THORE HUSFELDT

NOTES FOR EDAN 55

These notes are supplementary material for my course on advanced algorithms, EDAN55 at Lund University. The material works best as an extension to the textbook of Kleinberg and Tardos, *Algorithms Design*, even though it has been collected from various sources and does not present itself with any consistency in prose, depth, or ambition.

Chapter 10

This note extends the presentation in Kleinberg and Tardos, chapter 10 with a case study of finding a k-path in a graph. In particular, it introduces Bodleander's algorithm. This algorithm provides a good example of dynamic programming over a tree decomposition, but, more importantly, finds a (not necessarily optimal) tree decomposition in a natural way. Moreover, we see the classical colour coding technique. That section makes sense after an introduction to randomized algorithms, such as chapter 13 *ibid*. Section 10.7 makes sense after an introduction to exponential time algorithms, such as Chapter 10^7 in the present set of notes.

10.6 Case study: k-path

A k-path in a graph is a simple path of length k, i.e., a sequence of distinct vertices $(v_1, \ldots, v_k)$ such that $v_i v_{i+1} \in E$ for $1 \leq i < k$.

The k-path problem is given a connected graph $G = (V, E)$ and integer k, determine if G has a k-path. The problem makes sense for both directed and undirected graphs. Setting $k = |V|$ the problem is known as the Hamiltonian path problem, which is well known to be NP-hard. In particular, there is little hope of solving the k-path problem in time polynomial in n and k.

First attempts

The brute force attempt is to check every subset of k vertices and see if they form a simple path in G by considering all of their orderings. The running time is within a polynomial factor of $O(\binom{n}{k}k!k)$.

We can use *decrease-and-conquer* from each starting vertex $v \in V$ and iterate over all neighbours. Indeed, if we let $P_i(v_1)$ denote the set of sequences $v_1, \ldots, v_i$ of neighbouring vertices in G (not necessarily simple), then

$$P_i(v_1) = \bigcup_{v_2 \colon v_1 v_2 \in E} \{\, v_1 \cdot \alpha \colon \alpha \in P_{i-1}(v_2), v_1 \notin \alpha \,\}, \qquad (1)$$

where $\cdot$ denotes concatenation.[1] This produces all sequences of distinct neighbouring vertices of length k in G, their number is

[1] Maybe give as pseudocode instead.

$n(n-1)\cdots(n-k-1)$, the *falling factorial* $n^{\underline{k}} = O(n^k)$. We need to check each of them to see that it is simple; the total time is within a polynomial factor of $O(n^k k)$.

FPT for regular graphs

Assume that G is regular, i.e., all vertices $u \in V$ have the same degree $\deg(u) = \delta$. In this case it is easy to see that the k-path problem is FPT.

If $k \leq \delta$ then the k-path problem can be solved by depth first search: Start at an arbitary vertex, mark it, and go to an unmarked neighbour, until k vertices are marked. At no intermediate stage can you find yourself surrounded by k marked vertices, so there is always an unmarked neighbour. The running time is $O(k\delta) = O(kn)$.

If $k > \delta$ then the decrease-and-conquer approach works. The number of neighbours considered at each step in (1) is δ, so the total number of sequences constructed can be bounded by $\delta^{\underline{k}}$, and the total running time becomes $O(k^k)$.

Bodlaender's algorithm

Many of the algorithmic tools for algorithms on tree decompositions were developed by Hans Bodlaender. In particular, an early paper[2] develops an FPT algorithm for k-path for general graphs.

[2] Hans L. Bodlaender. *On linear time minor tests with depth-first search.* J. Algorithms, 14(1):1–23, 1993.

Perform a depth first search (DFS) from an arbitrary vertex r, constructing a DFS tree T rooted at r. If the depth of T is at least k, then the corresponding path from root to leaf is a k-path, and we're done.

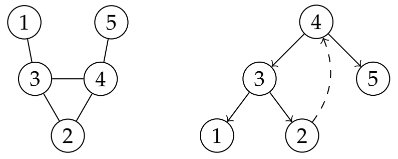


Figure 1: The Bull graph and a DFS traversal starting at vertex 4.

Otherwise, the root-leaf paths (all of which have length less than k) form the pieces of a tree-decomposition of size k. To be precise, for every node t of T, let V_t consist of the vertices on the unique path from t to r in T.

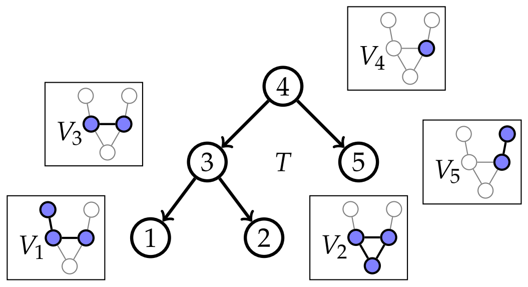


Figure 2: A tree decomposition for the Bull graph with V_t shown for every $t \in T$.

We claim that $(T, \{V_t : t \in T\})$ is tree decomposition of G of treewidth k.

Node coverage. Node coverage is easily established in this tree decomposion, because every vertex in v is also a node in T. In particular, v belongs to V_v. (In general, the tree T in a tree decomposition can have completely different nodes than G.)

Edge coverage. A fundamental property of DFS trees is that they have no "crossing edges:" every edge in the graph goes from a vertex in the DFS tree to its ancestor, cf. (3.7) in [KT, p. 85]. Therefore every graph edge is fully contained in some piece. (For instance, the graph edge $uv \in E$ is fully contained in V_t for every t in the subtree of T rooted at u.)

Coherence. Let t_1, t_2, t_3 be three nodes of T such that t_2 lies on the path from t_1 to t_3. Let $v \in V$ belong to both V_{t_1} and V_{t_3}. Since both V_{t_1} and V_{t_3} contain the vertex v, it must lie on both the path from t_1 to r and from t_3 to r in T. In particular, v is a common ancestor in T of t_1 and t_3. If t_2 lies on the path from t_2 and t_3 then v must be an ancestor of t_2.[3] But then v lies on the path from t_2 to r, in particular it belongs to V_{t_2}.

[3] True, but probably needs a case analysis. Note to self: work this out.

Tree-width. Every piece contains at most $k - 1$ elements, because the distance in T from t to r is less than k.

We solve the k-path problem using dynamic programming over the tree decomposition $(T, \{V_t \colon t \in T\})$ in same the fashion of the maximum independent set algorithm of section 10.4.

Modifying the tree-decomposition. We will exploit the fact that the tree decomposition defined above has very special structure, namely that if t_1 is a parent of t_2 in the tree decomposition thne $V_{t_1} \subseteq V_{t_2}$. In fact, we have $V_{t_1} = V_{t_2} - \{t_2\}$. This makes our constructions slightly simpler than for a general tree decomposition. However, our assumption is not crucial: an algorithm for finding a longest path in graph of tree-width k using a general tree decomposition can also be given, and within the same time bounds.

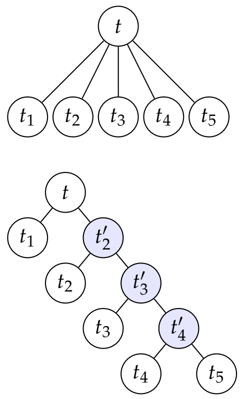


Figure 3: Transformation of a node $t \in T$ with more than 2 children.

