Finding the Exact Diameter of a Graph with Partial Breadth-First Searches

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

The diameter of a graph is the longest shortest path between two nodes. This paper presents an improved algorithm for finding the exact diameter of an undirected graph. Rather than performing complete breadth-first searches, these searches can be terminated early. The algorithm is readily parallelized, and is used to find the diameters of 4-peg Tower of Hanoi problem-space graphs with up to 18 discs. Performance improvements range from a factor of almost 2 to 5.88 over the previous state of the art.

Introduction

The eccentricity of a node in a graph is the maximum distance from that node to any other node. The diameter of a graph is the largest eccentricity of any node, or the longest shortest path between any two nodes. For a problem-space graph, these nodes correspond to the initial and goal states of a hardest instance of the problem. In a communication network, the diameter is the largest distance a message would have to travel. In a social network, the diameter is the maximum degree of separation between two individuals.

We assume that the graphs are undirected, which is required by the algorithm, and that all edges have unit cost, which is not. The algorithm generalizes to weighted graphs.

For many combinatorial problems, such as Rubik's Cube, all states are equivalent, and all nodes have the same eccentricity, which is the diameter of the problem-space graph. The diameter of such a graph can be computed by a single complete breadth-first search (BFS) from any state. Finding the diameter of the Rubik's Cube problem space was an open problem for 35 years, until it was shown to be 20 moves by (Rokicki et al. 2010). This required a great deal of domain-specific analysis, and 35 cpu-years of computation.

For other problems, such as the sliding-tile problems, there are only a few non-equivalent states, based on the position of the blank in this case. For the Fifteen Puzzle, for example, only three BFSs are required to find the diameter: one from a state with the blank in a corner, one with the blank on the side, and one with the blank in the interior.

For yet other problems, such as the 4-peg Tower of Hanoi, there are nodes with many different eccentricities, requiring multiple searches to find the diameter.

Related Work

Textbook Algorithm

The simplest algorithm for this problem is to perform a complete BFS from every node of the graph, and take the maximum of their eccentricities. For a graph with V nodes and and E edges this would require $O(VE)$ time, since each BFS requires $O(E)$ time. For graphs with constant-bounded degree, this is $O(V^2)$, whereas for dense graphs it is $O(V^3)$. Note that for problem-space graphs, V is typically exponential in the size of the problem, and hence $O(V^2)$ is prohibitively expensive. For example, the 18-disc 4-peg Tower of Hanoi graph has 4^{18} or over 68 billion nodes.

Theoretical Approaches

Most of the previous work on this problem is in theoretical computer science. One goal is to reduce the cubic complexity for dense graphs. The best result appears to be $O(V^w)$, where w is the exponent of fast matrix multiplication, which is currently about 2.37 (Yuster 2011). Another goal is to design approximation algorithms, which do not compute the exact diameter. See for example (Chechik et al. 2014). Most of this work focusses on proofs of worst-case complexity and/or bounds on the degree of approximation, and does not report any implementation nor experimental results. Only a few papers report experimental results.

Cresenzi et al

The algorithm of (Crescenzi et al. 2013) starts with a complete BFS from a single root node, generating a breadth-first search tree. Then, starting from the leaf nodes and working up toward the root, it performs a complete BFS from each node, determining the eccentricity of that node. Let M be the maximum of these eccentricites, which is a lower-bound on the diameter of the graph. This process continues until the eccentricity of every node at depth greater than d in the original BFS tree has been computed, where $2d \leq M$. At this point, any shortest path with at least one endpoint in a node at depth greater than d must be less than or equal to M, since M is the maximum eccentricity of all such nodes. All that remains to consider are shortest paths between two nodes at depth d or less in the original BFS tree. None of these paths can be longer that $2d$, since we can get from any of these nodes to any other by going through the root of the original BFS tree. Since $2d \leq M$, M must be the diameter of the graph, and we can terminate the algorithm.

As the authors acknowledge, this is only effective for graphs where most of the nodes are tightly clustered around a center, with only a few outlying nodes. This is not the case with most problem-space graphs, however, where most nodes are found well beyond half the maximum eccentricity of any node. For example, for the 15-disc, 4-peg Tower of Hanoi graphs, even if the initial BFS was done from a node of minimum eccentricity, this algorithm would have to perform a complete BFS from over two-thirds of the states in the space. The largest graph whose diameter Cresenzi et al found with their method contained 149 million nodes.

