Complexity Theory - Notes

Alperen Onur (z5161138)

August 2024

1 Polynomial Time Algorithms

A (sequential) algorithm is said to be polynomial time if for every input it terminates in polynomial many steps in the length of the input. Suppose the length of the input is n, this means the worst case complexity satisfies $T(n) = O(n^k)$ for some integer k.

Examples:

- merge sort requires $n \log n$ comparisons and since $n \log n = O(n^2)$ it runs in polynomial time, but
- The brute force algorithm for longest increasing subsequence takes at least $n2^n$ operations (each 2^n subsets of activities, check for increasing and update max length), so it is not in $O(n^k)$ for any integer k.

Length of input.
The length of input is the number of symbols needed to describe the input precisely. For example, if input x is an integer, then $|x|$ can be taken to be the number of bits in the binary representation of x ($log_2 x$).

Another example is the array A as an input. Then each entry $A[i]$ contributes $log_2 A[i]$ bits for a total of

$$\log_2 A[1] + .. + \log_2 A[n]$$

If the array entries are guaranteed to be each at most M, then this can be abbreviated to $n \log_2 M$. A polynomial time algorithm therefore runs in time polynomial n and $logM$.

0-1 Knapsack

The standard DP algorithm for this problem runs in $O(nC)$ with input size $n \log C$. Since $C = 2^{\log_2 C}$, we know that C isn't bounded by any power of $log_2 C$ and hence this is not a polynomial time algorithm. Such algorithms are pseudo-polynomial.

Weighted Graphs
If the input is a weight graph G with edge weights up to W, then G can either be described with an adjacency list or adjacency matrix. The adjacency list takes $O(E \log W)$ space, whereas the adjacency matrix takes $O(V^2 \log W)$.

Examples:

- For the maximum flow problem, there are E edges, each with capacity up to C, so the size of the input is $O(E \log C)$.
- Ford fulkerson runs in $O(E|f|)$. Since the value of the maximum flow could be up to EC, that is not a polynomial time algorithm.
- The Edmonds-Karp algorithm runs in $O(VE^2)$. Since $V \leq E + 1$, this is a polynomial time algorithm.

2 Class P

A decision problem is a problem with a YES or NO answer. A decision problem $A(x)$ is in class P if there exists a polynomial time algorithm that solves it.

3 Class NP

A decision problem $A(x)$ is in class NP if there is a problem $B(x, y)$ such that

- For every input x, $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true and,
- the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x only.
- y is called the certificate; y provides evidence for why x is a YES problem of A.
- B is called the certifier; B checks that evidence in polynomial time.

4 Satisfiability

A propositional formula in CNF form $C_1 \wedge C_2 \wedge .. \wedge C_n$ where each clause C_i is a disjunction of propositional variables or their negations, for example

$$(P_1 \vee \neg P_2 \vee P_3 \vee \neg P_5) \wedge (P_2 \vee P_3 \vee \neg P_5 \vee \neg P_6) \wedge (\neg P_3 \vee \neg P_4 \vee P_5)$$

Clearly, the evaluation of one of the propositional variables can determine in polynomial time whether the formula is true for such an evaluation. So satisfiability (SAT) is in class NP. If each clause C_i involves exactly two variables(2SAT), then it is also in class P.

