1 Exact exponential algorithms

1.1 Dynamic programming for TSP

1.1.1 The dynamic program

In the traveling salesman problem, we wish for the salesmand to do a full tour of the graph $G = (V, E)$ visiting each city/vertex $v \in V$ at a minimal cost. The cost of going from city c_i to c_j is given by the function $d(c_i, c_j)$. The task of constructing the tour can be seen as a permutation π of the n cities to visit, and we then wish to minimize the sum

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

that is find the "best" permutation of vertices. The brute-force way would be to just consider all $n!$ ways to visit each city, and this requires at least $n!$ steps.

The dynamic programming algorithm considers a pair (S, c_i). Here we have that S is a nonempty subset $S \subseteq \{c_2, c_3, ..., c_n\}$, and we have that $c_i \in S$. We then define the value $OPT[S, c_i]$ to be the minimum/optimal length tour starting at city c_1, visiting all cities in S and ending in city c_i. When $|S| = 1$ then it is trivial to compute $OPT[S, c_i]$, since we then know that $S = \{c_i\}$, so we get that $OPT[S, c_i] = d(c_1, c_i)$. The more interesting case in then when we get that the set has size $|S| > 1$. Then the value of $OPT[S, c_i]$ can be expressed at the recursion

$$OPT[S, c_i] = \min \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i) : c_j \in S \setminus \{c_i\}\}$$

which is the optimal way to visit all cities in S and end in c_i is to visit each vertex in $S \setminus \{c_i\}$ and end c_j and then go to vertex c_i for all $c_j \in S \setminus \{c_i\}$. This can be repeated recursively until the base case.. Stated in other words, if we have an optimal tour in S with city $c_j$preceeding c_i in the tour, then we have that

$$OPT[S, c_i] = OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)$$

the optimal tour is the tour in S without c_i ending in c_j and then taking the edge (c_j, c_i) (since if we take any other edge to get to c_i it will be at least as long). We however, need to find the minimum edge c_j that gets us to c_i so we take the minimum around it (if we took a random edge it would not be a minimum and the TSP has that the graph is fully connected, so there is always an edge from $c_j \to c_i$). The final optimal value for ending in city c_i will be $OPT[\{c_2, c_3, ..., c_n\}, c_i] + d(c_i, c_1)$.

1.1.2 Running time

For a fixed size set $|S| = k$ and all vertices $c_i \in S$, then the time taken for a single subset is $O(k^2)$, since there are k vertices, and for each of them we

calculate the cost of getting to each of the other vertices (this is just a lookup in OPT). There are however $\binom{n}{k}$ of such subsets of size k.

The algorithm wants to consider subsets of size $2, 3, ..., n-1$ since the case of 1 is trivial as explained earlier, and $n-1$ since we from the optimal tour in the subset of size $n-1$ take the minimum edge to the last vertex. There are $\binom{n}{k}$ of such subsets of size k. Furthermore, when we have a given set of size k, we make $k-1$ lookups into OPT to calculate the optimal path to the other vertices in the subset for each k vertex. This gives us the sum

$$\begin{aligned}
\sum_{k=2}^{n-1}\binom{n-1}{k}\sum_{j=1}^{k}(j-1) &\leq \sum_{k=2}^{n-1}\binom{n-1}{k}\sum_{j=1}^{k}n \\
&\leq \sum_{k=2}^{n-1}\binom{n-1}{k}\sum_{j=1}^{n}n \\
&= \sum_{k=2}^{n-1}\binom{n-1}{k}n^2 \\
&= n^2 \cdot \sum_{k=2}^{n-1}\binom{n-1}{k} \\
&\leq n^2 \cdot \sum_{k=1}^{n}\binom{n}{k} \\
&= O(n^2 \cdot 2^n)
\end{aligned}$$

this is much better than the brute force $O(n!)$ approach.

Why does dynamic programming work here?

When we for a subset of size k have an optimal tour $OPT[\{c_{\pi(2)}, ..., c_{\pi(k-1)}\}, c_{\pi(k)}]$ then we will have that the optimal tour for subsets of size $k+1$ is

$$OPT[\{c_{\pi(2)}, ..., c_{\pi(k)}\}, c_{\pi(k+1)}] = OPT[\{c_{\pi(2)}, ..., c_{\pi(k-1)}\}, c_{\pi(k)}] + d(c_{\pi(k)}, c_{\pi(k+1)})$$

So we have optimal substructure, and the solution to the smaller problems are part of the solution to the larger problem. The only drawback is that we have to allocate space for $OPT[S, c_i]$ for all sets and thereby the space is $\Omega(2^n)$

1.2 Independence set

A maximal independence is the largest set of vertices $I \subseteq V$ such that each pair of vertices in I are non-adjacent. This is the opposite of a clique. The naive approach is to check every subset if which takes $\Omega(2^n)$. The algorithm works under the following observartion:

If a vertex v is in a independent set I then none if its neighbors can by definition be I. If we assume that I is a maximum independent set, and v is not in I then a neighbor of v must be in I. This can be seen by the fact that if

a neighbor of v is not in I then we can extend $I = I \cup \{v\}$ and we would get an even larger independence set, ruining our assumption that I was a maximum independent set. This means, that for every independence set I and every vertex v, there must be a vertex y in the closed neighborhood $N[v]$(vertices adjacent to v and v itself), such that $y$is in I and no other $N[y]$ is in I. We use this observation to solve the problem on reduced instances and pick the best amongst the reduced instances.

