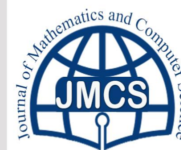
Available online at www.isr-publications.com/jmcs
J. Math. Computer Sci., 20 (2020), 349–355
Research Article

Online: ISSN 2008-949X
Journal of Mathematics and Computer Science
Journal Homepage: www.isr-publications.com/jmcs



A space-efficient algorithm for computing the minimum cycle mean in a directed graph

Paweł Pilarczyk

Gdańsk University of Technology, ul. Narutowicza 11/12, 80-233 Gdańsk, Poland.

Abstract

An algorithm is introduced for computing the minimum cycle mean in a strongly connected directed graph with n vertices and m arcs that requires $O(n)$ working space. This is a considerable improvement for sparse graphs in comparison to the classical algorithms that require $O(n^2)$ working space. The time complexity of the algorithm is still $O(nm)$. An implementation in C++ is made publicly available at http://www.pawelpilarczyk.com/cymealg/.

Keywords: Directed graph, weighted graph, sparse graph, algorithm, minimum cycle mean, mean cycle weight, rigorous numerics, floating-point arithmetic, rounding.

2010 MSC: 05C20, 05C22, 05C38.

©2020 All rights reserved.

1. Introduction

In the design of algorithms for finding the minimum or maximum mean weight of a cycle in a weighted directed graph, attention has been paid to the performance in terms of execution time (see, e.g., [3, 4, 8, 9]), but reducing the considerable storage requirements of all these approaches has not yet been seriously taken into consideration. The best algorithms currently known for solving this problem (see, e.g., [1, 2, 6]) are based on the original work by Karp [9], in which for each strongly connected component the weights of path progressions of lengths from 1 to n from a selected source vertex to any of the n vertices in the graph need to be computed by considering all the m arcs of the graph at each of n rounds; see Section 2 for the details. These weights need to be tabulated and thus occupy $O(n^2)$ space in addition to $O(n+m)$ space for storing the graph, which makes the entire algorithm run in $O(n^2+m)$ space and $O(nm)$ time. The improved algorithms do not get below these orders of complexity in the pessimistic case, but provide early termination criteria (e.g., [8]) or reduce the number of operations that are actually executed (e.g., [3]). Although these improvements indeed result in better performance, the need to tabulate the weights of path progressions is still there.

In some applications one might need to apply this algorithm to huge graphs, where the memory usage is a much more serious constraint than the running time. For example, in [5], the problem of

Email address: pawel.pilarczyk@pg.edu.pl (Paweł Pilarczyk)

doi: 10.22436/jmcs.020.04.08

Received: 2019-08-24 Revised: 2020-01-23 Accepted: 2020-02-01

computing a lower bound for the expansion exponent in one-dimensional dynamics was reduced to computing the minimum cycle mean in a directed graph that represents the dynamical system on a collection of intervals into which the domain of the map was subdivided. The larger the number of these intervals, corresponding to vertices, the better the estimate. In the specific implementation used in [5], the total system memory of 256 GB was exhausted when processing graphs with somewhat over 125,000 vertices, while the computation time did not exceed 24 hours, thus encouraging to try processing even larger graphs in order to achieve better estimates of the expansion. The improved algorithm, introduced in Section 3 below, considerable decreased memory usage and allowed for the extensive computations described in [7].

Decreasing the usage of temporary storage down from $O(n^2)$, which is required in the original algorithm, makes sense in the case of a sparse graph, in which the number m of arcs is below the maximum $O(n^2)$, or otherwise the memory needed to store the graph alone is already of the order of $O(n^2)$. In the most optimistic situation, m may be of the order of $O(n)$, which is actually the case in the discussed example, because the number of outgoing edges is bound by a constant that roughly corresponds to the Lipschitz constant of the map.