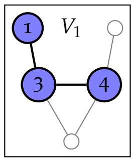
However, there is a further simplification of the tree decomposition that we want to perform before moving on. Unlike the example in fig. **??**, the DFS tree T can have high degree, so we transform T into a binary tree using a straightforward modification. Suppose that node t has children $t_1, \ldots, t_d$ with $d > 2$. Remove the edges (t_i, t) for $i = 2, \ldots, d$ and introduce fresh nodes t'_i for each $i = 2, \ldots, d - 1$. Then connect these nodes into a binary tree by adding the edges (t_i, t'_i) for $i = 2, \ldots, d - 1$, the edge (t'_i, t'_{i-1}) for $i = 3, \ldots, d - 1$, and finally the edges (t'_2, t) and (t_d, t'_{d-1}). See figure 3.

This results in a new, binary tree T'. We complete the tree decomposition by specifying the pieces associated with the new nodes t'_i by setting $V_{t'_i} = V_t$ for all $i = 2, \ldots, d$. In other words, every fresh node is associated with the piece of the original parent t. It is straightforward to check that this is still a tree-composition of tree-width k. We will abuse notation and continue using the letter T for the transformed tree T'.

Defining the subproblems. As in section 10.4, let T_t denote the subtree of T rooted at t, let V_t be the vertices associated with $t \in T$, and let G_t denote the subgraph of G induced by the vertices in the pieces associated with the nodes of T_t.

We define the subproblems of our dynamic programming solution for each subtree T_t as follows. Let $\overline{w} = w_1, \ldots, w_r$ be a sequence of vertices from V_t without repetitions, and set $W = w_1, \ldots, w_r$. Then the subproblem $f(\overline{w})$ is defined as the length of a longest simple path in G_t that is *consistent* with the ordering $w_1, \ldots, w_r$. By consistent we mean that the path visits the vertices from W in the order given by $\overline{w}$; the path can visit other vertices in $G_t \setminus V_t$ in between, but no vertices in $V_t \setminus W$. If no such path exists, the we set $f(\overline{w}) = 0$.

The number of subproblems at node t is $\sum_{r=0}^{k} \binom{k}{r} r! \leq k!2^k$.



$\overline{w}$	$f(\overline{w})$
1	0
3	0
4	0
1, 3	1
3, 1	1
1, 4	0
4, 1	0
3, 4	1
4, 3	1
1, 3, 4	2
1, 4, 3	0
3, 1, 4	0
3, 4, 1	0
4, 1, 3	0
4, 3, 1	2

Figure 4: The subproblems at V_1.

Building Up Solutions. We proceed to show how solutions to subproblems are constructed.

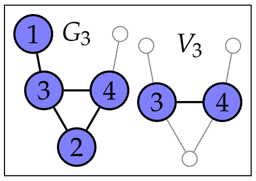
For a leaf t, the subgraph G_t is just the graph induced by V_t. We will solve this suproblem by exhaustive search. To be precise, to compute the subproblem f at a leaf node t we iterate over all choices of $W \subseteq V_t$, and all $r!$ orderings $w_1, \ldots, w_r$ of the $r = |W|$ vertices in W, and check that the sequence of vertices $w_1, \ldots, w_r$ defines a simple path, i.e., we check that that $w_i w_{i+1} \in E$ for $1 \leq i < r$. If so, we set $f(\overline{w}) = r - 1$, otherwise 0.

Suppose node t' is a child of node t in T. Let $\overline{w}' = w'_1, \ldots, w'_r$ denote a subproblem associated with t' and let $\overline{w} = w_1, \ldots, w_r$ denote a subproblem associated with t. We say that $\overline{w}'$ is *compatible* with $\overline{w}$ if the two sequences contain the same vertices in the same order, except for possibly t' itself as a detour. Formally, either $\overline{w}' = \overline{w}$ or there is some j such that $\overline{w}' = w_1, \ldots, w_j, t', w_{j+1}, \ldots, w_r$ with $w_j t' \in E$ and $t' w_{j+1} \in E$.

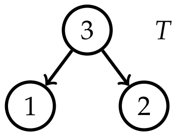
Now consider a node t with two children t_1 and t_2 and assume we already computed the optimum solutions f_1 and f_2 for all subproblems associated with the children. Then for subproblem $\overline{w}$ of length r set

$$f(\overline{w}) = \max_{\overline{w}_1, \overline{w}_2} \{ f_1(\overline{w}_1), f_2(\overline{w}_2), f_1(\overline{w}_1) + f_2(\overline{w}_2) - (r-1) \},$$

where the maximum is taken over all subproblems $\overline{w}_i$ associated with t_i for $i = 1, 2$ that are compatible with $\overline{w}$. Note that these subproblems encode all ways of traversing G_t because there is no edge between t_1 and t_2 in G.

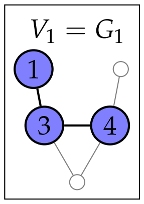


$\overline{w}_3$	$f(\overline{w}_3)$
3, 4	3
4, 3	3



$\overline{w}_2$	$f_2(\overline{w}_2)$
2, 3	1
2, 4	1
3, 2	1
3, 4	1
4, 2	1
4, 3	1
2, 3, 4	2
2, 4, 3	2
3, 2, 4	2
3, 4, 2	2
4, 2, 3	2
4, 3, 2	2

$\overline{w}_1$	$f_1(\overline{w}_1)$
1, 3	1
3, 1	1
3, 4	1
4, 3	1
1, 3, 4	2
4, 3, 1	2



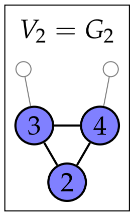


Figure 5: Computation of the subproblems associated with subtree T_3 in the Bull graph example. The value $f_3(3,4)$ at T_3 is computed as follows. At the left subproblem T_1, the sequence $1,3,4$ is consistent with $3,4$. (The value $f_1(1,3,4) = 2$ corresponds to the path $1,3,4$ in G_1.) At the right subproblem T_2, the sequence $3,2,4$ is consistent with $3,4$. (The value $f_2(3,2,4) = 2$ corresponds to the path $3,2,4$ in G_2.) The value $f_3(3,4) = f_1(1,3,4) + f_2(3,2,4) - 1 = 3$ is then the the longest simple path in G_3 that visits the vertices 3 and 4 in the order $3,4$. (It corresponds to the path $1,3,2,4$ in G_3.)

Nodes with only one child are handled in a similar fashion.

The time for the computation of $f_t(\overline{w})$ is $O(k^2)$, so f_t is computed for all subproblems in time $O(k!2^k k^2)$. Finally, the total time for the entire computation becomes $O(k!2^k k^2 \cdot m)$. This is of the desired form $f(k)n^{O(1)}$.

Colour coding

A famous randomized algorithm by Alon, Yuster, and Zwick improves Bodlaender's algorithm both in running time and simplicity of exposition.

1. Give each vertex $v \in V$ a random value $\chi(v) \in \{1, 2, \ldots, k\}$, called a *colour*.
2. Find a *rainbow coloured* k-path, i.e., a k-path on which every colour appears. (And therefore appears exactly once.)

Note that the colouring is not "proper" in the sense of vertex colouring, so edges uv with $\chi(u) = \chi(v)$ can occur.

Consider a k-path P in the given graph. The vertices of P admit k^k different colourings, of which $k!$ are rainbow colourings. Thus the event R that P is rainbow coloured happens with probability

$$\Pr(R) = \frac{k!}{k^k} \geq \sqrt{2\pi k}\frac{1}{e^k}\,.$$

using Stirling's formula, $r! \geq \sqrt{2\pi r}\left(\frac{r}{e}\right)^r$. The remarkable thing is that $\Pr(R)$ is merely exponential in k, instead of $1/k!$.