The algorithm thus works by picking a vertex v with least cardinality(least incident edges). Then it considers the subproblem of finding a maximum independence set of for each subset $G \setminus N[y] : y \in N[v]$, and picks the best solution amongst these. The branching happens after we pick v - we can find a solution for $G \setminus N[v]$ and for $G \setminus N[v']$ where v' is a adjacent vertex.

Running time:

Each time we branch in the algorithm, we reduce the problem to two subproblems proportional to $G \setminus N[v]$ and $G \setminus N[v']$. We can see the algorithm as a search tree T where each node corresponds to the subproblem of finding the maximum independent set. The root of the tree is the problem of finding independence set in G and the one child node will then be the subproblem of finding the MIS in $G \setminus N[v]$ and this keeps recursing. We get the solution to a subproblem in a node by solving the subproblems of the child nodes. Therefore, we can analyze the running time be looking at the size of the "search tree" T and the time spent in each node. We will therefore look at the size of the search tree $T(n)$ when give a graph G of n vertices.

Let say the algorithm picks a vertex v with minimum degree $d(v)$ and we have input graph G. Then to solve the subproblem of finding independent set of G boils down to solving it for the subgraphs $G \setminus N[v], G \setminus N[v1], ..., G \setminus N[v_{d(v)}]$. The reason for this is in part since $v, v_1, v_2, ..., v_{d(v)}$ is the closed neighborhood of v. Furthermore, due to the observation of maximum independence set, then if v is in I, then none of its neighbors can be in it. Furthermore if a neighbor v_i is in I , then none of its neighbors $N[v_i]$ can be in I. Therefore we have to solve $G \setminus N[v], G \setminus N[v1], ..., G \setminus N[v_{d(v)}]$ and pick the best solution among them. This gives us the total work performed is

$$T(n) \leq 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v_i) - 1)$$

we add 1 for the addition. We get the $T(n - d(v) - 1)$ since, there are now $d(v) - 1$ vertices less (neighbors plus the node itself). We always choose a vertex of least degree in the algorithm, then the degree of v must be less than its neighbors

$$\begin{aligned} & d(v) \leq d(v_i) \\ & \implies n - d(v_i) - 1 \leq n - d(v) - 1 \text{ subtract less} \end{aligned}$$

since $T(n)$ is monotonically increasing

$$T(n - d(v_i) - 1) \leq T(n - d(v) - 1)$$

It can be assumed that $T(0) = 1$, since it takes constant time to see there is no tree. We thus get

$$\begin{aligned} T(n) \leq & 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v) - 1) \\ \leq & 1 + (d(v) + 1) \cdot T(n - d(v) - 1) \end{aligned}$$

now the authors make a mistake by setting $s = d(v) + 1$***, since the degree depends on*** v ***which is different at each level of the recursion (the minimum degree edge will be different at each recursive call)***

$$T(n) \leq 1 + s \cdot T(n - s) \leq 1 + s + s^2 + ... + s^{n/s} = \frac{1 - s^{n/s+1}}{1 - s} = O^*(s^{n/s})$$

The for $s > 0$ the function $f(s) = s^{1/s}$ has maximum value for some $s = 3$. Thereby we get that

$$T(n) \leq O^*(s^{n/s})$$

2 Parameterized algorithms

2.1 Bar fight prevention problem

The premise of the problem is a bouncer that wants to avoid fights. It assumed he/she knows all the attendants and which of them will fight. The bouncer would likek to rejct all the guests that will cause a fight, but management will only let the bouncer reject k guests. We have an optimization problem of letting as many people in as possible without rejecting more than $k$guests. This is known as the vertex cover problem where we would like to cover a graph of size k, and this is NPC. If we can cover the graph we can never solve the bar fight prevention problem.

2.1.1 Brute force algorithm

For $n = 1000$ we can always try to brute force it with 2^{1000} guesses and pick a permutation of guests that causes no guests, but this is so slow (10^{301}) it will never terminate. We are however informed that we can bounce $k = 10$ guests, so instead we need to try the $\binom{1000}{10}$ permutations of bouncing people and pick one if any set of the k people to reject that does not cause a fight. This is still very large of around 10^{23} possibilities.