The main idea to decrease the order of memory usage is to get rid of the need for tabulating the weights of path progressions of all the lengths up to n, and to use a two-pass approach, as suggested in [4]. Indeed, it is possible to conduct the computations in such a way that only a fixed number of path progressions is stored for each of the n vertices of the graph, and thus the memory requirement drops from $O(n^2)$ down to $O(n)$, in addition to storing the graph itself. The latter can be done in $O(n+m)$ space, but in some applications, the edges might be given by a specific formula, and thus need not be actually stored; instead, they can be quickly calculated whenever needed, which reduces the overall memory usage to $O(n)$ in that case. The time complexity remains at the order of $O(nm)$.

2. Background

The classical result [9] is the following. Let λ^* denote the minimum mean weight of a cycle in a weighted directed graph $G = (V, E, w)$, where V is the set of n vertices of G, E is the set of m arcs (directed edges) in G, and $w\colon E \to \mathbb{R}$ is a function that defines the weight of each arc. Without loss of generality, one can assume that G is strongly connected, or otherwise one can decompose G into strongly connected components in $O(n+m)$ time and space. Choose an arbitrary vertex $v_0 \in V$. For each $v \in V$, let $F_k(v)$ denote the minimum weight of any path of length k whose starting vertex is v_0 and ending vertex equals v, or set $F_k(v) := \infty$ if no such path exists, for $k = 0, \ldots, n-1$.

Theorem 2.1 ([9, Theorem 1])**.** *Let $G = (V, E, w)$ be a strongly connected weighted directed graph with $n := |V|$ vertices and $m := |E|$ arcs. Let λ^* denote the minimum mean weight of a cycle in G. Then*

$$\lambda^* = \min_{v\in V} \max_{k<n} \frac{F_n(v) - F_k(v)}{n-k}. \tag{2.1}$$

As a consequence, λ^* can be computed in $O(n^2)$ time, once the values $F_k(v)$ have been tabulated. The functions F_i can be computed using the recursive formula

$$F_k(v) = \min_{(u,v)\in E} \big(F_{k-1}(u) + w(u,v)\big)$$

for $k = 1, \ldots, n$, with the initial condition $F_0(v) = \infty$ for $v \in V \setminus \{v_0\}$, and $F_0(v_0) = 0$, which takes $O(nm)$ time. Since G is strongly connected, $m \geqslant n-1$, and thus the overall time complexity of this algorithm is $O(nm)$. The amount of working space is $O(n^2)$, needed for tabulating all the values of $F_k(v)$. Keeping the graph in memory requires at least $O(n+m)$ storage in general, thus making the overall space complexity of $O(n^2+m)$.

3. Main result

The following algorithm shows that in the case of sparse graphs, where $m \ll n^2$, the computation of the minimum cycle mean can be actually done better than using Karp's original algorithm. The difference is especially important when m is of the (optimal) order of $O(n)$, which actually appears in the application to computational dynamics discussed in [5].

In the description of the algorithm, a convention is used that the minimum or maximum of an undefined value and a defined value is the latter. Moreover, a data structure for representing a mapping (such as α_k and γ) contains its finite domain (denoted dom) and the values assigned to the elements of the domain, which can be re-defined by making a new assignment; moreover, the domain of a mapping can be dynamically changed by defining additional assignments.

The algorithm consists of two passes, indexed by the value of p. For $p = 0$, the weights of path progressions of length n are computed, and for $p = 1$, the maximum possible quotients that appear in (2.1) for each vertex are determined.