Determining if a coloured graph contains a rainbow k-path can be accomplished using dynamic programming. For every subset $X \subseteq \{1, \ldots, k\}$ of colours and vertex $u \in V$, let $P(X, u)$ be true if there is a path of length $|X|$ starting in u that uses exactly the colours in X. (In particular, such paths are simple.) Then

$$P(X,u) = \bigwedge_{uv \in E} P(X - \chi(u), v) \quad \text{for } \chi(v) \in X, |X| > 1\,,$$

and $P(\{r\}, u)$ true if and only if $r = \chi(u)$. The graph G has a rainbow coloured k-path if and only if $P(\{1, \ldots, k\}, u)$ holds for some u. Using dynamic programming, the values $P(\{1, \ldots, k\}, u)$ can be computed in time $O(2^k n)$ for every u, so the rainbow coloured P can be detected in time $O(2^k n^2)$. The algorithm takes $O(2^k n)$ space.

By repeating the procedure $t = 1/\Pr(R)$ times, the path P becomes rainbow coloured (and is therefore detected) with constant nonzero probability

$$\bigl(1 - \Pr(R)\bigr)^{1/\Pr(R)} \geq \tfrac{1}{4}$$

in FPT time

$$O(2^k n^2 \Pr(R)^{-1}) = O(2^k n^2 e^k) = O(5.44^k n^2)\,.$$

10.7 Exponential time and parameterized complexity

FPT

The class FPT contains the class of parameterized decision problems that are *tractable* in the following sense: If the problem is parameterized by n (say, the size of the vertex set of a graph) and k (say, the size of the solution) then we say that the problem *is FPT* if it admits an algorithm with running time $f(k)n^{O(1)}$ for some function k.

Think of the function f as a singly- or doubly exponential function, typically $f(k) = \exp(O(k))$ or $f(k) = O(k!)$ or $f(k) = 2^{2^k}$. In general, f can be much crazier, famous examples include towers of exponents. For a rigorous presentation (which does not interest us here), we need to require f to be computable.

The canonical example of such a problem is k-Vertex Cover, the problem considered in [KT, sec. 10.1], which admits an algorithm with running time $O(2^k kn)$. (Every parameterised problem in P also admits such an algorithm.)

Counterexamples are:

1. The k-Clique problem. Here, the naive algorithm takes time $O(n^k)$ (by iterating over all $\binom{n}{k}$ subsets), and we do not know how to disentagle the dependencies on n and k.

2. The k-Colouring problem. Here, the best algorithm takes time $O(2^n n^2)$, so k does not even seem to appear in the running time.

We are pretty confident about the second point. If k-Colouring *were* FPT then there would be an algorithm that solved the special case $k = 3$ in polynomial time. (Because $f(3)$ is just a constant, no matter what f is.) So an FPT algorithm for k-Colouring would imply that P and NP collapse.

We return to an argument for the first point shortly.

Exponential Time Hypotheses

We have seen a bunch of algorithms that solve the 3-Sat problem faster than mindlessly checking all 2^n assignments. The resulting running times were all of the form $(1+\epsilon)nn^{O(1)}$ for smaller and smaller ϵ. At the time of writing, the best algorithm is $O(1.31^n)$. Can this process continue? If yes, how far? Is there a $O(2^{n/\log\log\log n})$ algorithm? $O(2^{\sqrt{n}})$?

Many people think not. To formalise this intuition, computational complexity theorists have introduced the *Exponential Time Hypothesis* (ETH) Here are some formulations of this idea:

- There is no $\exp(o(n))$ algorithm for 3-Sat, where n is the number of variables.

- There is no $\exp(o(k))$ algorithm for 3-Sat, where k is the number of clauses.

- There exists a constant $\delta > 0$ for which there is no $O((1+\delta)^n)$ algorithm for 3-Sat.

It turns out that the first two formulations are equivalent; this is the "Sparsification Lemma," which is not trivial to prove. Less interestingly, the third formulation is not equivalent (and slighly stronger), but these details need not concern us here.

For completeness, we note that ETH is a (much) stronger hypothesis than P $\neq$ NP.

Independent Set. We can use reductions to argue "under ETH" much like we use them to argue "under P $\neq$ NP." For instance, return to the lower bound for independent set in [KT, (8.8)]. (Recall that we have an algorithm with running time $O(1.3808^n)$ in chapter 2^{10}.) The 3-Sat instance with k clauses was transformed into an Independent Set instance with $O(k)$ vertices. If there was an $\exp(o(|V(G)|))$ algorithm for Minimum Independent Set then there would be an $\exp(o(n))$ algorithm for 3-Sat. We conclude that under ETH, Independent Set requires $O((1+\delta)^n)$ time for some $\delta > 0$. This result implies the same lower bound for Maximum Clique.

3-Colouring. The current best bound for 3-colouring is $O(1.3289^n)$. But we can argue as above, using the reduction in [KT, Sec. 8.7] that no $\exp(O(n))$ algorithm exists.

Dependency on m. Here's something more surprising:

Consider the Maximum Clique problem. It admits and algorithm in time $\exp(o(m))$. Let us look for k-cliques of size $k = 1, 2, \ldots,$ until the answer is no. If $m < \binom{k}{2}$ we can quickly answer 'no,' because a k-clique needs at least $\binom{k}{2}$ edges. Otherwise, we check all $\binom{n}{k}$ subsets by brute force. The running time within a polynomial factor of

$$\binom{n}{k} \leq n^k = 2^{k \log n} \leq 2^{O(\sqrt{m} \log m)}$$

where the last inequality uses that $m \geq \binom{k}{2} = \frac{1}{2}k(k-1)$. This, *very much unlike 3-Sat*, there is an algorithm for Maximum Clique that is subexponential in the instance size.

Even more surprising is the observation that this argument does not work for Independent Set. Let's try to prove that. First, a failed attempt: Redo the proof of [KT, (8.8)] and note that that gadget construction does *not* give us such a result (because the number of edges in the resulting graph is not linear in k). But a simple trick fixes this. (Introduce $2n$ new vertices, pairwise neighbouring, one for each variable. Connect them to the triangle gadgets and remove the 'old' Conflict edges. The resulting construction has $3k + 2n$ vertices and $O(k)$ edges.) We conclude that under ETH, ther is no $\exp(o(m))$ algorithm for independent set.

ETH and FPT

We are finally able to connect the various complexity theories. We return first to our open mystery about k-Clique from two subsections ago.

Assume that k-Clique were in FPT, so that there exists an algorithm with running time $f(k)n^r$ for some increasing f and constant r.

Given an instance G to 3-colouring of n vertices. Choose k maximal such that $f(k) \leq n$, and not for later reference that k is a growing function of n. (For instance, if $f(k) = 2^{2^k}$ then k is $\Omega(\log\log n)$.)

Construct an instance H to k-clique as follows. Split the vertex set of G into k equal sized sets $V_1, \ldots, V_k$. The graph H is k-partite with partition $U_1, \ldots, U_k$. Each vertex $c \in U_i$ corresponds to a proper 3-colouring of V_i. Thus, the size of U_i is bounded by roughly $3^{n/k}$. An edge joins c and c' in different parts of H if the two colourings are consistent. (In the following sense: if uv is an edge in G with u and v in different parts, then c maps u to a different colour than c' maps v.) Then H contains a k-clique if and only G can be 3-coloured.

The running time of this algorithm would be $f(k)(3^{n/k})^r$. By our choice of k, the first factor is bounded by n, and the second factor is $\exp(o(n))$, violating the Exponential Time Hypothesis.

In particular, we conclude that k-Clique does not allow an FPT algorithm unless ETH fails.

