Fixed-Parameter Algorithms

Often an NP-hard optimization problem has many parameters besides the input size. For example, the Knapsack problem has a special parameter $B = \lceil \log p_{max} \rceil$, the number of bits used to encode the maximum profit of an item. This parameter does not necessarily directly relate to the input size. One can have an input size of n objects, but B can be bounded by a constant, or $O(\log n)$, or even $O(n)$. Recall that we had this algorithm that takes $O(n^2 \cdot p_{max}) = O(n^2 \cdot 2^B)$, which is polynomial if we **fix** B. So, the Knapsack problem is called Fixed Parameter Tractable (or FPT) with respect to parameter B.

In general, a problem is FPT with parameter k, if it has an algorithm with time complexity $O(n^c \cdot f(k))$ for a constant c, and **any** function $f(k)$.

Notice that by "fixing" a parameter, we did not mean that the parameter must be a constant. We only mean that if we ignoring the $f(k)$ factor in the time complexity, then the time complexity is polynomial. The hope is that if the parameter k is selected carefully, $f(k)$ may be relatively small, even if $f(\cdot)$ is a super-polynomial function.

Many NP-hard problems can be solved this way. We learn a few examples in this chapter of the course.

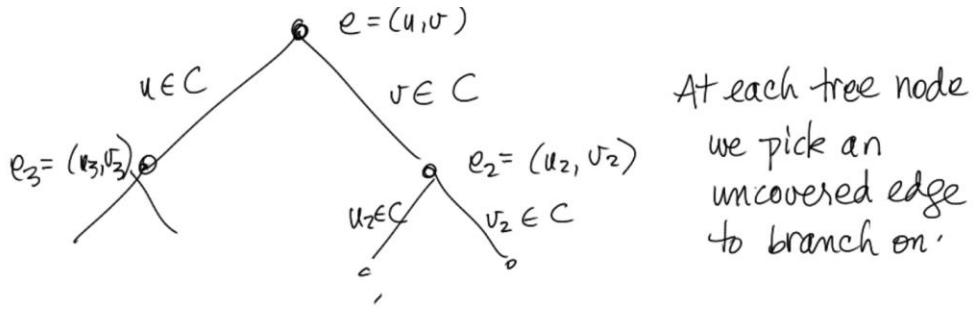
Vertex Cover

Recall that in vertex cover, we want to find the minimum sized subset of vertices such that each edge is covered. Let k be the optimal size of the cover. k can be regarded as a parameter of the problem. For easy presentation, we consider the decision version: Whether a graph G has a vertex cover of size $\leq k$. But the algorithms can be easily modified to find the solution if the answer to the decision version is yes.

If k is a constant, a straightforward way to solve this problem is to try the $\binom{n}{k} = O(n^k)$ possible subsets of vertices, and then check which one is a cover. This only takes polynomial time if k is a constant. But n^k is still bad. Also, it is not regarded as a FPT algorithm for parameter k, because the super polynomial part is not a function of the parameter k, it is a function of both k and n.

Bounded Search Tree

Consider the following fact. Let V' be one of the optimal vertex cover. Then for each edge $e = (u, v)$, either u or v appears in V'. So, we can perform the following search:



Algorithm VC(G, k)

1. If $E = \emptyset$ return YES.
2. Else if $k = 0$ return NO.
3. Pick an arbitrary edge $e = (u, v)$,
4. Return VC($G - u, k - 1$) OR VC($G - v, k - 1$).

Here OR is the logical or; and G-u means the subgraph obtained by removing u and all incident edges from G.

Lemma. Algorithm VC correctly computes the answer.

Proof: If VC(G,k) has a YES answer, since one of (u,v) needs to be in the cover, one of VC(G-u, k-1) and VC(G-v, k-1) should have a YES answer. Vice versa. QED.

Theorem. Algorithm VC correctly computes the answer in time $O(2^k \cdot n)$.

Proof: The search tree has a depth of k, and each branching node of the search tree has two children. Each branching needs $O(n)$ to compute the subgraph.