Pennycuff and Weninger

The algorithm of (Pennycuff and Weninger 2015) is based on a model called vertex programming, where nodes of the graph exchange messages with neighboring nodes to determine their eccentricities. Unfortunately, this method requires $O(VE)$ messages, and $O(V^2)$ memory in most cases, making it infeasible for large graphs. The largest graphs they report experimental data for have 100 thousand nodes.

Borassi et al

The previous state-of-the-art for this problem is represented by (Borassi et al. 2015), which is heavily based on (Takes and Kosters 2011). A complete BFS from any node gives us the eccentricity of that node, but we can get more information from each BFS. Let M be the maximum eccentricity found so far, and hence a lower-bound on the diameter of the graph. Assume the eccentricity of a given root node r is e, where e is less than M, and consider a node n at a distance d from r. Since we can reach r from n via a path of length d, and we can reach any node from r via a path of length e or less, we can reach any node from n by a path of length $d + e$ or less. Thus, the eccentricity of n is no greater than $d + e$. If $d + e$ is not greater than M, then there is no need to do a BFS from node n. We say that node n is pruned in this case. Using this idea, Borassi et al perform a series of complete breadth-first searches until all nodes are pruned.

In the above scenario, the eccentricity of n is no greater than $d + e$. We could maintain a table with the least upper bound on the eccentricity of each node. If the maximum eccentricity found so far ever increases, we can prune those nodes with upper bounds on their eccentricities less than or equal to the new maximum eccentricity. This functionality is not used here, because for the 4-peg Tower of Hanoi, the initial estimate of the diameter turned out to be exact in each case, and the maximum eccentricity never increased.

Another idea in their paper is that any root node with eccentricity e produces a upper bound of $2e$ on the diameter of the graph. The reason is that we can get from any node to any other by going through the root, resulting in a path no longer than $2e$. Thus, if M is the maximum eccentricity found so far, and we find a node with eccentricity e, where $2e \leq M$, then M is the diameter of the graph. This also doesn't apply to the 4-peg Tower of Hanoi, since no state has an eccentricity close to half the maximum eccentricity.

The largest graph for which they compute the diameter consists of about 4.2 million nodes. Much of (Borassi et al. 2015) is focussed on heuristics for deciding the order of the root nodes for the BFSs, in order to find a large eccentricity early to maximize subsequent pruning. This also does not apply to our domain, since the first search returned the maximum eccentricity for each problem size.

All of their searches are complete BFSs. The main contribution of this paper is showing how to terminate these searches early, which can also be used to improve the performance of their full algorithm on any undirected graphs.

Sagharichian et al

(Sagharichian, Langouri, and Naderi 2016) describe how to get even more information out of each complete BFS. A BFS generates a rooted breadth-first tree of the graph. We can get a more accurate upper bound on the eccentricity of any node by transforming the given tree into a new tree rooted at that node. Imagine the tree consisting of a set of rigid edges connected by flexible links at each node that allow arbitrary rotations, and that we "pick up" the original tree by the node of interest, allowing the rest of the tree to fall from its new root. The depth of the deepest node in this new tree is an upper bound on the eccentricity of this new root node.

Unfortunately, computing this new tree for each node is very expensive. It requires storing the entire tree in memory, and the time complexity for each BFS is $O(E + V + Vd)$, where d is the tree depth. This last term is prohibitively expensive for the 4-peg Tower of Hanoi problem, since the diameter of the 18-disc problem is 225, for example. We will see at the end of this paper that even one step in this direction, considering the tree rooted at the neighbor of the original root along a longest path, does not pay for its overhead.

Hinz, Klavzar, and Petr

Hinz, Klavzar and Petr (Hinz, Klavzar, and Petr 2018) have done more work on the Tower of Hanoi than anyone. They computed the eccentricity of every node of the 4-peg problem with up to 15 discs, and hence the diameter, by performing a complete BFS on every state. I worked independently, but periodically informed them of my results. They have also computed the diameters for up to 18 discs, using the basic algorithm of Takes and Kosters, on a large supercomputer.

The Main Contribution of this Paper

Borassi et al perform a series of complete BFSs, each of which generate every node in the graph. These searches can be early however, resulting in partial BFSs.

