Treewidth

2017-08-12, rev. 9e50395

Implement an algorithm for independent set using dynamic programming over a (given) tree-decomposition.

2017 is the second time we try this exercise. Not all problems from last year are resolved.

The algorithm

The algorithm takes as input an unweighted, undirected graph G and a tree-decomposition T of G with width w. A detailed explanation of the algorithm can be found in *Tree Decompositions of Graphs*, Section 10.4 of Kleinberg and Tardos, *Algorithms Design*, Addison–Wesley 2005.

The input files come in pairs with extension `.td` and `.gr`, respectively. The format is described in `data/README.md`.

You need to achieve a running time of $\exp(O(w))\operatorname{poly}(n)$; a straightforward implementation of the pseudo-code in the book will achieve that, as analysed at the end of Section 10.4.

My thoughts about implementation. I parse both G and T as graphs. Note that the input representation of T is just an undirected, connected graph without cycles, so we can pick any node as the root r of the tree decomposition. From r, I perform a (very simple) graph traversal that allows me to associate with each node t of T the list of its children (in T) and a topological ordering. I ended up associating the following information with each node t in the tree-decomposition:

1. A list of its children.

2. The piece V_t (sometimes called 'bag' in the literature), as a set of vertices from G.

3. A table of 2^{w+1} values $f_t(U)$, for each $U \subseteq V_t$. Initially, these values are undefined. They get filled in by the dynamic programming algorithm.

The graphs called `web`k ($k \in \{1, \ldots, 4\}$) and `eppstein` are meant to be useful for initial debugging.

A lot of my attention was spent on handling sets. (We need to iterate over subsets, take set intersections, and test for set equality.) I can see two approaches for this.

1. At node t, rename the vertex names so as to identify V_t with $\{0, \ldots, w\}$ and store each subset $U \subseteq \{0, \ldots, w\}$ as a bit string

$b_0 \cdots b_w$ where $b_i = 1$ if and only if $i \in U$. If you choose this implementation, you are allowed to assume that w is never larger than the word length on your machine. Thus, such a representation can be stored in a single machine word. The set operations now become (hairy but compact) bit fiddling operations. This solution is very fast, and a low level language like C works extremely well for it. Table lookup is just array access, and iteration over subsets is (careful) incrementation. The difficulty here is to keep a cool head about which vertex in G (or in V_{t_i}, for that matter) corresponds to which vertex in V_t.

2. You use (or write) a data type for sets. For table look-up you can use an associative array (for instance, by making the data structure hashable). This is a lot slower and requires much more code, but the result is slightly more readable, in particular in a high-level language with neat syntax. A good suggestion is to use the programming language Scala, which combines good abstractions with reasonable running times.

The output of your program is just a number (the size of the maximum independent set). But you are strongly advised to actually compute the elements of a maximum independent set as well. (By adding the relevant information to $f_t(U)$ when you traverse the tree decomposition.) Otherwise your code will be very difficult to debug.

Treewidth report

by Alice Cooper and Bob Marley[1]

[1] Complete the report by filling in your names and the parts marked [...]. Remove the sidenotes in your final hand-in.

Results

The following table gives the indpendence number $\alpha(G)$ (the size of a maximum independent set) for each graph:

Instance name	n	w	$\alpha(G)$
web4	5	2	3
WorldMap	166	5	78
FibonacciTree_10	143	1	72
StarGraph_100	101	1	100
TutteGraph	46	5	19
DorogovtsevGoltsevMendesGraph	3282	2	2187
HanoiTowerGraph_4_3	64	13	16
TaylorTwographDescendantSRG_3	...		
CirculantGraph_20_5	...		
AhrensSzekeresGeneralizedQuadrangleGraph_3	...		
DesarguesGraph	...		
FranklinGraph	...		
FolkmanGraph	...		
GoldnerHararyGraph	...		
FriendshipGraph_10	...		
HerschelGraph	...		
HoltGraph	...		
Klein7RegularGraph	...		
McGeeGraph	...		
TaylorTwographSRG_3	...		
WellsGraph	...		
SierpinskiGasketGraph_3	...		
...			

Our implementation

We implemented sets as $\cdots$. The largest n and w for which this implementation worked in 60 seconds on our machine was $n = [\cdots]$ and $w = \cdots$ (the graph called $\cdots$), or $n = \cdots$ and $w = \cdots$ (the graph called $\cdots$).