EDAN55, supplementary notes for fixed parameter tractability

October 24, 2012

° rev. fa5a008

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*.

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 $(v_1, \dots, v_k)$ such that $v_i v_{i+1} \in E$ for $1 \le i < k$.

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 $v_1, \dots, v_i$ of neighbouring vertices in G (not necessarily simple), then

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

where $\cdot$ denotes concatenation.[1] This produces all sequences of distinct neighbouring vertices of length k in G, their number is $n(n-1)\cdots(n-k-1)$, 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 $O(n^k k)$.

[1] Maybe give as pseudocode instead.

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.

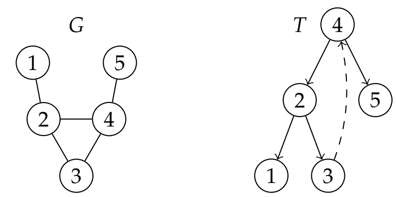

Figure 1: The Bull graph G and a DFS tree T of G starting at $r = 4$.

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.

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

Node coverage. Node coverage is easily established in this tree decomposition, 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.

However, there is a further simplification of the tree decomposition that we want to perform before moving on. 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 2.

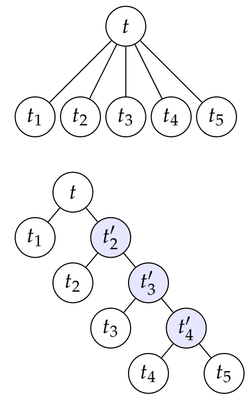

Figure 2: Transformation of a node $t \in T$ with more than 2 children.

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 rootet 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 $\sum_{r=0}^{k} \binom{k}{r} r! \leq k! 2^k$.

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 $f(\overline{w}) = r$, 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

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

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.

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*.[4]
2. Find a *rainbow coloured* k-path, i.e., a k-path on which every colour appears. (And therefore appears exactly once.)

[4] This is not a proper colouring in the sense of graph 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.[5] The remarkable thing is that $\Pr(R)$ is merely exponential in k, instead of $1/k!$.

[5] $r! \geq \sqrt{2\pi r}\left(\frac{r}{e}\right)^r$

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

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

in FPT time

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