A Taxonomic Introduction to Exact Algorithms

Thore Husfeldt

Lecture notes for AGAPE 2009 Spring School on Fixed Parameter and Exact Algorithms, May 25-29 2009, Lozari, Corsica (France).[1]

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 Levitin's framework. I have made no attempt to be comprehensive.

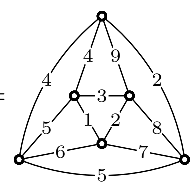
[1] Alpha release from 19 maj 2009. Errors are to be expected, and there are no references. Proceed with caution.

1

Brute force

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

TSP. For a first example, given a weighted graph like

$G =$ 

with n vertices $V = \{v_1, \ldots, v_n\}$ (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. A subset of vertices $U \subseteq V$ in an n-vertex graph $G = (V, E)$ is *independent* if 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 $O^*(2^n)$.

3-Satisfiability. 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.)

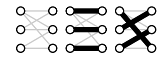


FIGURE 1. A bipartite graph and 2 of its 3 perfect matchings.

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.

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

These are all difficult problems, typically hard for 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 $O(1.732^n)$.

2

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.*

3

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.

1. 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.

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.)

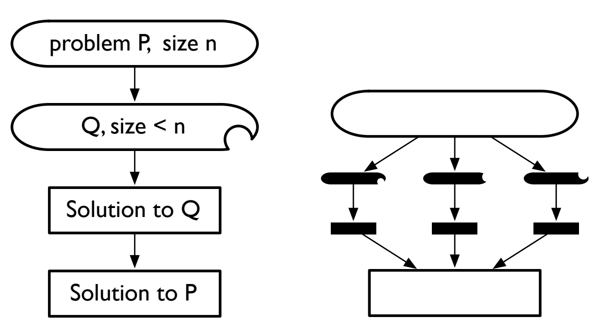


FIGURE 1. Decrease and conquer with one (left) and many (right) subproblems.

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

$$\mathrm{OPT}(T,v) = \min_{u \in T\setminus\{v\}} \mathrm{OPT}(T \setminus \{v\}, u) + w(u,v)\;. \tag{1}$$

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.

2. 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*

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

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

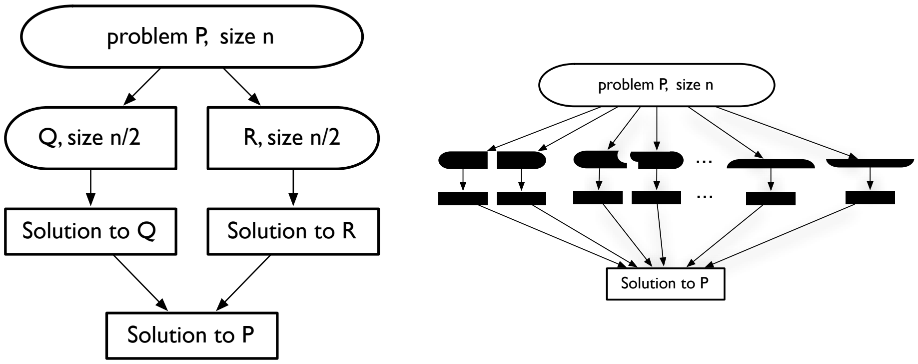


FIGURE 2. Divide and conquer with one (left) and an exponential number (right) of divisions.

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

$$\mathrm{OPT}(U,s,t) = \min_{m,S,T} \mathrm{OPT}(S,s,m) + \mathrm{OPT}(T,m,t)\,, \tag{2}$$

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, so the algorithm uses polynomial space.

4

Transformation

Transformations compute a problem by computing a different problem in its stead.

1. Moebius inversion

For a function f on subsets define

$$g(X) = \sum_{Y \subseteq X} f(Y) .$$

Then

$$f(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} g(Y) .$$

This result is called Moebius inversion, or, in a special case, the *principle of inclusion–exclusion*. For a proof see Dr. Kaski's presentation.

TSP. We'll do Hamiltonian path, because the idea stands cleaner.

For $X \subseteq V$ with $s, t \in X$, let $g(X)$ denote the number of walks of length n from s to t using only vertices from X. (A walk can use the same vertex many times, or once, or not at all.) Although it is not obvious, $g(X)$ can be computed in polynomial time for each $X \subseteq V$; the value is given in row s and column t of A^n, where A is the adjacency matrix of $G[X]$.