QED.

Notice that $O(2^k \cdot n)$ is much better than $O(n^k)$. Importantly, 2^k is now a function of k. So, this is a FPT algorithm for parameter k.

Remark: One can easily modify the algorithm to find the actual minimum vertex cover, instead of returning a yes or no answer.

This technique is called the "bounded search tree". The idea is to bound the search tree's degree of branching, as well as the depth, by the parameter k. In the above example, in stead of picking any vertex as in the $O(n^k)$ algorithm, we use an uncovered edge (u, v) to help us bound the branching cases to be between u and v. Also, we bound the depth of the search tree by k.

Kernerlization

There is another technique called "kernelization" that can be used to further reduce the time complexity. Kernelization means to find the most difficult part (the kernel) of the instance, usually by removing the trivial cases.

Now consider VC again. If a vertex v has degree $> k$, then v has to be included in the cover. Otherwise, all the other vertices adjacent to v need to be in the cover, exceeding the parameter k. So, given instance (G, k), if we find a degree $> k$ vertex, we can remove it from the graph to reduce the problem to a smaller instance $(G', k-1)$.

Similarly, if there is any vertex that is isolated, then it can be removed too.

By repeating this procedure at most k times, in $O(m+n)$ time, we get a kernel (G', k'), where each vertex in G' has degree $\leq k'$.

What is the size of G'? If the original instance (G, k) has an YES answer, then the size is $O((k')^2)$. This is because it has a vertex cover of size $\leq k'$, and each vertex has degree $\leq k'$.

Then we can use the former Algorithm VC to solve the kernel in time $O\left(2^{k'} \cdot (k')^2\right) = O\left(2^k \cdot k\right)$. The total time is therefore $O\left(2^k \cdot k + m + n\right)$.

Algorithm VC-Kernel

1. Use the above procedure to find a kernel (G', k').
2. If G' has more than $(k')^2$ vertices, return NO.
3. Else return VC(G', k').

Branch-and-Bound

The complexity $O\left(2^k \cdot k + m + n\right)$ is practical for small k such as 30. With some heuristics, such as branch-and-bound (B&B), you may be able to do much better than 30. The idea is given below.

Consider the bounded search tree for VC. If the previous t choice from the root to the current node is so bad, that the resulting subgraph's minimum VC has a size $> k - t$. Then we do not need to continue the search beyond this point. But how do we obtain the lower bound for the VC size? We can:

1. Maximal matching is a lower bound.
2. Maximum matching is a lower bound.
3. Solution of the LP relaxation is a lower bound.
4. Any approximation algorithm with proven ratio can provide a lower bound.
5. Kernelization on the subgraph.

These lower bounds are not tight (except for Kernelization). But in many cases, they are good enough to exclude a subtree of the search tree, and therefore speed up significantly.

Remark: B&B doesn't guarantee an asymptotically better time, but works well in practice. It is a widely used heuristics in many circumstances.

Remark: The best FPT algorithm with provable time complexity for VC has time complexity $O\left(1.2738^k \cdot n^{O(1)}\right)$.

Closest String

Recall that the closest string asks for a center string t that is the closest to n input length-L strings $s_1, \ldots, s_n$. We consider the radius d as given. (Otherwise, we can simply search for d). In bioinformatics, this d is usually required to be small. An instance with large d is not practically useful. So, it's nice to have a FPT algorithm with d as a parameter.

General alphabet size

The bounded search tree method can be used again. We start the search by assigning $x = s_1$. We know that the center string t we are looking for is such that $d(t, x) \leq d$. We want to modify x one letter at a time, until it becomes t. So the search tree depth will be bounded by d.

A straightforward way is to try each of the $O(L)$ positions for modification. This guarantees the success, and has time complexity $O\left(L^d \cdot n\right)$. This is not good enough as L is part of the exponential function.