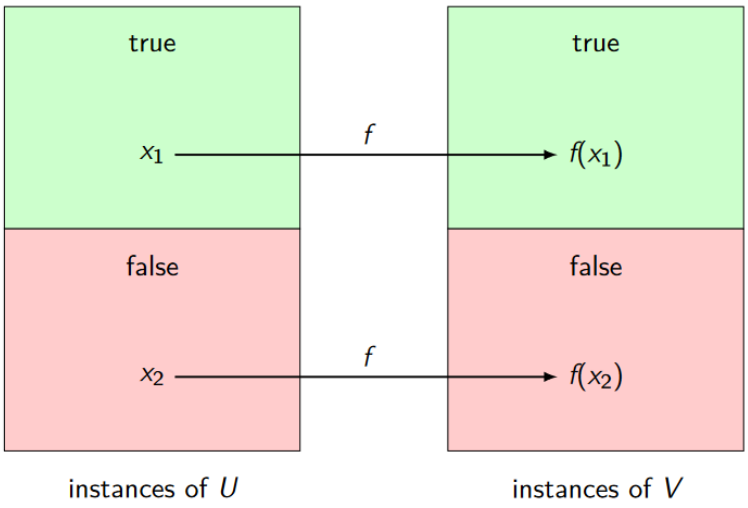
5 Reductions

Reductions gives an algorithm for the new problem, by applying the mapping function followed by the known algorithm for the old problem. The mapping doesn't necessarily mean that the problems are equivalent.

An example is instances of Balanced Partition map to instances of $0-1$ Knapsack where all items have value equal to their weight. Any algorithm solving $0-1$ Knapsack is also able to solve these special cases where $v_i = w_i$ for all i, from which we can infer an algorithm for balances partition.

Polynomials Reductions
Let U and V be two decision problems. We say that U is polynomially reducible to V if and only if there exists a function $f(x)$ such that

- $f(x)$ maps instances of U into instances of V
- f maps YES instances of U to YES instances of V and NO instances of U to NO instances of V
- $f(x)$ is computable by a polynomial time algorithm



If there is a polynomial reduction U to V, we can conclude that U is no harder than V. If you can solve provlem V in polynomial time, then problem U would also have a polynomial time solution. The contrapositive ($p \Rightarrow q$ is $\neg q \Rightarrow \neg p$) is also true: if there is no polynomial time algorithm for U, then there also can't be a polynomial time algorithm for V.

6 LIS and LCS

Longest Increasing Subsequence (LIS): Given an integer sequence A of length n, find the longest increasing subsequence of A.

Longest Common Subsequence (LCS): Given two sequences A and B of lengths m and n respectively, find the longest subsequence common to both A and B.

Suppose there already exists an algorithm that solves LCS in $T(m, n)$ time. Give a $T(n, n) + O(n \log n)$ time algorithm that solves an instance if LIS by reducing it an instance of LCS.

Algorithm
For an integer sequence A, we note that $LIS(A) = LS(A, sort(A))$. Therefore our algorithm works as follows

- We sort A in increasing order and store the result in $sort(A)$.
- Run LCS on A and $sort(A)$

Sorting the array clearly runs in $O(n \log n)$. We also see that $|A| = n$ and $|sort(A)| = n$ since $sort(A)$ is simply a permutation of A. Therefor, running LCS on A and $sort(A)$ has running time $T(n, n)$. On other works, we have a $T(n, n) + O(n \log n)$ time algorithm that solves LIS.

Reduction
To prove the reduction, we need to show that the proof holds true for both directions of it.

Let A be an instance of LIS, and let B be the longest increasing subsequence of A. Since sort(A) is a non-decreasing sequence of characters of A, it follows that any increasing subsequence must be a subsequence of $sort(A)$. Therefore, B must be a subsequence of $sort(A)$. By definition, B is also a subsequence of A; therefore, B is a common subsequence of A and $sort(A)$. By the correctness of LCS, it follows that B corresponds to the solution of LCS(A,sort(A).

For any string A of LIS, let $B = LCS(A, sort(A))$. We now argue that B is a longest increasing subsequence of A. By the correctness of LCS, it follows that B is a common subsequence of A and $sort(A)$. First, since it is a subsequence of $sort(A)$, it is an increasing subsequence. Secondly, B is also a subsequence of A. Finally, by the correctness of LCS, it follows that B must be a longest increasing subsequence of A.