Algorithm 3.1.
Input:
  $G = (V, E, w)$—a strongly connected weighted directed graph;
begin
  $n := |V|$; $\gamma := \emptyset$; take any $v_0 \in V$;
  for $p := 0$ **to** 1 **do**
    $\alpha_0 := \emptyset$; $\alpha_0(v_0) := 0$;
    for $k := 1$ **to** $n - p$ **do**
      $\alpha_k := \emptyset$;
      for all $v \in \operatorname{dom} \alpha_{k-1}$ **do**
        for all v' **such that** $(v, v') \in E$ **do**
          $\alpha_k(v') := \min\{\alpha_k(v'), \alpha_{k-1}(v) + w(v, v')\}$;
          if $p = 1$ **then**
            $\gamma(v') := \max\{\gamma(v'), (\alpha_n(v') - \beta_k(v'))/(n - k)\}$;
      forget α_{k-1};
  return $\min\{\gamma(v) : v \in V\}$;
end.

Remark 3.2. If the arithmetic operations on the weights cannot be done precisely, e.g., because of using floating-point representation of numbers, in order to compute an actual lower bound for the minimum cycle mean in G, one must round downwards the results of computing $\alpha_k(v')$ in the first pass (when $p = 0$) and $\gamma(v')$ in the second pass, and round upwards the results of computing $\alpha_k(v')$ in the second pass. The interested reader is referred to [10] for a comprehensive introduction to rigorous numerics, rounding issues, and interval arithmetic.

Proposition 3.3. *Given a strongly connected weighted directed graph G with n vertices and m arcs, Algorithm 3.1 computes the minimum mean weight of a cycle in G in $O(n)$ working space. The time complexity of the algorithm is $O(nm)$.*

Before proving Proposition 3.3, it is worth to note that since one can decompose an arbitrary directed graph G into strongly connected components in $O(n + m)$ time and space, the following result holds true.

Theorem 3.4. *The minimum mean cycle weight in an arbitrary weighted directed graph G with n vertices and m arcs can be computed in $O(n + m)$ space and $O(nm)$ time.*

Proof of Proposition 3.3. Since it is obvious that the functions α_k computed in the algorithm are the same as F_k in Karp's original algorithm (see Section 2), the returned value is the same as the one given by (2.1), and thus it follows from Theorem 2.1 that the algorithm indeed returns the minimum cycle mean of G.

In order to achieve the claimed space and time complexity, suitable data structures must be used. Namely, since no more than two mappings α_k, for $k = 0, \dots, n$, are stored at a time, in addition to α_n which is not deleted after has been computed, in order to show that the amount of additional storage needed by the algorithm is $O(n)$, it is enough to show that both α_k and γ can be stored in $O(n)$ space each. Moreover, if the collection of arcs leaving any vertex v of G can be determined in $O(1)$ time independent of v, the weight of each arc can be computed in $O(1)$ time, the values of $\alpha_k(v)$ and $\gamma(v)$ can be determined in $O(1)$ time independent of v (and it can be determined in $O(1)$ whether $\alpha_k(v)$ or $\gamma(v)$ are still undefined), and the initialization $\alpha_k := \emptyset$ can be done in $O(n)$ time, then the claimed time complexity of the algorithm can be immediately proved by estimating the number of times the loops are executed. Appropriate data structures are discussed in the remainder of the proof.

The graph can be represented in the following way. The vertices are identified by consecutive integers, that is, $V = \{0, \dots, n-1\}$. The arcs leaving consecutive vertices are stored in a single array ϑ of size m which only keeps the targets of the arcs, and an additional array η of length n stores the offsets immediately past the ends of the groups of arcs starting from each vertex. More precisely, assuming that $\eta_{-1} = 0$, the targets of all the arcs emanating from a vertex j are stored in the array ϑ at the positions $\eta_{j-1}, \dots, \eta_j - 1$. The weights of the arcs can be stored in an array of size m. Using this implementation of the directed graph, one can determine the first arc emanating from a vertex v in $O(1)$ time by looking into the array ϑ, and take each next vertex also in $O(1)$ time. Determining the weight of each arc can also be done in $O(1)$ time.