Examine $d(x, s_i)$ for every i. If $d(x, s_i) \leq d$, then x is already a center, and we do not need to search further. If not, suppose $d(x, s_i) > d$ for a certain i.

Claim 1: If $d(x, s_i) > d$, let $P = \{j \mid x[j] \neq s_i[j]\}$. Then there is $j \in P$ such that $t[j] = s_i[j]$.

Proof: This is because $|P| > d$, t and s_i cannot disagree at all of them because $d(t, s_i) \leq d$. QED.

So, if we try each position in P, and try to flip $x[j]$ to $s_i[j]$ at that position, for at last once, we make x closer to t.

Algorithm ClosestString

1. $x \leftarrow s_1$
2. for Δ from d to 1
3. If $d(x, s_i) \leq d$ for every i
4. Output x and halt.
5. Else
6. Let i be such that $d(x, s_i) > d$.
7. Guess $j \in \{j' | x[j] \neq s_i[j]\}$.
8. $x[j] \leftarrow s_i[j]$.
9. Return FAIL.

Notice that we use "Guess" in line 7 to indicate that the algorithm needs to branch the search for each possible j. It really means "For each $j \in \{j' | x[j] \neq s_i[j]\}$, do the following and then recursion".

The branch degree at line 7 is $|P| \leq 2d$. The bounded search tree height is d. Checking lines 3 and 6 take $O(nL)$. So, the time complexity is $O\big((2d)^d \cdot nL\big)$.

The following theorem follows the above discussion.

Theorem: Algorithm ClosestString computes the closest string in time $O\big((2d)^d \cdot nL\big)$.

So, this is a FPT algorithm with respect to parameter d.

Notice that in Claim 1, if we replace P with any size-$(d+1)$ subset of $\{j \mid x[j] \neq s_i[j]\}$, the Claim still holds. Thus, we can improve the algorithm by replacing line 7 with

7. Arbitrarily take a size $(d+1)$ subset $P \subseteq \{j' | x[j] \neq s_i[j]\}$. Guess $j \in P$.

This will reduce the branching degree to $d+1$. So the time complexity becomes $O\big((d+1)^d \cdot nL\big)$.

Fixed alphabet size

When the alphabet size is fixed (regarded as a parameter), one can do better. Here let us limit our discussion on binary strings. But algorithms in this section can be extended so they work for fixed alphabet size.

We still do something similar to the bounded search tree algorithm in last section. We start with $x = s_1$ and tries to modify x one letter at a time to get the center string t. Suppose initially $d(x,t) = \Delta \leq d$. Each correct flip will reduce the value of Δ by one.

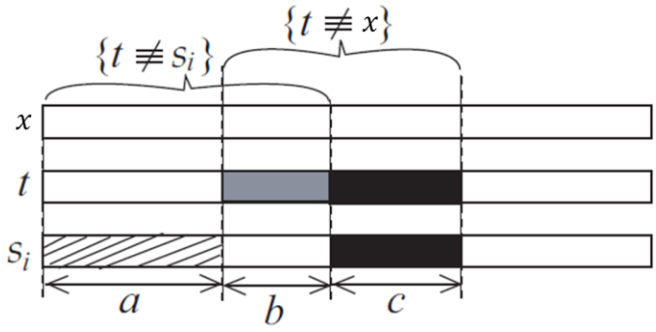
For each step, we find s_i such that $d(x,s_i) = d + l \geq d$. Let $P = \{j \mid x[j] \neq s_i[j]\}$. The previous algorithm needs to try every j in P in order to ensure that at least once we flip it correctly. Now let me try something really simple: I'll randomly select a bit in P and flip it.

Algorithm RandomFlip

1. $x \leftarrow s_1$
2. For k from 1 to d
3. If $d(x,s_i) \leq d$ for every i, then output x and halt.
4. Else
5. Find s_i such that $d(x,s_i) > d$.
6. Randomly pick a position j where $x[j] \neq s_i[j]$, and flip it.

Theorem: Algorithm RandomFlip finds the center string with probability $\geq (2e)^{-d}$.