When a node is pruned, it is because we know that its eccentricity is less than or equal to M, the maximum eccentricity found so far. This means that it cannot be the endpoint of any shortest path longer than M. Any longer path in the graph must be between two nodes which have not yet been pruned, which we call active nodes.

The main contribution of this paper is the observation that we can terminate each BFS as soon as all active nodes have been generated. Since we are only searching for paths between active nodes as endpoints, there is no reason to continue a BFS once all active nodes have been generated.

Once we terminate a search when all active nodes have been generated, we can use the maximum depth reached at that point as if it were the eccentricity of the root, for purposes of pruning other nodes. The reason is that we are only searching for shortest paths between active node end points. Let e' be the maximum distance reached from the root to any active node. Given a node n at a distance d from the root, the maximum distance from n to any other active node can be no greater than $d+e'$, since we can reach any node by going through the root. Thus, if $d+e'$ is less than or equal to M, then node n can be pruned.

There are two benefits to this early termination. The first is that by terminating early, we reduce the time of each BFS. Experimentally, this effect is very small, however. More significantly, the lower value of e' compared to the actual eccentricity e allows us to prune many more nodes than if the search had completed, reducing the number of searches.

The Serial Algorithm

The algorithm uses a state table with one entry per state. This entry stores whether the state is active or pruned, whether it has been visited in the current BFS, and if so, its depth. Each state is mapped to a unique index, so we don't have to store the states themselves. In addition, a circular list is used to store the queue of nodes to be expanded by the BFS. Each queue entry must be large enough to store a state. The number of queue entries needed is the maximum width of a BFS, which is the largest number of nodes at any depth.

Note that since the queue is accessed strictly sequentially, it can be stored on disk instead of in memory. One file can be used for nodes at each depth, with small buffers in memory for input and output. This is slower than an in-memory implementation however.

We begin with all states marked as active, and perform a BFS from each active state. Before each BFS, we initialize every state to unvisited, regardless of its active or pruned status, and count the number of active states. Then we perform the BFS, storing the depth of each state visited, and counting the number of active states generated, until that number equals the total number of active states. Let e' be the maximum depth reached by this search, and let M be the maximum eccentricity so far. Then we linearly scan the state table, and for each state at a distance d from the root for which $d+e' \leq M$, we mark that state as pruned, including the root. This scan is also used to initialize all the states to unvisited, and count the number of active states, in preparation for the next BFS. We then perform a BFS on the next active state in the order, until all active states have been pruned.

One bit of the state table indicates if the state is active or pruned, and the remaining bits store the depth of the state in the current BFS. States that have not been visited in the current BFS have all their depth bits set to one. Since we can only prune nodes close to the root, we don't need to store the full range of search depths, but can set a maximum depth value, and use that value for any deeper states, if we don't prune any states at this depth. With one byte per state, using one bit for active or pruned leaves 7 bits for the state depth in the current BFS. Using 127 to denote states not yet visited in the current BFS gives us a depth range of 0 to 126.

The state table can be reduced to just two bits per state, storing only active and visited status, but not the state depths, in two different ways. One is after each BFS, we restart the same BFS and run it only to the maximum depth at which nodes can be pruned. In most cases this depth is relatively shallow. Alternatively, during each BFS we could write each state and its depth sequentially to a disk file, then after the BFS we read the first part of that file, pruning the nodes at shallow depths.

The Parallel Algorithm

The serial version of this algorithm is limited by time, not memory. Almost all modern computers have multiple processors and cores, suggesting parallelizing the algorithm to reduce its runtime. We assume here a shared memory model, where each thread has access to the entire memory of the machine. Since the algorithm consists of many individual BFSs, the most natural way to parallelize the algorithm is for different threads to perform different searches.

Each BFS is independent, and each thread has its own state table and queue of nodes to be expanded. The active or pruned status of a node is a global property, however, stored in a single table shared by all threads, using one bit per state. No locking of this shared table is needed, since each state goes from active to pruned, and multiple writes marking the same state as pruned have no effect on the final result.

Each thread needs a local copy of this table as well. The reason is that each BFS starts with a target number of active states, and runs until they have all been generated. If another thread marks one or more of those active states as pruned while the first BFS is running, it won't count that state in its tally of visited active states, and never reach its target number. In that case, it will perform a complete BFS, negating the advantage of the algorithm. The local copy of this table is initialized with the active/pruned status from the global table at the start of each BFS.