An appropriate implementation of a function f defined on a finite subset of $V = \{0, \dots, n-1\}$, to be used for α_k and γ, is a little more subtle. Let κ be the cardinality of the domain of f. Let A be an array of n integers in the range $\{0, \dots, n-1\}$ such that the first κ elements of the array contain the elements of the domain of f in an arbitrary order. Let B be an array of n integers in the range $\{0, \dots, n\}$, where the j-th number is set to n if $j \notin \operatorname{dom} f$, and equals the index of the element j in A otherwise. Let F be an array of size n such that its first κ entries correspond to the values of the function f on the corresponding elements of its domain in the order they are listed in A. Note that this representation of f uses $O(n)$ space. Creating a new representation of $f = \emptyset$ takes $O(n)$ time, because all the n entries of B must be set to n. Checking if $f(j)$ is defined takes $O(1)$ time, because this information can be determined by the value of B at position j. Recording the assignment $f(j) := \omega$ can be done in $O(1)$ time: One first checks the j-th entry in B; if it equals n then the element j is not in Q, and thus one must set the following: $A_\kappa := j$, $B_j := \kappa$, $F_\kappa := \omega$, and then increase κ by 1. Otherwise, the j-th entry in B contains the index of F that needs to be updated to store the new value ω. □

Remark 3.5. Clearing the data structure that stores a previous α_k for re-use in the assignment $\alpha_k := \emptyset$ can be done in the time $O(\kappa)$ instead of $O(n)$, where κ is the size of the domain of the map previously stored in that data structure. Namely, while $\kappa \geqslant 0$, one can do the following: Let j be the entry of A at position $\kappa - 1$; write n in B at position j; decrease κ by 1. Although this does not provide a better order of the time complexity, using this faster way to reset the mapping data structure may save considerable amount of time for the initial values of k, when the number of vertices reachable from the selected source v_0 is still small. This idea was used in [3] as part of the strategy to speed up the computations.

4. Tests and experiments

In order to confirm the usefulness of Algorithm 3.1 and to compare its demand for resources with Karp's original algorithm [9] in a practical application, both algorithms were tested on a collection of graphs that arise naturally in the computation of expansivity estimates in a dynamical system, studied in [5, 7]. Graphs in a wide range of sizes were considered, with the number of vertices ranging from 1,000 to 16,000. Due to the specific application, the number of edges going out from every vertex was very small, and its average was 3 in all the examples, thus ensuring that each graph was sparse indeed. Computations were conducted with approximate double-precision floating-point arithmetic, and also

with rounding control that provides rigorous results. The software was running on a personal computer with Intel® Core™2 Duo CPU E8400 at 3GHz, with 4GB RAM installed on the system, and under the control of GNU/Linux Debian 4.19.67-2+deb10u2 (2019-11-11) x86_64. The measurements were made by the popular utility program "GNU time."

Table 1: Comparison of memory usage and processing time for a collection of graphs of various sizes. The results of running Karp's original algorithm are gathered at the left-hand side, and the results of applying Algorithm 3.1 are shown at the right-hand side. In both cases, non-rigorous double-precision floating-point arithmetic was used.

Karp's original algorithm			Algorithm 3.1		
Number of vertices	Memory usage [MB]	Processing time [s]	Number of vertices	Memory usage [MB]	Processing time [s]
1000	11.35	0.1	1000	3.65	0.1
2000	34.77	0.3	2000	3.80	0.6
3000	74.33	0.8	3000	3.73	1.5
4000	129.16	1.3	4000	4.05	2.7
5000	199.48	2.1	5000	4.12	4.2
6000	286.02	3.1	6000	4.11	6.0
7000	386.92	4.2	7000	4.02	8.0
8000	504.40	5.5	8000	4.11	10.4
9000	636.58	7.8	9000	4.32	13.4
10000	784.80	8.7	10000	4.63	16.7
11000	948.21	10.9	11000	4.38	20.5
12000	1126.96	13.2	12000	4.68	24.8
13000	1318.41	18.7	13000	4.82	29.7
14000	1531.88	26.9	14000	5.12	33.4
15000	1755.44	29.7	15000	5.28	38.0
16000	1995.78	34.6	16000	5.31	44.1

Table 2: Comparison of memory usage and processing time for a collection of graphs of various sizes. The results of running Karp's original algorithm are gathered at the left-hand side, and the results of applying Algorithm 3.1 are shown at the right-hand side. In both cases, double-precision floating-point arithmetic with controlled rounding was used to compute upper bounds and lower bounds on the results of all operations, thus providing a mathematically rigorous (validated) result.

