(S)| \geq |S|$ for all subsets $S \subseteq L$

- **Marriage Theorem:** G has P.M iff $|N(S)| \geq |S|$
- **Circulation:** directed graph $G = (V, E)$
  edge capacity $c(e)$, $e \in E$
  node supply & demands $d(v)$
  $d(v) < 0$ $\quad d(v) > 0$



sum supplies = sum of demands

$$\sum_{v: d(v) > 0} d(v) = -\sum_{v: d(v) < 0} d(v)$$

Classify Problem. (Reduction)

$X \le_p Y$: Problem X polynomial reduces to problem Y

Transitivity : if $X \le_p Y$ and $Y \le_p Z$, then $X \le_p Z$

Equivalence : if $X \le_p Y$ and $Y \le_p X$, then $X \equiv_p Y$

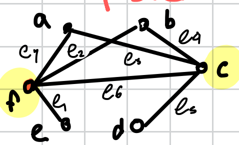
Decision Prob $\equiv_p$ Optimization Prob $\equiv_p$ Search Prob

- **Independent Set**: From $G = (V, E)$, a subset of vertices $S \subseteq V$ is I.S. if no two vertices of S share an edge.
- **Vertex Cover**: $S \subseteq V$ such that $|S| \le k$, and for each edge, at least one of its endpoints in S.

I.S $\equiv_p$ V.C ∵ $|V.C| = k$ iff $|I.S| = n - k$

- **Set Cover**: Given set U of elements, a collection $S_1, S_2, \ldots S_n$ of subsets of U & k that collection of $k = U$

V.C $\le_p$ S.C



$U = \{1,2,3,4,5,6,7\}$
$k = 2$
$S_a = \{3, 7\}$
$S_c = \{3, 4, 5, 6\}$
$S_f = \{1, 2, 6, 7\}$

- **Satisfiability** : literal : x or $\bar{x}$

clause : A disjunction of literals. $C_j = x_1 \vee x_2 \vee \bar{x_3}$

Conjunctive Normal Form (CNF) : a propositional formula Φ that is the conjunction of clause

$\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT : given CNF formula Φ, does it have a satisfying truth assignment. can assign True/False values so that Φ evaluates to true.

3-SAT : SAT each clause contains 3 literal

$(x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3)$

YES : $x_1 = T$, $x_2 = T$, $x_3 = F$

Proving NP-Completeness : show problem is in NP

~~provide~~ certificate, and polynomial time certifier

Proving NP-Hardness : ~~give polynomial time reduction~~ from NP-Complete problem to show that every NP reduces to it.

∴ every NP-complete problem can be reduced to/from ~~every other~~ ~~NP~~-complete problem.

Problem 3

Prove that CLIQUE is NP-complete by using a reduction from 3-SAT.

Guided Solution:

Proving Clique is in NP

Proving a problem is in NP is often very straight forward. To define a certificate we have to think about what defines a solution to the problem. Let a certificate for Clique be the graph $G(V, E)$, a subset of nodes $S \subseteq V$, and an integer limit k.

The certifier first checks that the size of S is k (it can do this in $O(k)$ time). It then iterates through every possible pair $\{u, v\}$ in S and checks that the edge (u, v) is in E (it can do this in $O(k^2)$ time).

Proving Clique is NP-Hard

Following the steps in the "guide for answering tutorial questions":

1. Identifying which problem to reduce from:

The question already tells us to reduce from 3-SAT, which is an NP-complete problem. But let's think about why this might be a good choice. Let's start by formally defining both decision problems:

Clique Given a graph G, does there exist a complete subgraph of size at least k?

3-SAT Given a CNF formula, does the formula have a satisfying truth assignment?

At the moment, it doesn't seem like these two problems have much in common. So let's re-phrase the problems as follows:

Clique Given a graph G, does there exist a subgraph such that each node in this subgraph is connected to every other node in the subgraph?

3-SAT Does there exist a truth assignment such that for each clause in the CNF formula the assignment of its variables allows all other clauses to have at least one literal that evaluates to TRUE?

With this new formulation we can see that the problems share some similarities, such as the fact that they both have components that need to be "satisfied" in some way.

2. Translating an instance of 3-SAT into an instance of Clique:

We can visualise the reduction as shown in Figure 1.