7 Subset Min Cut to Min Cut

Given two subsets of vertices S, T and an integer k, define the subset minimum cut problem to be the problem of determining whether there exists a cut in G of capacity at most k that completely disconnect all vertices in S from all vertices in T; that is, no vertex in S can be connected to a vertex in T. Given an instance of the subset minimum cut problem, transform the instance into an instance of the minimum cut problem. Hence, show tha the subset minimum cut problem is also in P.

Transformation
Given the orginal graph G,

- Construct a super source s and super sink t vertex. Connect s to every vertex $u \in S$ in S with edge capacity ∞. Every vertex $v \in T$ is connected to t with edge capacity ∞.
- We keep the values of k the same.

Then $G^{'}$ in the graph of G with two extra vertices s, t and edges from s to every vertex in S, and edges from every vertex in T to t.

Reduction
We solve the minimum cut problem on $G^{'}$ to obtain a minimum cut using Edmonds-Karp. Now, we observe that a finite minimum cut must separate S and T completely.

If the minimum cut did not separate S and T, then either: the minimum cut must have separated s from some vertex in S, or the minimum cut must have separated t from some vertex in T. In either of these cases, the minimum cut must have cut through an infinite edge, implying that it would have been impossible to completely separate S and T in the first place. Thus, the minimum cut would have to be infinite.

Therefore, the edges that lie on the minimum cut must have been edges in the original graph in the first place. These correspond to the edges in G in the subset minimum cut such that S and T are completely disconnected. Clearly, the transformation is in polynomial-time since the construction of additional edges is at most $O(|V|)$. In other words, we now have a valid reduction from subset minimum cut to minimum cut.

Class P
We can transform each of the subset minimum cut instances into an instance of minimum cut. In other words, for each instance in subset minimum cut, we can make a transformation to a problem instance of minimum cut such that the instance from subset sum minimum cut is a YES instance and only if the transformed instance of minimum cut is a YES instance. Since minimum cut is polynomial-time solveable, we solve the transformed instance. This gives us the solution to the original problem. Since each operation is polynomial time, we have a polynomial-time algorithm to solve subset minimum cut, which means that the problem is in P.

8 Hamilton Path

Consider the following pair of problems.

- **HamiltonianPath:** Given a graph in $G = (V, E)$, does there exist a simple path in G that visits every vertex exactly once?
- **AlmostHamPath:** Given a graph $G = (V, E)$, does there exist a simple path in G that visits all but 2024 vertices exactly once?

Use the fact that $HamiltonianPath$ is in $NP - C$ to show that $AlmostHamPath$ is in NP.

Show AlmostHamPath is in NP

For a sequence of vertices $P = [v_1, ..., v_k]$, if P forms a simple path then that means $v_i \neq v_j$ for $i \neq j$ and for $1 \leq i \leq k-1$, we have that $(v_1, v_{i+1}) \in E$ forms an edge in G. This can easily be done in polynomial time ($O(n^2)$ for first condition, $O(n)$ for second condition). Now, to show that P misses 2024 vertices can be checked with $k = |V| - 2024$. Therefore, checking that P forms and $AlmostHamPath$ can be done in polynomial time, whoch shows that $AlmostHamPath$ is in NP.

Transformation

Let $G = (V, E)$ be an arbitrary graph. We construct $H = (V^{'}, E^{'})$ by adding 2024 isolated vertices. In other words, we construct $H = (V^{'}, E^{'})$, where

- $V^{'} = V \bigcup \{v_i : 1 \leq i \leq 2024\}$, and
- $E^{'} = E$

If G has a Hamiltonian path P, then this path must miss the 2024 vertices that we added. Therefore, P forms a simple path that visits all but 2024 vertices in H.