Now, let $f(X)$ denote the number of walks of length n from s to t that use exactly the vertices from X. In particular, $f(V)$ is the number of Hamiltonian paths from s to t. By Moebius inversion,

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y) .$$

so the number of Hamiltonian paths can be counted in time $O^*(2^n)$ and polynomial space.

To make this work for TSP, we need to handle every total distance separately, details omitted.

Perfect matchings. For $Y \subseteq V$ Let $f(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using exactly the vertices in Y (all of them); the number of perfect matchings is then given by $f(V)$. For a moment, let $g(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using only the vertices in Y (but not necessarily all of them). Then, by Moebius inversion

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y) .$$

Since $g(Y)$ is easy to compute for given Y, we can count the number of perfect matchings in time $O^*(2^n)$.

For bipartite graphs, we can do slightly better. So, let $V = L \cup R$ with $|L| = |R| = \frac{1}{2}n$ and assume all edges have an endpoint in L and an enpoint in R. Now, for $Y \subseteq R$ (rather than $Y \subseteq V$), let $g(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using *all* the vertices in L and *some* of the vertices in Y, and let $f(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using *all* the vertices in L and *all* the vertices in Y. Then

$$f(V) = \sum_{Y \subseteq R} (-1)^{|R \setminus Y|} g(Y)\,,$$

in particular, the sum has only $2^{n/2}$ terms. For each $Y \subseteq R$, the value $g(Y)$ is easy enough to compute: if vertex $v_i \in L$ has d_i neighbours in R then

$$g(Y) = d_1 \cdots d_{n/2}\,.$$

Thus, the total running time is $O^*(2^{n/2}) = O(1.732^n)$. See fig. 1 for an example.

In fact, this is a famous results in combinatorics, the *Ryser formula* for the permanent, often expressed in terms of a 01-matrix A of dimension $k \times k$ as

$$\text{per}\, A = \sum_{Y \subseteq \{1,\ldots,k\}} (-1)^{k-|Y|} \prod_{i=1}^{k} \sum_{j \in Y} A_{ij}. \tag{3}$$

The connection is that A is the upper right quarter of the adjacency matrix of G.

2. Finding triangles

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

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

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 sixfold 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 $O(d^\omega)$, $\omega < 3$ (see Dr. Kaski's presentation).

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 $\binom{n}{k}$.

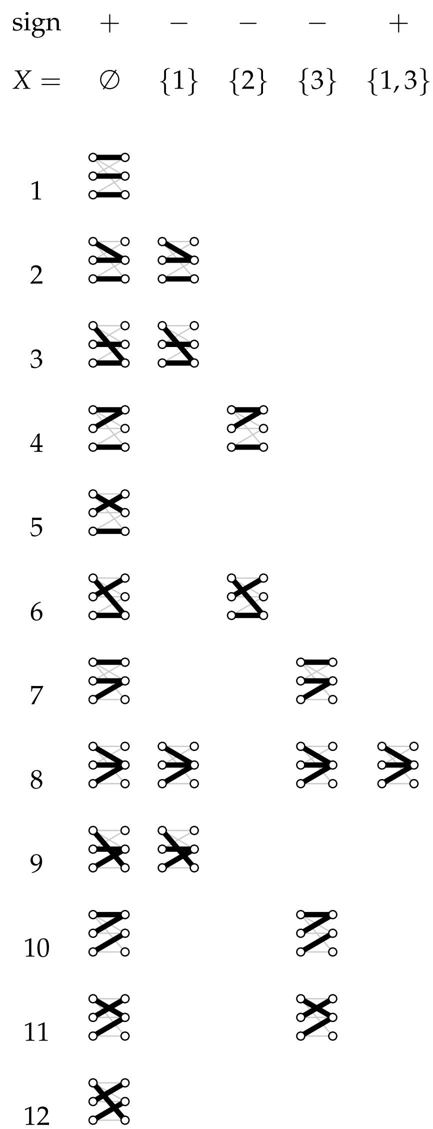


FIGURE 1. The input graph has three perfect matchings, in columns 1, 5, and 12. The first row shows all $12 = 3 \cdot 2 \cdot 2$ ways to map the left vertices to the right. Every row of the table shows the mappings that avoid various vertex subsets X, drawn as $\circ$. We omit the rows whose contribution is zero, like $X = \{1,2\}$, $X = \{2,3\}$ and $X = \{1,2,3\}$. Of particular interest is column 8, which is subtracted twice and later added again. The entire calculation is $12 - 4 - 2 - 4 + 1 + 0 + 0 - 0$, with is indeed 3.