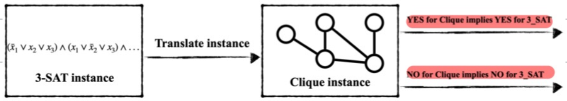


Figure 1: Visualisation of the reduction

To translate a CNF formula for the 3-SAT instance to a clique instance we must first identify the components that make up both a 3-SAT and a clique instance. We know from the lecture that a clique instance consists of a graph containing nodes and undirected edges. We also know that a 3-SAT instance consists of n literals and m clauses, which are each a disjunction of 3 literals.

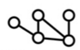
	3-SAT	Clique
Instance	$(\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge \ldots$	
Components	Literals $x_1, x_2, x_3, \ldots, x_n$ Clauses $(\bar{x}_1 \vee x_2 \vee x_3)$	Nodes Edges
Properties of a YES-instance	There exists a truth assignment such that every clause contains at least 1 TRUE	There exists a complete subgraph of size k

Figure 2: A breakdown of the components and properties of each problem

you can come up with an example yourself).

What we learn from this is that we actually want to be quite careful in how we define the components of the problem we are reducing to (clique in this case) to encode the properties we want to achieve in the instance we are reducing from (3-SAT in this case).

(Attempt 2)

The property we are trying to achieve is that Clique can only have the necessary edges for a complete subgraph of a certain size (in other words, a YES-instance for clique) if and only if there exists a truth assignment in the original 3-SAT instance s.t. every clause contains at least 1 TRUE (in other words, a YES-instance for 3-SAT). So a second approach from the description above could be to create a node for every clause and only connect them by an edge if there exists a truth assignment such that both contain at least 1 literal that evaluates to TRUE.

We are now approaching what we can assume is a valuable reduction so it's time to test how bulletproof it really is. In particular, can we think of an example of where a YES-instance might map to a No-instance or vice-versa?

Issues with attempt 2

Imagine a CNF where all clauses are all 8 possible permutations of 3 literals. No matter what we use as the truth assignment for the 3 literals, there will always be one of the clauses where all literals evaluate to FALSE. Hence, this is a NO-instance for 3-SAT. However, for every pair of clauses, *there exists* a satisfiable truth assignment such that the clauses both evaluate to TRUE. Hence, our resulting graph will have a complete subgraph of size 8 and thus the translated Clique instance will be a YES-instance.

So not quite bulletproof yet it seems. Let's unpack the cause of this issue. It seems that it is possible to assign edges to a clause that each require a different/contradictory truth assignment and hence the clique instance can get a complete subgraph even though the original 3-SAT instance was not satisfiable. How can we fix this?

(Attempt 3)

To avoid clauses having multiple edges that require different/conflicting truth assignments, we can do the following. For each clause c of F we create one node for every partial assignment to variables in c that satisfies c (i.e. the clause evaluates to TRUE). E.g., say we have:

$$F = (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \ldots$$

Then in this case we would create nodes like this:

$$
\begin{array}{ll}
(x_1 = 0, x_2 = 0, x_4 = 1) & (x_1 = 0, x_3 = 0, x_4 = 0) \ldots \\
(x_1 = 0, x_2 = 1, x_4 = 0) & (x_1 = 0, x_3 = 0, x_4 = 1) \\
(x_1 = 0, x_2 = 1, x_4 = 1) & (x_1 = 0, x_3 = 1, x_4 = 0) \\
(x_1 = 1, x_2 = 0, x_4 = 0) & (x_1 = 0, x_3 = 1, x_4 = 1) \\
(x_1 = 1, x_2 = 0, x_4 = 1) & (x_1 = 1, x_3 = 0, x_4 = 1) \\
(x_1 = 1, x_2 = 1, x_4 = 0) & (x_1 = 1, x_3 = 1, x_4 = 0) \\
(x_1 = 1, x_2 = 1, x_4 = 1) & (x_1 = 1, x_3 = 1, x_4 = 1)
\end{array}
$$

We then put an edge between two nodes if the partial assignments are consistent. This way, we can be certain that every complete subgraph of the resulting clique instance must correspond to a truth assignment with no contradictory assignments s.t. at least 1 literal evaluates to TRUE. Hence, we can conclude that a complete subgraph in this new graph must indeed imply the original 3-SAT instance was satisfiable.

