TLDC: Too Long Didn't Count
Entry for the International Competition on Graph Counting Algorithms

Rafael Kiesel
rafael.kiesel@web.de

Markus Hecher
Massachusetts Institute of Technology
United States
hecher@mit.edu

Abstract

We present Too Long Didn't Count (TLDC), a solver for the length limited path counting problem in directed graphs. Our solver is a portfolio of multiple approaches that are chosen based on graph parameters that we can exploit. Common techniques are caching of subproblems, propagation of forced choices, and pruning based length of the current partial solution. The different approaches exploit a subset of pathwidth, treewidth, high "automorphicity", and the difference of the longest and shortest possible path.

1 Introduction

The length limited path counting problem is the following:

Given: A graph $G = (V, E)$, with vertices V and undirected edges $E \subseteq V \times V$, and a maximum length $l \geq 0$.
Compute: The number of simple paths in G whose length is less or equal to l.

Here, a *path* is a list $p = (v_0, \ldots, v_n)$ of vertices $v_i \in V$ and $n > 0$, such that for all $0 \leq i < n$ it holds that $\{v_i, v_{i+1} \in E$. Such a path is *simple* if $\{v_0, \ldots, v_n\}$, i.e., there are no repeated vertices, and has length n.

For the purposes of this paper and our solver, two paths $(v_0, \ldots, v_n), (w_0, \ldots, w_n)$ are equal iff either $v_i = w_i$, for all $0 \leq i \leq n$ or $v_i = w_{n-i}$, for all $0 \leq i \leq n$. That is, it does not matter if we go "forward" or "backward".

We consider two variants of the problem. AllPairs and OnePair, where the number of all paths is requested and the number of paths between two vertices t_1, t_2 is requested, respectively. Both problems are #P-complete [6].[1] Nevertheless, they are relevant for problems such as network reliability [5].

In this extended abstract, we shortly describe the approaches used in TLDC to solve both the AllPairs and OnePair problem. We first discuss common techniques, before we give the main ideas of the separate approaches.

2 Common Techniques

2.1 Caching

We use caching to reuse the solutions of subproblems we previously encountered. The keys for the cache differ, however, the value is usually the same, namely a result vector, that represents

[1] Not directly stated in the cited work but follows trivially.

for all possible lengths the number of solutions found.

2.2 Propagation

In all approaches we are faced with a choice at each step. For each of these choices, we generate the result of making that choice (for example by taking an edge or skipping it). For each of these results we then check how many choices they will have in the next step. If there are zero, we cannot continue with this entry. If there are many, we proceed with caching for this entry. However, if there is exactly one choice, then it is forced. In this case, we make the choice and start again.

2.3 Distance Pruning

For complicated graphs, it is hopeless to compute the number of all paths and only count those of a certain length. However, if the maximum length l is low, we can perform pruning based on the distance of the partial path we already have and how many steps we at least need to take in order to arrive at a solution.

3 Approaches

Broadly speaking, we have two main approaches.

3.1 Backtracking Search

Firstly, we use "backtracking search", where we exhaustively perform depth first search with backtracking. This solution is only used in the OnePair case.[2] There, given t_1, t_2, we choose one of the neighbors v of t_1, mark t_1 as visited, and proceed with new terminals v, t_2.

Here, it is performance critial, to apply distance pruning to mark vertices that cannot be on a path anymore as visited. Furthermore, it is important to process the subproblems in order of the size of the unvisited graph. I.e., we first generate for each entry $\{t_1, t_2\}$ for the remaing graph size n the new entries $\{v, t_2\}$, which have remaining graph size $< n$. Only then, we consider remaining graph size $n - 1$.

4 Frontier-based Search

Frontier-based search is a well-known standard approach [2, 3] in this line of research. We refer the interested reader to the standard literature for the basics.

Important aspects of frontier-based search are

- the order in which edges are eliminated, and
- whether we allow decomposition

The order of the edges has a strong influence on the number of cache entries. We try using htd [1] to generate an order from a tree decomposition and compare with the possibility of eliminating edges in order of their (integer) labels.

Decomposition, means we split subproblems into two independent ones and merge the solutions. While this can be beneficial if it reduces the number of cache entries significantly, it can also have detrimental effects, since merging the solutions of two independent components requires considering each pair of solutions for the different components.

[2] However, one could in principle of course also solve the AllPairs case by solving all OnePair instances for the graph.

5 Caching Modulo Automorphisms

Note, that we are in principle not required to store the exact subproblem in the cache. As long as two subproblems have the same number of solutions, we can use them interchangeably.

Clearly, for given two subproblems we cannot easily decide whether they have the same number of solutions. However, we can use NAUTY [4] to generate a canonical representation[3] of the remaining problem. If two subproblems have the same canonical representation, then they have the same number of solutions. If the given graph has many automorphisms this can greatly reduce the number of cache entries.

We implemented this technique for backtracking search as well as frontier-based search without decomposition. For frontier-based search with decomposition we are unaware (but do not exclude) the possiblity to use this approach.

References

[1] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd–a free, open-source framework for (customized) tree decompositions and beyond. In *Integration of AI and OR Techniques in Constraint Programming: 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings 14*, pages 376–386. Springer, 2017.

[2] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(9): 1773–1784, 2017.

[3] Richard E Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier search. *Journal of the ACM (JACM)*, 52(5):715–748, 2005.

[4] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014.

[5] J Scott Provan and Michael O Ball. Computing network reliability in time polynomial in the number of cuts. *Operations research*, 32(3):516–526, 1984.

[6] Leslie G Valiant. The complexity of enumeration and reliability problems. *siam Journal on Computing*, 8(3):410–421, 1979.

[3] modulo graph automorphism