Karp's original algorithm (rigorous)			Algorithm 3.1 (rigorous)		
Number of vertices	Memory usage [MB]	Processing time [s]	Number of vertices	Memory usage [MB]	Processing time [s]
1000	19.08	0.6	1000	3.75	0.8
2000	66.10	2.5	2000	3.79	3.0
3000	143.98	5.7	3000	3.88	7.1
4000	253.02	10.2	4000	3.94	13.0
5000	392.30	15.9	5000	4.04	19.6
6000	564.17	23.0	6000	4.18	27.9
7000	764.83	31.2	7000	4.25	36.9
8000	997.29	40.8	8000	4.30	48.0
9000	1259.38	52.0	9000	4.25	61.9
10000	1553.45	64.6	10000	4.62	77.4
11000	1877.80	79.0	11000	4.38	94.6
12000	2232.80	95.6	12000	4.86	112.8
13000	2613.04	117.3	13000	4.86	133.6
14000	3036.34	150.1	14000	5.12	157.0
15000	3479.57	178.7	15000	5.28	177.0
16000	3768.88	213.3	16000	5.21	206.8

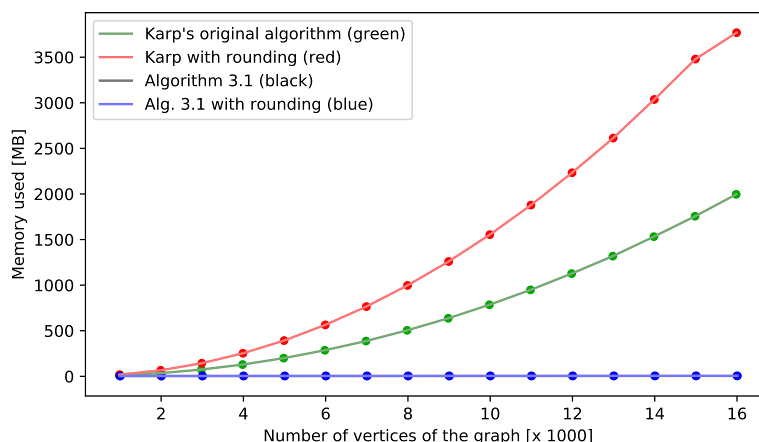


Figure 1: The amount of memory used by Karp's algorithm and the new Algorithm 3.1, both in the case of approximate and rigorous numerics. See Tables 1 and 2 for the exact data.

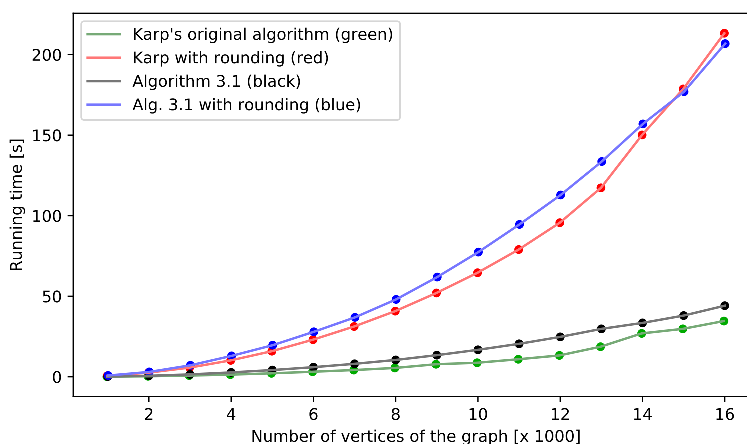


Figure 2: The running time (single CPU) of Karp's algorithm and the new Algorithm 3.1, both in the case of approximate and rigorous numerics. See Tables 1 and 2 for the exact data.