Now that we have designed our reduction, let's argue it's correctness.

Arguing the correctness:

It's important to understand how to prove the correctness of a reduction so let's go through it in detail. As explained in the lectures, to argue the correctness of a reduction, we want to prove two claims (as visualised in Figure 3).

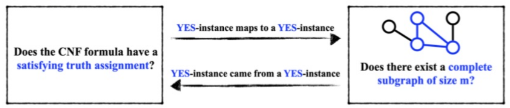


Figure 3: Mapping yes-instances.

1. A YES-instance of 3-SAT will always be transformed into a YES-instance of Clique.
2. Any YES-instance of Clique must stem from a YES-instance of 3-SAT.

If we can prove the above claims then we have proven that the blue arrows in the visualisation of our reduction in Figure 4 is correct.

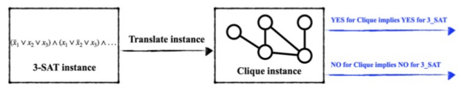


Figure 4: Proving correctness of a reduction.

Let's start with the first claim. In this claim we *assume we have a YES-instance of 3-SAT*. Hence, we can ignore all NO-instances of 3-SAT. The goal of the proof is to *use this assumption* along with the definition of *our transformation* to prove that the transformed instance must contain a complete subgraph of size at least m. So how do we do this.

Let's assume we have a YES-instance of 3-SAT.

The reduction will create a node for every possible partial assignment to variables that results in the clause having at least 1 variable that is assigned TRUE. Since our 3-SAT instance is a yes instance, we know that there exists a truth assignment, which ensures every clause has at least 1 literal that evaluates to True. Hence, this truth assignment will be amongst the truth assignments in our nodes. Since, by definition of the truth assignment being satisfiable, and the construction of our edges in the graph, there exist m nodes in the graph that are all connected to one another, there will exist a complete subgraph of size m in the new graph.

To prove our second claim we want to do something similar to the above only this time we can only assume that the reduced Clique instance is a YES-instance, but we cannot assume anything yet about our 3-SAT instance. So let's assume we have a YES-instance of Clique. This means that there exists a subgraph of size m such that all nodes in the graph are connected by an edge. Having this information, which properties can we conclude the original 3-SAT instance, from which this instance was reduced, must have? Well, since each node represents a partial assignment of variables in a clause and an edge means their truth assignments are compatible, we can conclude that, given their corresponding truth assignment, at least m clauses are all compatible with one another (we know they must all be distinct nodes since 2 nodes of the same clause cannot share an edge as they have conflicting partial truth assignments). Since m is the number of clauses in our original problem, we can conclude that there exists a truth assignments such that all m clauses are satisfied.

Confirming the reduction is polynomial-time

Finally, we need to argue that we can translate any arbitrary instances of 3-SAT to an instance of Clique in polynomial time.

Since our transformation creates a node for every possible truth assignment for every node, we know that our resulting graph will have at most 2^3 nodes for every clause. Hence the construction of the nodes takes $O(m)$ time. Since we create edges based on whether a truth assignment is compatible, we can do this in constant time for every pair of nodes. Hence, the construction of the edges will take at most $O(m^2)$ time.

This means the overall time for step 1 is $O(m^2)$, which is polynomial in the size of our input to 3-SAT. Hence, we can conclude that we have designed a polynomial-time reduction from 3-SAT to Clique.

Given a graph $G = (V, E)$, a distinguished subset of vertices $X \subset V$ and a number k, the Steiner Tree problem is to decide whether there is a set $S \subseteq V \setminus X$ of size at most k such that $G[X \cup S]$ is connected.

Prove that this problem is NP-complete.

← Steiner Tree

Solution: The problem is clearly in NP. The certificate is the set S of k nodes whose addition to X connects the set, which is trivial to check in polynomial time.

To show that it is NP-hard, we reduce 3-SAT to it. Recall that an instance of 3-SAT consists of a formula $\phi = C_1 \wedge \cdots \wedge C_m$ over variables $x_1, \ldots, x_n$ such that each clause C_i is the disjunction of three literals. The question is whether there is a truth assignment that satisfies all clauses.