Perfect matchings. The next example 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 $O^*(n^{\omega k/3}) = (1.732^n)$.

5

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.

1. Local search

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.

6

Time–Space tradeoffs

Time–space tradeoffs avoid redundant computation, typically "recomputation," by storing values in large tables.

1. 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'll solve TSP in $O(2^n n^2)$, a bound that is still the best known. 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 recomputation, 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 2 shows the first few steps.

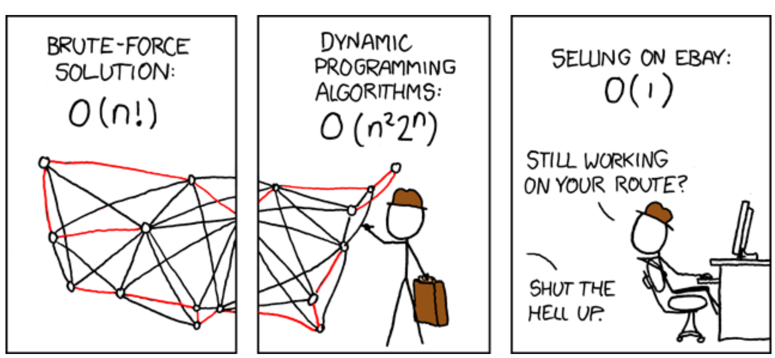


FIGURE 1. The dynamic programming algorithm for TSP mentioned in *xkcd* nr. 399.

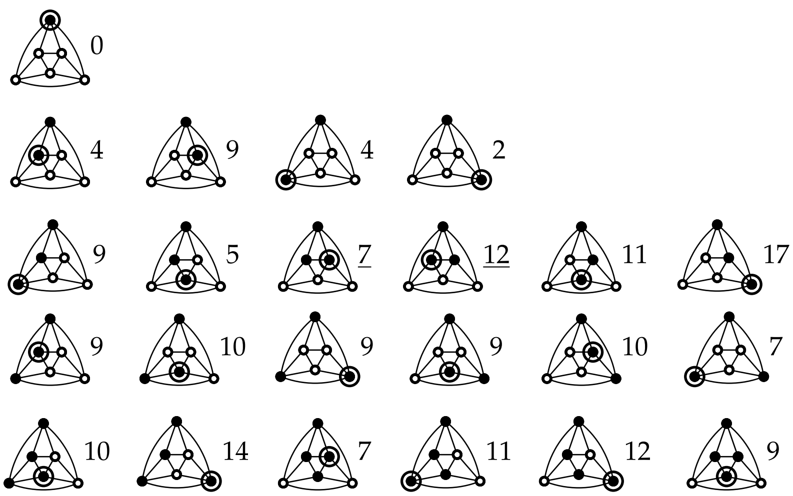


FIGURE 2. The first few steps of filling out a table for $\mathrm{OPT}(T, v)$ for the example graph. The starting vertex s is at the top, v is circled, and T consists of the black vertices. At this stage, the values of $\mathrm{OPT}(T, v)$ have been computed for all $|T| \leq 3$, and we just computed 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").

2. Dynamic programming over a tree decomposition

The second major application of dynamic programming is over the tree decomposition of a graph. We don't cover that here, for lack of time.

3. Meet in the middle

TSP. 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 and $T \ni t$ with $|T| = n - \lfloor \frac{1}{2}n \rfloor$. 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 choices of $S \ni s$ with $|S| = \lfloor \frac{1}{2}n \rfloor + 1$ the same way, using $3^{n/2}$ iterations. For each S and m we check our dictionary for the entry corresponding to m and $V - T$.

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.

4. Fast Transforms

→ Dr. Kaski

5. Fast Matrix multiplication

→ Dr. Kaski

7

Exercises

An 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 $O^*(n^k)$.

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 $O^*(2^n)$.