Reduction
Suppose that H has a simple path that visits all but 2024 vertices. Such a path must miss the vertices $v_1, ..., v_{2024}$. Therefore, the simple path must only consist of vertices in V. Moreover, the simple path must hit every vertex in V. This forms a Hamiltonian Path in G. This is a reduction that can clearly be done in polynomial time and since $HamiltonianPath$ is in $NP - C$, the reduction shows that $AlmostHamPath$ is in $NP - H$. Therefore, $AlmostHamPath$ is in both NP and $NP - H$, which means that the problem is in $NP - C$.

9 Cook's Theorem

Every NP problem is polynomially reducible to the SAT problem. This means that for every NP decision problem U there is a polynomial time computable function f such that

- for every instance x of U, $f(x)$ is a valid propositional formula for the SAT problem, and
- $U(x)$ is true if and only if the formula $f(x)$ is satisfiable

10 NP-Hard

A decision problem V is $NP-$hard if every other NP problem is polynomially reducible to V. A note is that a $NP-$hard problem may not be in class NP itself.

11 NP-Complete

A decision problem is $NP-$complete if it is class NP and class $NP - H$. SAT is in NP and from Cook's Theorem it is also $NP-$hard. So it is $NP-$complete. **Distinctions between class P and class NP-C** Problems in P:

- Given a graph G and vertices s and t, is there a simple path from s to t length at most K?
- Given a propositional formula in CNF form such that every clause has at most two propositional variables, does the formula have a satisfying assignment? (2SAT)
- Given a graph G, does G have a tour where every edge is traversed exactly once? (Euler tour)

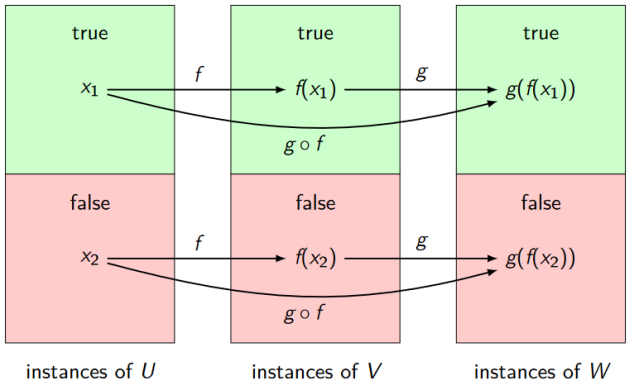
Problems in $NP - C$:

- Given a graph G and two vertices s and t, is there a simple path from s to t of length at least K?
- Given a propositional formula in CNF form such that every clause has at most three proposional variables, does the formula have a satisfying assignment? (3SAT)
- Given a graph G, does G have a tour where every vertex is visited exactly once? (Hamiltonian cycle)

Proving NP-Completeness
Let V be an $NP-$complete problem, and let W be another NP problem. If V is polynomially reducible to W, then W is also $NP-$complete.

Proof Outline

- Let $g(x)$ be a polynomial reduction of V to W, and let U be any other NP problem.
- Since V is $NP-$complete, there is a polynomial reduction $f(x)$ of U to V.
- Then $(g \circ f)(x)$ is a polynomial reduction of V to W



Proof
We first claim that $(g \circ f)(x)$ is a reduction of U to W.

- Since f is a reduction of U to V, $U(x)$ is true iff $V(f(x))$ is true.
- Since g is a reduction of V to W, $V(f(x))$ is true iff $W(g(f(x)))$ is true.

Thus $U(x)$ is true iff $W(g(f(x)))$ is true. Since $f(x)$ is the output of a polynomial time computable function, the length $|f(x)|$ of the output $f(x)$ can be at most a polynomial in $|x|$. Since $g(y)$ is polynomial time computable as well, there exists a polynomial Q such that for every input y, computation of $g(y)$ terminates after at most $Q(|y|)$ many steps. Thus, the computation of $(g \circ f)(x)$ terminates in at most:

- $P(|x|)$ many steps, for the computation of $f(x)$, plus

- $Q(|f(x)|) \leq Q(P(|x|))$ many steps, for the computation of $g(y)$

In total, the computation of $(g \circ f)(x)$ terminates in at most $P(|x|)+Q(P(|x|))$ many steps, which is polynomial in $|x|$. Therefore $(g \circ f)(x)$ is a polynomial reduction of U to W. But U could be any NP problem! We have now proven that any NP problem is polynomially reducible to the NP problem W.