FPT and Approximation

Assume that P is an optimisation problem and let k denote the solution size. For concreteness, assume P is a maximisation problem. (Think of P as the k-clique problem, for instance.)

We will show that if P admits an FPTAS then P, parameterised by k, is an FPT problem.

Assume P can be solved with error ϵ in time $f(\epsilon)n^{O(1)}$ by algorithm A.[4]

[4] This is a weaker assumption than FPTAS, where f would be a polynomial. So we're proving a much stronger results.

We will design an FPT algorithm for P. Set $\epsilon = \frac{1}{2k}$ and run A. If the result is at least k, return 'yes', else return 'no.'

The running time is $f(2k)n^{O(1)}$, clearly FPT. It remains to check correctness. If the optimum solution size is at most $k-1$ then by assumption, the approximation algorithm can't find a larger solution either, so it returns at most $k-1$ and the FPT algorithm returns 'no.' If the optimum solution is at least k, the the approximation algorithm finds a solution of size at least

$$\frac{k}{1 - 1/(2k)} = k - \tfrac{1}{2},$$

so that the FPT algorithm is able to distinguish the two cases. In particular, the algorithm distinguishes the two cases, so it works as a decision algorithm for P.

Chapter 11

This note extends the presentation in Kleinberg and Tardos, chapter 11 with some inapproximability results, making plausible that the tools from computational complexity (from chapter 8) can be used to argue about approximation quality.

11.9 Inapproximability

No FPTAS for Max 3-Sat

In [KT, 11.8] we saw that there is an algorithm for the Knapsack problem that given any $\epsilon > 0$ computed a feasible solution to an instance of size n in time $O(n^3\epsilon^{-1})$ that is at most a factor $(1+\epsilon)$ below the maximum possible.

Such an algorithm is called a *fully polynomial time approximation scheme.*

Such an algorithm cannot exist for Max 3-Sat, the problem considered in [KT 13.4].

Theorem 1 *There is no FPTAS for for Max 3-Sat unless P = NP.*

Proof. Assume there was such an FPTAS. Let ϕ be an instance to the decision problem 3-Sat with m clauses. (The total size of ϕ is $O(m)$.) Set $\epsilon = \frac{1}{2m}$ and run the FPTAS.

If ϕ is satisifiable, all $m =:$ OPT clauses can be satisfied, so the FPTAS returns a solution of size at least $(1-\epsilon)m = m - \frac{1}{2}$. Since the solution size is an integer, the solution size is equal to m, so the FPTAS has solved the decision problem.

The running time of the FPTAS is polynomial in the input size m and inverse polynomial in the approximation guarantee $\frac{1}{2m}$, so the algorithm runs in polynomial time. $\square$

It is known (but far beyond the scope of these notes) that for any $\epsilon > 0$ there is no polynomial-time algorithm for 3-Sat that satisfies more than $\frac{7}{8} + \epsilon$ of the clauses, unless P = NP.[5]. This is a tight bound for Johnson's algorithm from [KT 13.4], so this particular problem is fully understood.

[5] Johan Håstad, *Some optimal inapproximability results*, Journal of the ACM (ACM) 48: 798–859, 2011.

TSP

We first present a simple approximation algorithm for the Traveling Salesman Problem in *metric* graphs, i.e., the distance function satisfies the triangle inequality $d(u,w) \leq d(u,v) + d(v,w)$ for all vertices u, v, w.

Theorem 2 *If the distances in an instance to TSP satisfy the triangle inequality then there is a polynomial-time 2-approximation algorithm.*

Proof.

1. Find a minimum spanning tree T of the given graph G.
2. Perform a depth-first search of T from vertex 1.
3. Visit the vertices of the given graph in the depth first search order, and finally return to 1.

To see that this works, let OPT denote the length of an optimal TSP tour in G. Note that the total weight $|T|$ is at most OPT, because removing any edge from a tour makes the tour a spanning tree, and T is the minimum spanning tree in G. Consider now the nonsimple tour T' given by a traversal of T. This tour has length $2|T|$, because it travels along every edge of T exactly twice. Note that the depth first search order is a subsequence of T', so it can be viewed as the result of a successive application of operations that replace sequences u, v, w by the sequence u, w. If the distances in G satisfy the triangle inequality, none of these operations can increase the length of the tour. In particular, the resulting tour has length at most 2OPT. □

A simple modification of this algorithm reduces the approximation factor from 2 to $\frac{3}{2}$; this is a classical result.[6] With even more restrictions on the distance measure, much faster algorithms are possible: If the require that the distance function is Euclidian then TSP can be approximated within a factor $(1 + \epsilon)$ in polynomial time for any given $\epsilon > 0$.[7]

[6] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1976.

[7] Sanjeev Arora. *Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems.* Journal of the ACM, Vol.45, Issue 5, pp.753–782, 1998. J.S.B. Mitchell, *Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems*, SIAM Journal on Computing 28 (4): 1298–1309, 1999.

Is the assumption about the triangle inequality crucial? It turns out that the answer is yes.

Theorem 3 *There can be no polynomial-time 2-approximation algorithm for TSP unless $P = NP$.*

Proof. We reduce from Hamiltonian Cycle, which is known to be NP-hard. Given an instance $G = (V, E)$ to Hamiltonian Cycle, build an instance $K = (V, E')$ to TSP as follows. The edge set is the complete set $E' = \{\,\{u,v\} : u \in V, v \in V\,\}$, and the distances are given by

$$c(e) = \begin{cases} 1, & e \in E; \\ |V| + 2, & e \notin E. \end{cases}$$

If G is a yes-instance, the shortest TSP tour has length $|V|$, so the hypothetical 2-approximation algorithm would return a solution of value at most $2|V|$.

If G is a no-instance, the shortest TSP tour must include at least one edge from $E' - E$, so the optimal tour has length at least $(|V| + 2) + |V| - 1 = 2|V| + 1$.

In particular, the approximation algorithm distinguishes yes- and no-instances to the Hamiltonian Cycle problem in polynomial time. □

The same proof rules out much less impressive approximation factors as well, such as $\rho = 10$ or $\rho = 10^{10}$. To show that TSP admits no polynomial-time ρ-approximation algorithm, the cost function in the proof has to be set to

$$c(e) = \begin{cases} 1\,, & e \in E\,; \\ (\rho - 1)|V| + 2\,, & e \notin E\,. \end{cases}$$

In fact, the proof works even for nonconstant approximation factors, and even superpolynomial ones. To prove that TSP cannot be approximated within $\exp(\Omega(|V|^{1/3}))$, set

$$c(e) = \begin{cases} 1\,, & e \in E\,; \\ 2^{|V|}|V| + 1\,, & e \notin E\,. \end{cases}$$

This is as far as we can push this proof, because the space needed to store the values on the edges start to dominate the instance size.

Approximation-preserving reductions

The examples so far established approximation hardness by reducing directly from a decision problem. However, there is a rich theory about reducibility *among* approximation problems as well.

For a simple example, consider the well-known reduction from 3-Sat to Independent Set in [KT, (8.8)]. There, it was given as a reduction between two *decision* problems. But he same construction can be used to reason about interdependencies with respect to approximation quality instead. These are called approximation-preserving reductions. A completely formal treatment of these notions is beyond the scope of these notes.

Theorem 4 *If Independent Set can be approximated within a factor $(1 + \epsilon)$ in polynomial time then so can Max 3-Sat.*

Proof. Given an instance ϕ to Max 3-Sat with m clauses, follow the reduction in [KT, (8.8)] to build an instance G of Independent Set.

The graph G contains $3m$ vertices. We note from the reduction that if the maximum number of satisfiable clauses ϕ is OPT then the largest independent set in G has size OPT as well. Conversely, every indpendent set of size k in G corresponds to an assignment in ϕ that satisfies at least k clauses.