The results of the tests are gathered in Tables 1 and 2, and also illustrated in Figure 1 and in Figure 2. One can immediately notice the profound difference in the memory usage, which is the major improvement provided by Algorithm 3.1 over Karp's original algorithm. Indeed, while the amount of 4GB RAM available in the system didn't allow running Karp's algorithm on larger graphs (with rigorous numerics, see Table 2), the memory usage reported in the case of Algorithm 3.1 was negligible (see Figure 1), and is mainly due to the size of the graph that is being processed.

As it was emphasized in Section 3, the weights of path progressions are computed twice in Algorithm 3.1, and this fact is indeed reflected in the actual performance. For example, the graph with 10,000 vertices was processed by Karp's algorithm in 8.7 s, while Algorithm 3.1 needed almost twice as much time. However, when rigorous numerics is taken into account (as discussed in Remark 3.2), the weights of path progressions need to be computed twice also in Karp's algorithm (in order to obtain the lower and upper bounds). This diminishes the advantage of Karp's algorithm over Algorithm 3.1, which is confirmed in the results, especially clearly in Figure 2. Moreover, it is interesting to see that for small graphs Karp's algorithm performs slightly faster, but at some points Algorithm 3.1 actually outperforms it. The most likely reason for this might be the overhead related to memory management.

The C++ implementation of Algorithm 3.1 used for the experiments, as well as the sample graphs discussed in this section, have been made publicly available at [11] as a reference and for testing purposes.

Acknowledgment

The author expresses his gratitude to Stefano Luzzatto for his encouragement to do extensive computations of expansion exponents in the quadratic map family for the paper [7], which motivated the development of the improved minimum cycle mean algorithm.

The research leading to these results has received funding from the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no. 622033.

References

[1] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, (1993). 1

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, MIT Press, Cambridge, (2001). 1

[3] A. Dasdan, R. K. Gupta, *Faster maximum and minimum mean cycle alogrithms for system performance analysis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, **17** (1998), 889–899. 1, 3.5

[4] A. Dasdan, S. S. Irani, R. K. Gupta, *Efficient algorithms for optimum cycle mean and optimum cost to time ration problems*, Proc. 36th Design Automation Conf. (DAC), **1999** (1999), 37–42. 1

[5] S. Day, H. Kokubu, S. Luzzatto, K. Mischaikow, H. Oka, P. Pilarczyk, *Quantitative hyperbolicity estimates in one-dimensional dynamics*, Nonlinearity, **21** (2008), 1967–1987. 1, 3, 4

[6] Egerváry Research Group on Combinatorial Optimization (EGRES), *Minimum Mean Cycle Algorithms*, Library for Efficient Modeling and Optimization in Networks (LEMON) 1.2.3 Documentation, (2020). 1

[7] A. Golmakani, S. Luzzatto, P. Pilarczyk, *Uniform expansivity outside a critical neighborhood in the quadratic family*, Exp. Math., **25** (2016), 116–124. 1, 4, 4

[8] M. Hartmann, J. B. Orlin, *Finding minimum cost to time ratio cycles with small integral transit times*, Networks, **23** (1993), 567–574. 1

[9] R. M. Karp, *A characterization of the minimum cycle mean in a digraph*, Discrete Math., **23** (1978), 309–311. 1, 2, 2.1, 4

[10] R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, (1966). 3.2

[11] P. Pilarczyk, *Minimum Cycle Mean Algorithms for Directed Graphs: A C++ Implementation*, The CyMeAlg Software, (2020). 4