12 NP-C Problems examples

The following examples are classic problems in class $NP - C$.

Travelling Salesman Problem
Suppose a map (weight directed graph) with

- vertices representing locations
- edges representing road between pairs of locations
- edge weights representing the length of these roads

and a number L. Is there a tour along the edges which visits each vertex exactly once and returns to the starting location, with total length of at most L?

Vertex Cover Problem
Given an undirected unweighted graph G and a number k, is it possible to choose k vertices so that every edge is incident to at least one of the chosen vertices?

13 Independent Sets

Consider the following pair of problems.

- **IndependentSet:** Given a graph $G = (V, E)$ and an integer k, does there exist a subset $S \subset V$ of at least k vertices such that any pair of vertices in S are not connected by a direct edge?
- **HighDegreeIndependentSet:** Given a graph $G = (V, E)$ and an integer k, does there exist a subset $S \subset V$ of at least k vertices such that S forms an independent set and each vertex in S has degree at least k in G?

To show that **HighDegreeIndependentSet** is in $NP - C$, we will use the fact that it is known that **IndependentSet** is in $NP - C$.

HighDegreeIndependentSet in NP
Given a set I of vertices, we need to check two conditions:

- I forms an independent set of size at least k; for every pair u, v of vertices in I, we check that $(u, v) \notin E$. This takes $O(n^2)$ time, using any brute force method. We can also keep track of the number of vertices in I.
- We need to check that every vertex in I has degree at least k; this can be easily checked in polynomial-time/

Therefore, checking that I forms a high-degree independent set can be done in polynomial-time, which shows that HighDegreeIndependentSet is in NP.

Transformation
Let $G = (V, E)$ be an arbitrary graph. We construct $H = (V^{'}, E^{'})$, where

- $V^{'} = V \bigcup \{u_{i,v} : 1 \leq i \leq k, v \in V\}$, and
- $E^{'} \bigcup \{()v, u_{i,v} : 1 \leq i \leq k\}$.

If G has an independent set I of size at least k, then I forms an independent set of H of size at least k. Moreover, each of the vertices in I have degree at least k.

Reduction
Suppose, now, that H has an independent set $I^{'}$ of size at least k where every vertex of $I^{'}$ has degree at least k. The vertices that we added all have degree 1, so they cannot be part of $I^{'}$. We conclude that $I^{'}$ must only

contain vertices from G and so, $I^{'}$ forms an independent set of G. In other words, G contains an independent set of size at least k.

Proof
The above reduction is clearly done in polynomial time. Since $IndSet$ is in $NP - C$, the reduction shows that $HighDegreeIndependantSet$ is in $NP - H$. Therefore, $HighDegreeIndependantSet$ is in both NP and $NP - H$, which means that the problem is in $NP - C$.

14 Salad

You want to make a salad of lettuce, tomato and onion. You know that:

- 100g of lettuce \$1.00, and gives 6 points of texture,
- 100g of tomato costs \$0.80, and gives 4 points of flavour, and
- 100g of onion costs \$0.60, and gives 1 point of flavour and 2 points of texture.

Find an amount of each vegetable so that you salad has:

- weight exactly 300g,
- flavour at least 7, and
- texture at least 7,

at the cheapest possible cost.

If we take x_L, x_T, x_O times 100g of each vegetable, we need

- weight $x_L + x_T + x_O = 3$
- flavour $4x_T + x_O \geq 7$, and
- texture $6x_L + 2x_O \geq 7/$

Subject to these restrictions, we want to minimise the cost $1x_L + 0.8x_T + 0.6x_O$. The restrictions and cost deal with only linear functions of variables. This is an example of linear programming.