Thus, a $(1+\epsilon)$-approximation algorithm for Independent Set would achieve the same approximation guarantee for Max 3-Sat. □

In particular, we can use this reduction and the result of Håstad mentioned earlier to establish that no polynomial-time algorithm can approximate the size of a maximum independent set in a graph better than $\frac{7}{8}+\epsilon$. However, unlike the case for Max 3-Sat, this is far from optimal. It is known (but far beyond the scope of these notes) that Independent Set cannot be approximated in polynomial time within a factor $n^{1-\epsilon}$ for every $\epsilon > 0$, unless $\mathrm{P} = \mathrm{NP}$.[8]

[8] David Zuckerman, *Linear degree extractors and the inapproximability of max clique and chromatic number*, Proc. 38th ACM Symp. Theory of Computing, pp. 681–690, 2006.

Chapter 2^{10} Exponential Time Algorithms

These lecture notes were originally prepared for the AGAPE 2009 Spring School on Fixed Parameter and Exact Algorithms, May 25-29 2009, Lozari, Corsica (France).

This document attempts to survey techniques that appear in exact, exponential-time algorithmics using the taxonomy developed by Levitin. The purpose is to force the exposition to adopt an alternative perspective over previous surveys, and to form an opinion of the flexibility of the taxonomic framework of Levitin.[9]

I have made no attempt to be comprehensive. A recent textbook by Fomin and Kratsch covers the material in much more depth.[10]

[9] Levitin, *Introduction to the Design & Analysis of Algorithms*, Addison–Wesley, 2003.

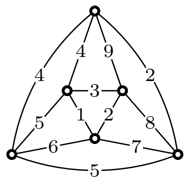
[10] Fedor Fomin and Dieter Kratsch, *Exact Exponential Algorithms*, Springer, 2010.

Brute force

A brute force algorithm simply evaluates the definition, typically leading to exponential running times.

Some representative problems

TSP. Our first example is the Traveling Salesman Problem. Given a weighted graph like

$G =$ 

with n vertices $V = \{v_1, \dots, v_n\}$ (sometimes called "cities") the *traveling salesman problem* is to find a shortest Hamiltonian path from the first to the last city, i.e., a path that starts at $s = v_1$, ends at $t = v_n$, includes every other vertex exactly once, and travels along edges whose total weight is minimal. Formally, we want to find

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

where the sum is over all permutations π of $\{1,2,\ldots,n\}$ that fix 1 and n. When the weights are uniformly 1, the problem reduces to deciding if a Hamiltonian path at all.

This above expression can be evaluated within a polynomial factor of $n!$ operations. In fact, because of certain symmetries it suffices to examine $(n-2)!$ permutations, and each of these requires take $O(n)$ products and sums. On the other hand, it's not trivial to iterate over precise these permutations in time $O((n-2)!)$. We will normally want to avoid these considerations, since they only contribute a polynomial factor, and write somewhat imprecisely $O^*(n!)$, where $O^*(f(n))$ means $O(n^c f(n))$ for some constant c.

Independent set. An *independent set* in an n-vertex graph $G=(V,E)$ is a subset of vertices $U \subseteq V$ where no edge from E has both its enpoints in U. Such a set can be found by considering all subsets (and checking independence of each), in time $O^*(2^n)$.

Satisfiability. The *3-Satisfiability problem* is given by a Boolean formula ϕ on variables $x_1,\ldots,x_n$ is on 3-conjunctive normal form if it conists of a conjunction of m *clauses*, each of the form $(a \vee b \vee c)$, where each of the *literals* a, b, c is a single variable or the negation of a single variable. The satisfiability problem for this class of formulas is to decide if ϕ admits a satisfying assignment. This can be decided by considering all assignments, in time $O^*(2^n)$. (Note that m can be assumed to be polynomial in n, otherwise ϕ would include duplicate clauses.)

Counting perfect matchings. A *perfect matching* in a graph $G=(V,E)$ is an edge subset $M \subseteq E$ that includes every vertex as an endpoint exactly once; in other words

$$|M| = \tfrac{1}{2}|V| \qquad \bigcup M = V.$$

In fact, famously, a matching can be found in polynomial time, so we are interested in the counting version of this problem: how many perfect matchings does G admit? From the definition, this still takes $O^*(2^m)$ time.

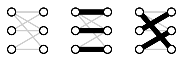


Figure 6: A bipartite graph and 2 of its 3 perfect matchings.

We will look at this problem for bipartite graphs as well as for general graphs.

THE PROBLEMS ABOVE ARE ALL DIFFICULT PROBLEMS, hard for complexity classes such as NP or #P, so we cannot expect to devise algorithms that run in polynomial time. Instead, we will improve the exponential running time. For example, for some problems we will find vertex-exponential time algorithms, i.e., algorithms with running time $\exp(O(n))$ instead of $\exp(O(m))$ or $O^*(n!)$ $O^*(n^n)$. Other algorithms will improve the base of the exponent, for example from $O^*(2^n)$ to $O(1.732^n)$.

Generating permutations and combinations

It is not completely straightforward to iterate over all subsets, k-subsets or permutations of $[n] = \{1,\ldots,n\}$. Knuth devotes over 300 pages to these questions.[11]

[11] Donald E. Knuth, *The Art of Computer Programming*, Vol. 4: Combinatorial Algoriths, sec. 7.2.1.1–7.2.1.4. Addison–Wesley, 2011.

Subsets. If n is smaller than the number of bits in a machine word, we can use the simple correspondance between binary numbers and incidence vectors of subsets. The set $S \subseteq [n]$ then corresponds to the bit string with $b_i = [i \in S]$. We can then generate all subsets of $[n]$ using machine arithmetic, counting from 0 to 2^n. For larger n one needs to simulate the "binary counter" logic: starting from the right, find the first 0, flip it. If its left neighbour is a 1, proceed 0ing it and continue to the left.

Permutations. Let $(a_1,\ldots,a_n)$ be a permutation of $[n]$. Then the lexicographically next permutation is given by the following procedure:

1. find the largest index i such that $a_i < a_{i+1}$
2. switch a_i with the smallest value in $(a_{i+1},\ldots,a_n)$ larger than a_i.
3. sort $(a_{i+1},\ldots,a_n)$

Combinations. Let $(a_,\ldots,a_k)$ denote a k-subset of $[n]$ in sorted order. Starting with $(1,2,\ldots,k)$, and ending in $(n-k+1, n-k+2,\ldots,n)$, the next k-subset is given by the following procedure:

1. find the largest index i such that $a_i \neq n-k+i$
2. increase a_i by 1
3. for $j = i + 1tok$, set $a_j = a_i + j - i$

Greedy

A greedy algorithm does "the obvious thing" for a given ordering, the hard part is figuring out *which* ordering. A canonical example is interval scheduling.

In exponential time, we can consider *all* orderings. This leads to running times around $n!$ and is seldom better than brute force, so this class of algorithms does not seem to play a role in exponential time algorithmics. An important exception is given as an exercise.

Recursion

Recurrences express the solution to the problem in terms of solutions of subproblems. Recursive algorithms compute the solution by applying the recurrence until the problem instance is trivial.

Decrease and conquer

Decrease and conquer reduces the instance size by a constant, or a constant factor. Canonical examples include binary search in a sorted list, graph traversal, or Euclid's algorithm.

In exponential time, we produce several smaller instances (instead of just one), which we can use this to exhaust the search space. Maybe "exhaustive decrease and conquer" is a good name for this variant—this way, the technique becomes an umbrella of exhaustive search techniques such as branch-and-bound.