2.1.2 Using kernelization

1. Anyone with no conflicts (equivalent to $d(v) = 0$ in vertex cover problem) can safely bet let in, since they will have no conflicts

2. Reject anyone with $k+1$ conflicts, and decrease k by one. The only way to prevent fight with this one would be to reject $k+1$ more guests, and this is not allowed. This is equivalent to having to cover $k+1$ more vertices if selecting this one

3. Note: If there are no more people with the above two characteristics, then we know that each guest has at most k potential conflicts ($d(v) \leq k, \forall v \in V$). Note, there can now be around $O(n \cdot k)$ fights and this can be larger than k^2. The problem is that if we have more than k^2fights at this points, there is no way to prevent fights. The reason is that solving a potential conflict solves at most k fights, and if we do this k times, we resolve k^2fights. However we now there are more than k^2 and fights and we can never solve it. This is equivalent to having $|E| > k^2$

4. Now assuming we have finished step 1 and 2, we know that each guest has at most k and at least 1 fight (we have accepted the ones with 0 fights and rejected those with more than $k+1$). If we look at this as a graph with $|V|$ guests with undecided fate, and edges representing conflicts then we know
$$|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$$
The first inequality holds since each gust has at least one conflict. The second equality holds since when we count each edge twice when calculating the degree over each edge (every edge is incident on two vertices and gets counted twice). The second inequality holds since we know from step 3 that there would be no solution if $|E| > k^2$ and we would already have terminated. Thereby we can now check the $\binom{2k^2}{k}$ choices to see if any results in fights.

5. For the guests with only 1 conflict we can also safely accept them. Since all guests in the "conflict" graph have a degree of at least 1 (one possible fight), then accepting those with exactly 1 conflict cannot be worse than accepting anyone else. Assume Alice has a single conflict with Bob. Accepting Bob would resolve in rejecting Alice, and possibly more guests (We do not know anything about the number of conflicts Bob has) so in this way it is no worse to accept Alice than anyone else. Therefore, accept those with 1 conflict and decrease k by 1. Now all the remaining have at least 2 and at most k fights. This gives us
$$|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 \leq \frac{1}{2} \sum_{v \in V} d(v) = \frac{1}{2} \cdot 2|E| \leq k^2$$

where the first inequality holds since each guest has at least 2 fights. The third inequality holds again since each edge is counted twice in the degree. This gives us time $\binom{k^2}{k} = 1.73 \cdot 10^{13}$ and can easily be computed.

Using reduction rules to reduce the problem instance is known as kernelization.

2.1.3 Bounded search tree

We only have success if we resolve every conflict be rejecting at most k people. Furthermore, a conflict between two guests is resolved by rejecting one of them. This leads us to the following algorithm. Say there is a conflict between Alice and Bob. We first reject Bob, check if the remaining conflicts can be resolved by recursively rejecting at most $k-1$ guests. In case is fails, we try to reject Alice, and recursively check if we can resolve all conflicts by rejecting at most $k-1$ additional guests. If this fails, we can never solve all conflicts.

The running time of this algorithm can be analyzed in terms of the number of recursive calls. The idea of the algorithm was to pick an arbitrary conflict - an edge, (u, v), in the graph over potential conflicts between two guests. Then we try to reject u decrease k by 1, and see if we can solve all conflicts by rejecting additional $k-1$ guests. The same is done by rejecting v and see if we have success by rejecting an additional $k-1$ guests. Each recursive call does this again, and we thus try to solve 2^kdifferent subproblems (2^kdifferent combinations of rejecting k guests). Each path from root to leaf of the search tree is considered the set of rejected guests and is thereby a subproblem and the number of leaves in balanced binary search tree is 2^k - twice as many nodes as previously and we stop at depth k.

We can assume we have already done some kernelization so that the guests have at most $k$conflicts (step 1 and 2 of kernelization). The number of edges in the conflict graph is

$$2|E| = \sum_{v \in V} d(v) \leq n \cdot k$$

where the inequality holds since each remaining guest has at most $k$conflicts. This gives us $|E| \leq \frac{1}{2} n \cdot k$. So we approximately $\frac{1}{2} n \cdot k$ conflicts before we try the bounded search tree approach. So for each subproblem we have to check if removing those k guests solves the $O(nk)$ conflicts. This gives a running time of $O(nk \cdot 2^k)$.

If we consider k to be a constant, then this is actually a polynomial time algorithm. The algorithm is only exponential in k but linear in n. This algorithm is considers a fixed parameter algorithm FPT: $f(k) \cdot n^c$ for a constant c that is independent on n and k. For this algorithm we hade $f(k) = 2^k$ and $n^c = n^1$.

2.1.4 FPT and XP

The problem of vertex cover is NP-complete for finding a minimal value of k. But when we fixed k then we could actually construct two linear time algorithms for the problems. One using kernelization and the other a bounded search tree.

A slice-wice polynomail algorithm (XP) is if it has running time $f(k) \cdot n^{g(k)}$.