15 Linear Programming

For linear programming to work, we need constraints to include the $\leq$ equality. We can convert $\geq$ inequalities to $\leq$ by multiplying the equation out by -1. The same can be done to convert minimisation problems to maximisation problems. We can also represent equalities as a pair on inequality constraints

$$x_L + x_T + x_O = 3 \rightarrow \begin{cases} x_L + x_T + x_O \leq 3 \\ -x_L - x_T - x_O \leq -3 \end{cases}$$

16 Integer Linear Programming

Suppose that in the salad problem, the vegetables come in pre-portioned 100g packets, and to avoid waste you have decided to use the entire contents of any packets that you open. Now $x_L, x_{T,O}$ are integer values rather than real-valued. We note that there is no polynomial time algorithm that solves integer linear programming.

17 Independent set as ILP

Let $G = (V, E)$ be a graph. An independent set is a subset $I \subset V$ of vertices such that any pair of vertices in I are not connected by a direct edge. Consider the following optimisation problem. What us the size of the largest independent set in G?

We realise that if we define vertices as $v_1, ..., v_n \in \{0, 1\}$, where $v_i 1 = 1 \Leftrightarrow v_i \in I$ and 0 otherwise, we can begin to form this problem as an integer linear program.

Optimisation Functions

$$f(v) = \sum_{i=1}^{n} v_i$$

where $v = (v_1, ..., v_n) \in \{0,1\}^n$.

Constraints
Our constraints should ensure that for each edge $e \in E$, that both endpoints are not selected. Hence, for each $(v_i, v_j) \in E$, we should have $v_i + v_j \leq 1$.

Linear program

$$\begin{gathered}\text{maximise } f(v) = \sum_{i=1}^{n} v_i \\ \text{subject to } v_1 + v_j \leq 1, v_i, v_j : (v_i, v_j) \in E \\ v_i \in \{0,1\}, i = 1, .., n\end{gathered}$$

18 Maximum flow as a Linear Program

Given a flow network $G(V, E)$, we can model the max flow as a linear program.

Variables

Define variables $x_{uv}, u \neq v$ as the amount of flow flowing through each directed edge $(u, v) \in E$. Then we need to ensure flow conservation is respected for $v_i \in V \setminus \{s, t\}$ we must have

$$\sum_{v:(u,v_i)\in E} x_{uv_i} = \sum_{w:(v_i,w)\in E} x_{v_i w}$$

Objective function
Then we need to maximise flow out of the source or into the sink

$$f(x) = \sum_{v:(s,v)\in E} x_{sv} \quad \text{or} \quad f(x) = \sum_{v:(v,t)\in E} x_{vt}$$

Solution

$$\text{maximise f(x)} = \sum_{v:(s,v)\in E} x_{sv}$$

subject to

$$\begin{gathered} x_{uv} \leq c(u,v) \quad (u,v) \in E \\ \sum_{v:(u,v_i)\in E} x_{uv_i} = \sum_{w:(v_i,w)\in E} x_{v_i w} \quad u_i \in V \setminus \{s,t\} \end{gathered}$$

19 Intractable optimisation problems

Optimisation problems are interested in maximizing or minimizing some particular quantity. Consider a modification to the travelling salesman problem.

Optimised Travelling Salesman Problem
Suppose a map (weight directed graph) with

- vertices representing locations
- edges representing road between pairs of locations
- edge weights representing the length of these roads

Find a tour along the edges which visits each location exactly once and returns to the starting location, with minimal total length

This version of TSP is clearly no easier than the corresponding decision problem. If we could solve the optimisation version, we could then solve the decision version in constant extra time by comparing the length of the shortest path of the length L being tested. We don't know any polynomial time algorithm for the decision variant of TSP and certainly dont know a polynomial time algorithm for optimisation TSP. We have an intractable problem on our hands.