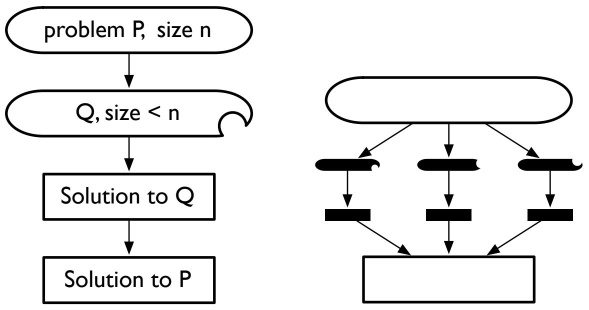


Figure 7: Decrease and conquer with one (left) and many (right) subproblems.

3-Satisfiability. An instance to 3-Satisfiability includes at least one clause with 3 literals. (Otherwise it's an instance of 2-Satisfiability, which can be solved in polynomial time.) Pick such a clause and construct three new instances:

T⋆⋆ set the first literal to true,

FT⋆ set the first literal to false and the second to true,

FFT set the first two literals to false and the third to true,

These three possibilites are disjoint and exhaust the satisfying assignments. (In particular, FFF is not a satisfying assignment.)

Each of these assignments resolves the clause under consideration, and maybe more, so some cleanup is required. In any case, the number of free variables is decreased by at least 1, 2, or 3, respectively. We can recurse on the three resulting three instances, so the running time satisfies

$$T(n) = T(n-1) + T(n-2) + T(n-3) + O(n+m)\,.$$

The solution to this recurrence is $O(1.8393^n)$. (The analysis of this type of algorithm is one of the most actively researched topics in exact exponential-time algorithmics and very rich.)

Independent set. Let v be a vertex of with at least three neighbours. (If no such vertex exists, the independent set problem is easy.) Construct two new instances to independent set:

$G[V-v]$ the input graph with v removed. If $I \not\ni v$ is an independent set in G then it is also an independent set in $G[V-v]$.

$G[V-N(v)]$ the input graph with v and its neighbours removed. If $I \ni v$ is an independent set in G, then none of v's neighbours belong to I, so that $I - \{v\}$ is an independent set in $G[V-N(v)]$.

These two possibilities are disjoint and exhaust the independent sets.

We recurse on the two resulting instances, so the running time is no worse than

$$T(n) = T(n-1) + T(n-4) + O(n+m)\,.$$

The solution to this recurrence is $O(1.3803^n)$.

TSP. Galvanized by our successes we turn to TSP.

For each $T \subseteq V$ and $v \in T$, denote by $\mathrm{OPT}(T,v)$ the minimum weight of a path from s to v that consists of exactly the vertices in T. To construct $\mathrm{OPT}(T,v)$ for all $s \in T \subseteq V$ and all $v \in T$, the algorithm starts with $\mathrm{OPT}(\{s\},s) = 0$, and evaluates the recurrence

$$\mathrm{OPT}(T,v) = \min_{u \in T\setminus\{v\}} \mathrm{OPT}(T \setminus \{v\}, u) + w(u,v)\ . \tag{2}$$

While this is correct, there is no improvement over brute force: the running time is given by

$$T(n) = n \cdot T(n-1)$$

which solves to $O(n!)$. However, we will revisit this recurrence later.

Divide and conquer

The divide and conquer idea partitions the instance into two smaller instances of roughly half the original size and solves them recursively. Mergesort is a canonical example.

An essential question is *how* to partition the instance into smaller instances. In exponential time, we simply consider *all* such partitions. This leads to running times of the form

$$T(n) = 2^n n^{O(1)} T\left(\tfrac{1}{2}n\right),$$

which is $O(c^n)$, and the space is polynomial in n. Maybe "exponential divide and conquer" is a good name for this idea.

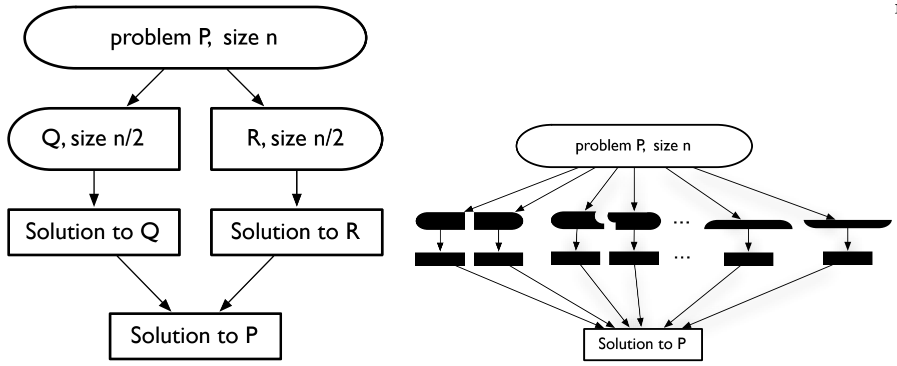


Figure 8: Divide and conquer with one division (top) and an exponential number of divisions (bottom).

TSP. Let $\mathrm{OPT}(U, s, t)$ denote the shortest path from s to t that uses exactly the vertives in U. Then we have the recurrence

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,, \tag{3}$$

where the minimum is over all subsets $S, T \subseteq U$ and vertices $m \in U$ such that $s \in S$, $t \in T$, $S \cup T = U$, $S \cap T = \{m\}$, and $|S| = \lfloor \frac{1}{2}n \rfloor + 1$, $|T| = n - |S| + 1$.

The divide and conquer solution continues using this recurrence until the sets U become trivial. At each level of the recursion, the algorithm considers $(n-2)\binom{n-2}{\lceil (n-2)/2 \rceil}$ partitions and recurses on two instances with fewer than $\frac{1}{2}n + 1$ cities. Thus, the running time is

$$T(n) = (n-2) \cdot \binom{n-2}{\lceil (n-2)/2 \rceil} \cdot 2 \cdot T(n/2 + 1)\,,$$

which solves to $O(4^n n^{\log n})$.

The space required on each recursion level to enumerate all partitionings is polynomial. Since the recursion depth is polynomial (in fact, logarithmic) in n, the algorithm uses polynomial space.

Transformation

Transformations compute a problem by computing a different problem in its stead. This can be called transform-and-conquer or reduction.

For exponential time algorithms, the reductions can involve the construction of an exponential number of instances (as in Moebius inversion), or be of exponential size (as in finding triangles).

Perfect matchings in bipartite graphs

Consider a bipartite graph on the disjoint vertex sets L and R, where $|L| = |R|$. Let A denote the biadjacency matrix of G defined as

$$a_{ij} = \begin{cases} 1\,, & \text{if } ij \text{ is an edge}\,; \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Then the number of perfect matchings in G is given by the expression

$$\sum_{f\colon L \to R} \prod_{i=1}^{|R|} a_{if(i)}\,,$$

where the sum is over all bijections f from L to R. This does not give us an interesting algorithm, because there are $|R|!$ such bijections.

We now construct a suprising reformulation of the above expression that can be evaluated much faster.