The main drawback of this parallel algorithm is that it performs searches on root nodes that the serial algorithm does not, because those nodes may be pruned by another search which is in progress when the BFS starts. This results in a parallel overhead represented by more total searches performed than by the serial algorithm. To minimize this overhead, the root nodes of the different threads are spaced as far apart in the problem space as possible.

As we will see, the function we use to index states in the 4-peg Tower of Hanoi problem has the property that states that have nearby indices also tend to be close in the problem space. Thus, we divide the entire interval of states evenly among the number of parallel threads, and each thread starts at a different point in this interval. Once it reaches the end of the interval, it wraps around to the beginning, and continues until it reaches the state at which it started. This mechanism keeps the threads working on different parts of the problem space initially, but they tend to bunch up near the end.

Another optimization is that before a thread begins a BFS on a root node, it immediately prunes that node in the global table of pruned nodes, so that another parallel thread will not start another search with the same root.

Experiments

4-Peg Tower of Hanoi Problem

Our experimental domain is the 4-peg Tower of Hanoi problem. It provides a set of problem-space graphs, each of size 4^n, where n is the number of discs. The eccentricity of states varies significantly, and the diameters of the problem spaces are not known a priori. For the 3-peg Tower of Hanoi, it is easy to prove that the diameter of the problem-space graph is $2^n - 1$ for n discs, which is also the number of moves needed to transfer all discs from one peg to another.

The eccentricity of a state with all discs on the same peg is a good estimate of the diameter of the graph, and the first BFS starts from such a state. For all problems that have been run to date, this is in fact the diameter of the graph, but there is no proof of this conjecture in general.

The pegs are indistinguishable. Thus, most states represent $4! = 24$ symmetric states that differ only by a permutation of the pegs, and have the same eccentricity. The relatively few states with two or more empty pegs have fewer than 23 symmetric partners. Whenever we prune a state, we also prune its symmetric states. This reduces the number of searches by almost a factor of 24.

A problem state is represented by specifying the peg each disc is on, since the discs on each peg are stacked in order of size. For the 4-peg problem, two bits per disc are both necessary and sufficient. This provides a bijection between the set of states and the numbers from zero to $4^n - 1$. These numbers are used as the index into the state table. The positions of the smallest discs are represented by the least significant bits. Since most nearby states differ in the positions of the smallest discs, this provides locality of reference in the state table that improves cache performance.

Experimental Results

The results are shown in Table 1. The top half of the table reports on the serial algorithms, and the bottom half on the parallel algorithms. "Complete BFS" is the previous state-of-the-art, performing complete BFSs, but without storing upper bounds on the eccentricity of each node, since that never leads to any pruning in this domain. "Partial BFS" is our improved algorithm that terminates each BFS when all active nodes have been generated. Each row gives the number of discs, the diameter of the problem space, the number of BFSs, and the running time in the form of days:hours:minutes:seconds. The last line of each row is the ratio of the running time of the complete BFS algorithm divided by the running time of the partial BFS algorithm. All runs were done on an Intel Xeon machine running CentOS Linux, version 7.8.2003, at 3.3 gigahertz, with 240 gigabytes of memory. Code was written in C, and compiled with GCC, with optimization level 3. Pthreads were used to implement the parallel algorithms. All algorithms were implemented entirely in memory, without the use of disk storage.

The order in which the BFSs are done affects the number of searches, by affecting the amount of pruning done with each BFS. BFS roots were chosen in increasing order of their index values. Several other orders were tried, such as reverse index order and random orders. Both performed worse than the forward sequential order, and thus results are reported for the forward index order.

Note that the reverse index order is not simply generating all the indices in reverse order, since this produces the same results as the forward index order, because of symmetry among states due to permuting the pegs. Rather, we first generate all canonical states, where each canonical state represents an entire group of 24 symmetric states in most cases. A canonical state is generated by sorting the pegs in order of the largest disc on each peg. Then we perform searches on just the canonical states in decreasing order of their indices.

Serial Performance

The first thing to notice is how effective the pruning of even the complete BFS algorithm is compared to performing a BFS from every state. The maximum number of searches in the table, 22296, is less than .05% of the total number of non-symmetric states in the 15-disc problem space.