Proof:



Consider the relation of the three strings t, x and s_i. Since this is binary string, their relation is shown in the above figure. In the figure, two strings are different at a position if they are labeled with different shadow. Clearly,

$$d(x,t) = \Delta = b + c$$

$$d(x,s_i) = d + l = a + c$$

and

$$d(t,s_i) = a + b \leq d$$

Add the three (in)equalities together, we get

$$\Delta + (d + l) + (a + b) \leq (b + c) + (a + c) + d$$

Or equivalently,

$$c \geq \frac{\Delta + l}{2}$$

Note that c is the number of bits where we need to flip x in order to get s_i. Also, notice that all the c bits belong to $P = \{j \mid x[j] \neq s_i[j]\}$. Thus, if we randomly flip a bit of x in P, the probability that we flip the right bit is at least

$$\frac{c}{|P|} = \frac{c}{d + l} \geq \frac{1}{2} \times \frac{\Delta + l}{(d + l)} \geq \frac{\Delta}{2d}$$

Thus, the probability that all of the d flips are correct is

$$\prod_{\Delta=1}^{d'} \frac{\Delta}{2d} \geq \prod_{\Delta=1}^{d} \frac{\Delta}{2d} = \frac{d!}{(2d)^d}$$

Here $d' = d(s_1, t)$. By Stirling's Formula, $d! \geq \left(\frac{d}{e}\right)^d$. QED.

With the theorem, we only need to repeat the algorithm $O\left((2e)^d\right)$ times and take the best solution found, in order to increase the probability close to 1. Thus, we have the following corollary:

Corollary: There is a randomized algorithm that computes the optimal solution of closest string with time $O\left((2e)^d \cdot (nL)^{O(1)}\right)$ with high probability.

K-Path

Given a graph G, a pair of vertices s and t, find a simple path with k internal vertices that connects s and t. A simple path is a path that does not visit a vertex twice.

K-path is NP-hard because Hamilton Path is a special case of k-path where $k = n - 2$. Now we want to design an FPT algorithm using k as a parameter.

Idea: Randomly color the vertices with k different colors; Find a simple path that uses all colors.

We call such a simple path as a colorful path.

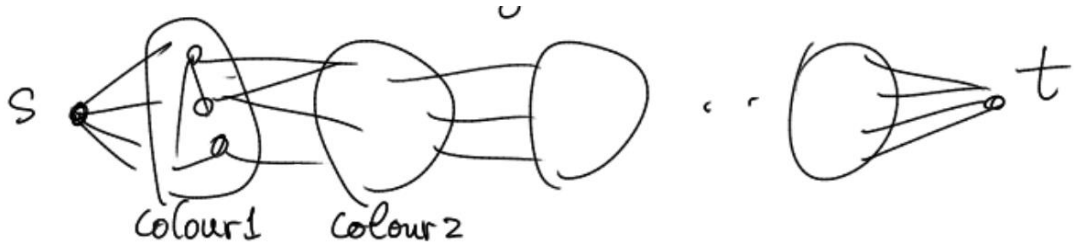
Two things need to be addressed:

1. If there is a k-path, what's the probability it is colored as a colorful path?
2. How to find a colorful simple path?

Lemma: If there is a k-path, the random coloring makes it colorful with probability at least e^{-k}.

Proof: The probability is $\frac{k!}{k^k} \geq \frac{\left(\frac{k}{e}\right)^k}{k^k} = e^{-k}$. QED.

Now let's worry about how to find a colorful simple path after the coloring. Try all $k!$ permutations of the colors. One of the permutations will be such that there is a path that goes through the colors in order.



Computing a path given this order is easy in $O(m)$ time where m is the number of edges. (How?)

So, the total running time is $O(m \cdot k!)$ with success probability e^{-k}. Repeating $O\left(e^k\right)$ times will give close to 1 probability with time $O\left(m \cdot e^k \cdot k!\right) = O\left(m \cdot k^{k+1}\right)$.