With foresight, for $S \subseteq R$ let $h(S)$ denote the number of ways to pick a neighbour in S for each vertex in L, such that each vertex in S is chosen at least once. (Algebraically, $h(S)$ can be given as

$$h(S) = \sum_{f\colon L \to S} \prod_{i=1}^{|R|} a_{if(i)}\,,$$

where the sum is over all surjective mappings from L to S.) The number of perfect matchings is $h(R)$. We have

$$\begin{aligned} \sum_{X \subseteq R} (-1)^{|R|-|X|} \sum_{S \subseteq X} h(S) &= \sum_{X \subseteq R} \sum_{S \subseteq X} (-1)^{|R|-|X|} h(S) \\ &= \sum_{S \subseteq R} \sum_{X\colon S \subseteq X \subseteq R} (-1)^{|R|-|X|} h(S) \\ &= \sum_{S \subseteq R} h(S) \sum_{X\colon S \subseteq X \subseteq R} (-1)^{|R|-|X|} = h(R)\,. \end{aligned}$$

The surprising step is the last. It holds because the inner alternating sum collapses to almost nothing:

$$\sum_{X\colon S \subseteq X \subseteq R} (-1)^{|R|-|X|} = \begin{cases} 1\,, & \text{if } S = R\,; \\ 0\,, & \text{if } S \neq R\,. \end{cases}$$

The first case is easy to see. The second case follows from a simple combinatorial fact, sometimes called the principle of inclusion–exclusion:

Lemma 1 *Let S and R be distinct sets with $S \subset R$. There are an equal number of odd-sized and even sized sets X with $S \subseteq X \subseteq R$.*

Proof. Let $i \in R \setminus S$. The mapping

$$X \mapsto X \oplus \{i\}$$

establishes a bijection between the odd-sized and even-sized sets X with $S \subseteq X \subseteq R$. □

We have

$$h(R) = \sum_{X \subseteq R} (-1)^{|R|-|X|} \sum_{S \subseteq X} h(S)\,,$$

which looks like no progress at all! The outer sum is over $2^{|R|}$ terms and the inner sum is over 2^k terms, where $k = |X|$. Worse, each term $h(S)$ is defined as a sum over all surjective mappings between two sets, which looks at least as hard to compute as the original problem.

But we can do much better, because the inner sum has a natural combinatorial interpretation: it is the number of way in which each vertex in L can pick a neighbour in X (withouth necessarily all neighbours in X getting picked.) Thus,

$$\sum_{S \subseteq X} h(S) = \prod_{i=1}^{|R|} \sum_{j \in X} a_{ij}\,.$$

Thus we have established *Ryser's formula*, that the number of perfect matchings is given by

$$\sum_{X \subseteq R} (-1)^{|R|-|X|} \prod_{i=1}^{|R|} \sum_{j \in X} a_{ij}\,,$$

and therefore computable in time $O(2^{|R|}|R|^2)$.

For more applications of this idea, see my survey.[12]

[12] T. Husfeldt, *Invitation to Algorithmic Uses of Inclusion–Exclusion*, 2011, arXiv:1105.2942.

Finding triangles

The number of triangles of undirected d-vertex graph T is given by

$$\tfrac{1}{6}\,\mathrm{tr}\,A^3\,,$$

where A denotes the adjacency matrix of T and tr, the *trace*, is the sum of the diagonal entries. To see this, observe that the ith diagonal entry counts the number of paths of length 3 from the ith vertex to itself, and each triangle contributes six-fold to such entries (once for every corner, and once for every direction).

To compute $A^3 = A \cdot A \cdot A$ we need two matrix multiplications, which takes time $O(d^\omega)$ for some $\omega < 3$, the best current bound is $\omega < 2.374$.

Independent set. We want to find an independent set of size k in $G = (V, E)$, and now we assume for simplicity that 3 divides k.

Construct $G' = (V', E')$, where each vertex $v \in V'$ corresponds to an independent set in G of size $\frac{1}{3}k$. Two vertices are joined by an edge $uv \in E'$ if their corresponding sets form an independent set of size $\frac{2}{3}k$. The crucial feature is that a triangle in G' corresponds to an independent set of size k in G. The graph G' has $\binom{n}{k/3} \leq n^{k/3}$ vertices, so the whole algorithm takes time $O^*(n^{\omega k/3})$, rather than the obvious $\binom{n}{k}$.

Perfect matching The next example, for Perfect Matchings, is somewhat more intricate, and uses both transformations from this section.

We return to perfect matchings, but now in regular graphs. Let $G[n = r; m = k]$ denote the number of induced subgraphs of G with r vertices and k edges. For such a graph, the number of ways to pick $\frac{1}{2}n$ edges is $k^{n/2}$, so we can rewrite

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y) = \sum_{k=1}^{m} \sum_{r=2}^{n} (-1)^r G[n = r; m = k] k^{n/2} .$$

Thus, we have reduced the problem to computing $G[n = r; m = k]$ for given r and k, and we'll now do this faster than in the obvious 2^n iterations.

We are tempted to do the following: Construct a graph T where every vertex corresponds to a subgraph of G induced by a vertex subset $U \subseteq V$ with $\frac{1}{3}r$ vertices and $\frac{1}{6}k$ edges. Two vertices in T are joined by an edge if there are $\frac{1}{6}k$ edges between their corresponding vertex subsets. Then we would like to argue that every triangle in T corresponds to an induced subgraph of G with r edges and k edges. This, of course, doesn't quite work because (1) the three vertex subsets might overlap and (2) the edges do not necessarily partition into such six equal-sized families. Once identified, these problems are easily adressed.

The construction is as follows. Partition the vertices of G into three sets V_0, V_1, and V_2 of equal size, assuming 3 divides n for readability. Our plan is to build a large tripartite graph T whose vertices correspond to induced subgraphs of G that are entirely contained in one the V_i.

Some notation: An induced subraph of G has r_1 vertices in V_1, k_1 edges with both endpoints in V_1, and k_{12} edges between V_1 and V_2. Define r_2, r_3, k_2 , k_3, k_{23}, and k_{13} similarly. We will solve the problem of computing $G[n = r; m = k]$ separately for each choice of these parameters such that $r_1 + r_2 + r_3 = r$ and $k_1 + k_2 + k_3 + k_{12} + k_{23} + k_{13} = k$. We can crudely bound the number of such new problems by $n^3 + m^6$, i.e., a polynomial in the input size.

The tripartite graph T is now defined as follows: There is a vertex for every induced subgraph $G[U]$, provided that U is entirely contained in one of the V_i, and contains exactly r_i vertices and k_i edges. An edge joins the vertices corresponding to $U_i \subseteq V_i$ and $U_j \subseteq V_j$ if $i \neq j$ and there are exactly k_{ij} edges between U_i and U_j in G. The graph T has at most $3 \cdot 2^{n/3}$ vertices and $3 \cdot 2^{2n/3}$ edges. Every triangle in T uniquely corresponds to an induced subgraph $G[U_1 \cup U_2 \cup U_3]$ in G with the parameters described in the previous paragraph.

The total running time is $O^*(n^{\omega k/3}) = (1.732^n)$.

Iterative improvement

Iterative improvement plays a vital role in efficient algorithms and includes important ideas such as the augmentating algorithms used to solve maximum flow and bipartite matching algorithms, the Simplex method, and local search heuristics. So far, very few of these ideas have been explored in exponential time algorithmics. An important exception is a local search procedure for satisfiability.

Local search

We consider 3-Satisfiability. Start with a random assignment to the variables. If all clauses are satisfied, we're done. Otherwise, pick a falsified clause uniformly at random, pick one of its literals unformly at random, and negate it. Repeat this local search step $3n$ times. After that, start over with a fresh random assignment. This proces finds a satisfying assignment (if there is one) in time $O^*\big((\frac{4}{3})^n\big)$ with high probability.