Making progress on intractable optimisation problem
If a problem is an optimisation problem, we can try to:

- Solve it exactly with a slower-than-polynomial time algorithm, or
- Solve it approximately with an efficient algorithm

Thus, for a practical problem which appears to be intractable, the strategy would be:

- confirm that the problem is indeed intractable, and
- look for an exact algorithm which provides an answer in slower than polynomial time, or
- look for an approximation algorithm which provide an answer that is always within acceptable factor of the optimum.

20 Superpolynomial algorithms

Even if a problem is intractable, we can still use our techniques for algorithm design. There are often degrees of intractability of a problem. A brute force approach much run unreasonably long time for all but smaller instance whereas a faster algorithm might still be viable for moderately small instances. So even if we can not find a polynomial time algorithm, there are somtimes significant benefits to:

- reducing the index of an exponential term, or
- changing from factorial time to exponential time

21 Subset Sum

Given an array of n positive integer, and a positive integer S, find the maximum sum of a subset with sum $\leq S$.

A brute force approach would consider every possible subset. There are two choice for each number (include it or exclude it in the subset), for a total of 2^n subset. We then have to add up each subset, for a total time complexity of $O(2^n)$. We can do better by adding some redundancy.

Each set of choices from the first half of the array is represented in $2^{\frac{n}{2}}$ different subsets. After a small change in the first half, we will again go through $2^{\frac{n}{2}}$ corresponding subsets for the various selections from the second half. We can instead try adding up every subset of the first half, and look for a pair which matches.

Algorithm

- Compute all the sums of subsets of the first half and the second half of the array by brute force, and store them in arrays A and B.
- Sort array B using mergesort.
- For every entry of array A, say $A[i]$, find the lower bound for $S - A[i]$ in array B, using binary search.
- Update the tentative answer to $A[i] + B[j]$

Time Complexity
Computing arrays A and B takes $O(2^{\frac{n}{2}}n)$. Array B has $2^{\frac{n}{2}}$ entries, so mergesort runs in $O(2^{\frac{n}{2}}n)$. For each of $2^{\frac{n}{2}}$ entries of A, we run a binary search in $O(\log 2^{\frac{n}{2}}) = O(n)$ time. Therefore the total time complexity is $O(2^{\frac{n}{2}})$. We still have an exponential time complexity but about a square root quicker than the previous. This can have massive implications for large inputs.

22 Travelling Salesman Problem

We now consider the optimisation TSP once again. A brute force approach considers all orders of vertices in which we can visit. This is an $O(n!n)$ time complexity. We can do better using shortest path dynamic programming however. Consider subproblems as shortest partial paths in which the the vertex v_k that we're considering isn't one we've seen already. Perhaps we need to only know the vertices we've visited, not the order we've visited them in.

Subproblems.
For $S \subseteq V$ containing s, and $v \in S$, let $P(S, v)$ be the problem of determining $opt(S, v)$, the length of the shortest path starting at s, visiting the vertices of S once each and finishing at vertex v.

Recurrence.
For $v \neq s$, we have

$$opt(S, v) = min_{u \in S \wedge (u,v) \in E}[opt(S \setminus \{v\}, u) + w(u, v)]$$

Base case
$opt(S, s) = \begin{cases} 0 \text{ if} S = \{s\} \\ \infty \text{ otherwise} \end{cases}$

Order of computation
As $P(S, v)$ depends only on $P(S^{'}, u)$ where $|S^{'}| = |S| - 1$, we can solve the subproblems in increasing order of $|S|$ to ensure all dependencies are respected. The overall answer considers all choices of penultimate vertices plus the edge that completes the tour; $min_{v \neq s}[opt(V, v) + w(v, s)]$

Time complexity
There are $2^n n$ subproblems each taking $O(n)$ time; $O(2^n n^2)$. This is exponential rather than the brute force factorial time.