We define a graph $G = (V, E)$ and a target k based on ϕ:

1. For each clause C_i we create a vertex $u_i \in X$; for each variable x_j we create a vertex $v_j \in X$; we also create a dummy vertex $d \in X$. Additionally, for each variable x_j we create two vertices v_j^T and v_j^F that belong to $V \setminus X$.
2. For each variable x_j, we create the edges (v_j^T, v_j) and (v_j^F, v_j). For each clause C_i, if C_i contains the literal x_j then we create the edge (u_i, v_j^T), while if C_i contains the literal $\bar{x}_j$ then we create the edge (u_i, v_j^F). Finally, we connect d with every v_j^T and v_j^F.
3. Finally, we set the target k to be n.

Notice that in order to connect v_j with the rest of X we must select either v_j^T or v_j^F into the set S; since $k = n$, exactly one of them must be chosen for each $j = 1, \ldots, n$. There is a one-to-one correspondence between truth assignments for the variables and a choice between v_j^T and v_j^F for each $j = 1, \ldots, n$, which will define our set S. It is not difficult to show that a truth assignment is satisfying if and only if for the corresponding S, the graph $G[X \cup S]$ is connected. (Notice that if we hadn't added the dummy node d the reduction would not work.)

Solution: (Sketch.) Let $G = (V, E)$ be an instance of 3-COLOR. Create a 4-color instance $G' = (V', E')$ where V' consists of V plus 4 new vertices v_1, v_2, v_3, v_4, and E' consists of E plus all the edges between the new vertices and all the edges between v_1 and V. The instance G' can be created in time $O(n + m)$ since we only need to copy over G, and create a constant number of new vertices and edges between them plus the n new edges between v_1 and V.

← 4 coloring NP-Comp

Suppose that G is 3-colorable. Consider a 3-coloring of G, call the colors R, G, B and let $c(v)$ denote the color used by vertex v. To get a 4-coloring c' of G', for each $v \in V$, we use the same color as c. Then, we color v_2, v_3, v_4 using the colors R, G, B and for v_1 we use a fourth color, call it Y. This is a valid 4-coloring of G': all the edges e in G have endpoints that are colored differently because c is a 3-coloring of G, the edges between v_1 and V are also fine because v_1 is colored differently than any vertex in G, and finally, the edges between v_1, v_2, v_3, v_4 are fine by construction.

Suppose that G' is 4-colorable. Then it must color V using at most 3 colors since otherwise, v_1 will have the same color as one of the vertices in V. Thus, G is 3-colorable.

(Alternative reduction.) Create G' by simply adding a single vertex v_1 that connects to all of V. Same proof as above.

Edge Disjoint Path : paths do not share edge

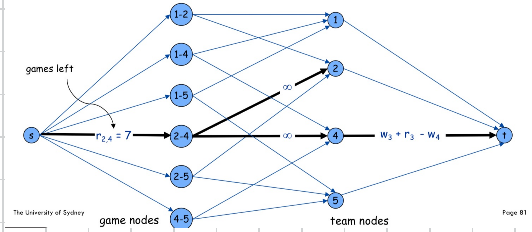
find max edge disjoint s-t paths

↳ equal to the max flow val

Baseball Elimination: Correctness

Theorem. Team 3 is not eliminated iff max flow equals the total number of games left.

- ⇐ (Need to translate flow into outcomes of remaining games)
- Suppose there exists an integral max flow with value = total # of games left
- Flow on middle edge from game node (x-y) to team node (x) equals # of matches between teams x and y that team i wins.
- Capacity on (x, t) edges ensure no other team wins too many games.



Segmented Least Square

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j} e(i, j) + c + OPT(i - 1) & \text{if } j > 0 \end{cases}$$

- **Hamiltonian Cycle** : Simple cycle C visits every node.

certificate : t = a permutation of n nodes

certifier : check perm contains each node in V exactly once & there is an edge between each pair of adjacent nodes in the permutation.

- **Graph Coloring** : a proper coloring is coloring of vertices such that every edge has 2-different colors at its endpoints. * 3-SAT REDUCES TO 3-COLO

*

- Constraint satisfaction problems: SAT, 3-SAT.
- Packing problems: SET-PACKING, INDEPENDENT SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3D-MATCHING, 3-COLOR.
- Numerical problems: SUBSET-SUM, KNAPSACK.

NP-Completeness.