The analysis considers the number d of differences between the current assignment A and a particular satisfying assignment A^* (the Hamming distance). In the local search steps, the probability that the distance is decreased by 1 is at least $\frac{1}{3}$ (namely, when we pick exactly the literal where A and A^* differ), and the probability that the distance is increased by 1 is at most $\frac{2}{3}$. So we can pessimistically estimate the probability $p(d)$ of reducing the distance to 0 when we start at distance d ($0 \leq d \leq n$) by standard methods from the analysis of random walks in probability theory to

$$p(d) = 2^{-d}.$$

(Under the rug one finds an argument that we can safely terminate this random walk after $3n$ steps without messing up the analysis too much.)

The probability that a 'fresh' random assignment has distance d to A^* is

$$\binom{n}{d} 2^{-n},$$

so the total probabilty that the algorithm reaches A^* from a random assignment is at least

$$\sum_{d=0}^{n} \binom{n}{d} 2^{-n-d} = \frac{1}{2^n} \sum_{d=0}^{n} \binom{n}{d} 2^{-d} = \frac{1}{2^n}(1+\tfrac{1}{2})^n = (\tfrac{3}{4})^n.$$

Especially, in expectation, we can repeat this proces and arrive at A^* or some other satisfying assignment after $(\frac{4}{3})^n$ trails.

Time–Space tradeoffs

Time–space tradeoffs avoid redundant computation, typically "re-computation," by storing values in large tables. In particular, this inludes dynamic programming.

Dynamic programming over the subsets

Dynamic programming consists of describing the problem (or a more general form of it) recursively in an expression that involves only few varying parameters, and then compute the answer for each possible value of these parameters, using a table to avoid redundant computation. A canonical example is Knapsack.

In exponential time, the dynamic programme can consider all subsets (of vertices, for example). This is, in fact, one of the earliest applications of dynamic programming, dating back to Bellman's original work in the early 1960s.

TSP. We turn to the Traveling Salesman Problem and show how to solve it in $O(2^n n^2)$. We go back to the decrease and conquer recurrence

$$\mathrm{OPT}(T, v) = \min_{u \in T \setminus \{v\}} \mathrm{OPT}(T \setminus \{v\}, u) + w(u, v) \ .$$

The usual dynamic programming trick kicks in: The values $\mathrm{OPT}(T, v)$ are stored a table when they are computed to avoid redundant re-computation, an idea sometimes called *memoisation*. The space and time requirements are within a polynomial factor of 2^n, the number of subsets $T \subseteq V$. Figure 9 shows the first few steps.

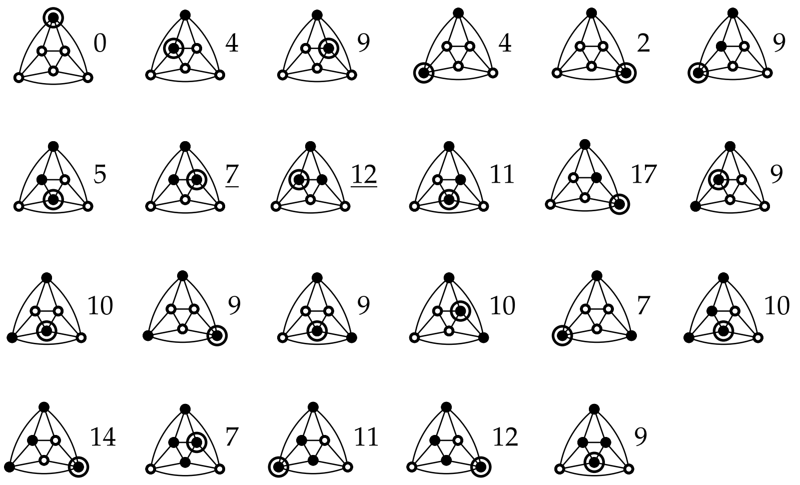


Figure 9: The first few steps of filling out a table for $\mathrm{OPT}(T, v)$ for the example graph. The starting vertex s is at the top, v is circled, and T consists of the black vertices. At this stage, the values of $\mathrm{OPT}(T, v)$ have been computed for all $|T| \leq 3$, and we just computed the value 9 at the bottom right by inspecting the two underlined cases. The "new" black vertex has been reached either via a weight 2 edge, for a total weight of $2 + 7$, or via a weight 1 edge for a total weight of $12 + 1$. The optimum value for this subproblem is 9.

It is instructive to see what happens if we start with the divide and conquer recurrence instead:

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,;$$

recall that S and T are a balanced vertex partition of U. We build a large table containing the value of $\mathrm{OPT}(X, u, v)$ for all vertex subsets $X \subseteq V$ and all pairs of vertives u, v. This table has size $2^n n^2$, and the entry corresponding to a subset X of size k can be computed by accessing 2^k other table entries corresponding to smaller sets. Thus, the total running time is within a polynomial factor of

$$\sum_{k=0}^{n} \binom{n}{k} 2^k = (2+1)^n = 3^n.$$

We observe that the benefit from memoisation is smaller compared to the decrease and conquer recurrence, which spent more time in the recursion ("dividing") and less time assembling solutions ("conquering").

Dynamic programming over a tree decomposition

The second major application of dynamic programming is over the tree decomposition of a graph. See, e.g., chapter 11 in the textbook by Kleinberg and Tardos.

Meet in the middle

Consider again the Traveling Salesman Problem. If the input graph is 4-regular (i.e., every vertex has exactly 4 neighbours), it makes sense to enumerate the different Hamiltonian paths by making one of three choices at every vertex, for a total of at most $O^*(3^n)$ paths, instead of considering the $O^*(n!)$ different permutations. Of course, the dynamic programming solution is still faster, but we can do even better using a different time–space trade-off.

We turn again to the "divide and conquer" recurrence,

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,.$$

This time we evaluate it by building a table for all choices of m, t-paths of length $l = n - \lfloor \frac{1}{2} n \rfloor$. To be precise, for each choice of $m \in V(G) - \{s\}$, and each of the 3^l paths Q from m to t, we store the length of $V(Q)$ in the entry indexed by $(m, V(Q))$ in a table (say, a dictionary). No recursion is involved, we brutally check all paths from m to t of length $|T|$, in time $O^*(3^{n/2})$. After this table is completed, we iterate over all s, m-paths of length $n - l$. For each path P, we look up the table entry at $(m, V - V(Q) \cup \{m\})$.

It is instructive to compare this idea to the dynamic programming approach. There, we used the recurrence relation at every level. Here, we use it only at the top. In particular, the meet-in-the-middle idea is qualitatively different from the concept of using memoisation to save some overlapping recursive invocations.

Exercises

A graph can be *k-coloured* if each vertex can be coloured with one of k different colours such that no edge connects vertices of the same colour.

This set of exercises asks you do solve the k-colouring problem in various ways for a graph with n vertices and m edges

Exercise 1. Using brute force, in time $O^*(k^n)$.

Exercise 2. Using a greedy algorithm, in time $O^*(n!)$.

Exercise 3. Using decrease-and-conquer, in time in time $O^*(((1 + \sqrt{5})/2)^{n+m})$. *Hint*: That's the solution to the "Finonacci" recurrence $T(s) = T(s-1) + T(s-2)$.

Exercise 4. Using divide-and-conquer, in time $O^*(9^n)$.

Exercise 5. Using Moebius inversion, in time $O^*(3^n)$. *Hint:* $\sum_{i=0}^{n} \binom{n}{i} 2^i = (2+1)^n$.

Exercise 6. Using dynamic programming over the subsets, in time $O^*(3^n)$.

Exercise 7. Using Yates's algorithm and Moebius inversion, in time $O^*(2^n)$.

Exercise 8. Using a transformation to counting triangles, count the nuber of 2-colourings in time $O^*(2^{\omega n/3})$.