23 Approximation Algorithms

As seen earlier, intractable problems are often too slow if they are to be optimised. We can however approximate them in much quicker time.

Minimum Vertex Cover
Consider an undirected unweighted graph $G = (V, E)$. Find the smallest number of vertices so that every edge is incident to at least one of the chosen vertices.

Algorithm

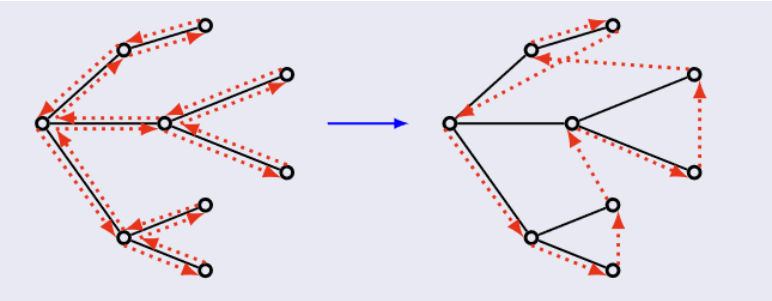
- Pick an arbitrary edge and cover both its ends.
- Remove all the edges whose one end is now covered. In this way you are left only with edges which have either both ends covered or no end covered.
- Continue picking edges with both ends uncovered until no edges are left

This produces a vertex cover since edges are removed if one of their ends is uncovered. We perform this procedure until no edges are left. The number of vertices covered is equal to twice the number of edges with both ends covered. But the minimal vertex cover must cover at least one vertex of each edge. Thus we have produced a vertex cover (an approximation) of size at most twice the size of the minimal vertex cover.

Metric Travelling Salesman Problem
Consider a complete weighted graph G with weights $d(i, j)$ of edges satisfying the "triangle inequality": for any three vertices i, j, k we have

$$d(i, j) + d(j, k) \geq d(i, k)$$

Algorithm
Find the minimum spanning tree T of G. Since the optimal tour with one of its edges e removed represents a spanning tree, we have that the total weight of T satisfies $w(T) \leq opt - w(e) \leq opt$. If we do a depth first traversal of the tree, we will travel a total distance of $2w(T) \leq 2opt$. We now take shortcuts to avoid visiting vertices more than once; because of the triangle inequality, this operation does not increase the length of the tour.



This algorithm produces an approximation by producing a tour of total length at most twice the length of the optimal length tour (this is denoted as *opt* above).

The take away
All $NP-C$ problems are equally difficult, because any of them can be polynomially reducible to any other. The related optimisation problems, as seen above, can be very different! The most general TSP problem does not have an approximate solution if $P \neq NP$. If $P = NP$ however, there does exist an algorithm that approximate the TSP problem.

Algorithm

- Let G be an arbitrary unweighted graph with n vertices
- We turn this graph into a complete weighted graph G^* by setting the weights of all existing edges to 1, and then adding edges of weight $K \cdot n$ between the remaining pairs of vertices.
- If an approximation algorithm for TSP exists, it produces a tour of all vertices with total length at most $K \cdot opt$, where *opt* is the length if the optimal tour through G^*

If the original graph G has a Hamiltonian cycle, then G^* has a tour consisting of edges already in G and of weight equal to 1, so such a tour has length of exactly n, Otherwise, if G does not have a Hamiltonian cycle, then the optimal tour through G^* must contain at least one added edge of length $K \cdot n$, so

$$opt \geq (K \cdot n) + (n-1) \cdot 1 > K \cdot n$$

Thus, our approximation TSP algorithm either returns:

- a tour of length at most $K \cdot n$, indicating that G has a Hamiltonian cycle, or
- a tour of length greater than $K \cdot n$, indicating that G does not have a Hamiltonian cycle.

If this approximation algorithm runs in polynomial time ($P = NP$), we now have a polynomial time decision procedure for determining whether G has a Hamiltonian cycle!