The running times of both algorithms are linear in the number of searches, and the time per search grows by about a factor of four with each additional disc, since the size of the problem space is 4^n. There is some overhead in the partial BFS algorithm to count the number of active nodes generated, making it about 11% slower per search than the complete BFS algorithm, but it performs many fewer searches.

For 10 through 13 discs, the partial BFS algorithm performs about half the searches of the complete BFS algorithm. As a result, it takes a little more than half the time. For 14 discs, the complete BFS algorithm performs 2.56 times as many searches as the partial BFS algorithm, and runs 2.35 times longer. For 15 discs, the complete BFS algorithm performs 6.22 times as many searches, and runs 5.88 times longer than the partial BFS algorithm. This ratio tends to increase with the number of searches.

Parallel Performance

For larger problems, the parallel versions of both algorithms were used. My machine has 12 cores distributed over two processors, and 12 parallel threads were used, for all but the 18-disc problem, for which there was only enough memory for 6 parallel threads. Also due to memory limitations, for the 17 and 18 disc problems, the depth of each state was not stored as it was generated, but the nodes to be pruned were regenerated after each BFS. For 15 discs, the largest problem for which the serial algorithm was run, the parallel speedup in the time to conduct each search was about 11 for the serial algorithm, and 10.5 for the parallel algorithm.

The overall parallel speedup is less than this, due to the increased number of searches performed by the parallel algorithm, compared to the serial algorithm. As explained above, this is due to searches of nodes that are pruned by other threads running simultaneously. In these experiments, this search overhead was a factor of 1.457, 1.622, and 1.664 for 13, 14, and 15 discs, respectively.

For 18 discs, it was impractical to run the complete BFS algorithm. The number of searches listed for this algorithm, 14332, comes from Andreas M. Hinz and Ciril Petr, who computed the diameter of this problem using the complete

Comparison of Serial Algorithms						
		Complete BFS		Partial BFS		Ratio
Discs	Diameter	Searches	Time	Searches	Time	
10	49	330	:16	159	:9	1.78
11	65	239	:46	120	:25	1.84
12	81	272	3:33	140	2:02	1.75
13	97	558	30:24	256	15:41	1.94
14	113	1597	5:58:50	624	2:32:37	2.35
15	130	13663	8:19:48:01	2195	36:00:18	5.88
Comparison of Parallel Algorithms						
		Complete BFS		Partial BFS		Ratio
Discs	Diameter	Searches	Time	Searches	Time	
13	97	858	4:24	373	2:12	2.00
14	113	2387	49:23	1012	23:15	2.12
15	130	22296	1:07:22:38	3652	5:41:24	5.51
16	161	7532	1:19:58:28	1981	12:09:21	3.62
17	193	6976	6:19:04:10	2022	2:05:25:18	3.05
18	225	14332*		2378	22:21:47:17	5.33*

Table 1: Experimental Results on 4-Peg Tower of Hanoi Problem

BFS algorithm on a large supercomputer[1]. Their running time is not listed, since they ran on a different machine, used a less efficient BFS algorithm, and typically employed 48 processors in parallel. Their implementation took 2.29 cpu hours per BFS on their machine, while mine took about 1.39 cpu hours on my machine.

The number of searches is not directly comparable either, since they used a random search order, and I used the index order. The ratio of 5.33 is an estimate based on the ratio of the number of searches performed, correcting this by the 13% slower speed per search of the partial BFS algorithm compared to the complete BFS algorithm on 17 discs.

In general, the parallel partial BFS algorithm runs between 2 and 5.51 times faster than the complete BFS algorithm. There are significant differences in this ratio for different size problems. As with the serial algorithms, the partial BFS algorithm is slightly slower than the complete BFS algorithm per search, despite generating fewer nodes. The performance gains come from performing fewer searches, since terminating a search at a shallower depth allows many more nodes to be pruned.

The number of searches doesn't vary smoothly with the number of discs. The smaller problems generate fewer searches, and the larger ones more, but it isn't monotonic. In particular, the 15-disc problem requires the most searches by far. I don't know why, but 15 discs is unusual for another reason. For almost all size problems up to 20 discs, the optimal solution length for moving all discs from one peg to another equals the eccentricity of the canonical initial state with all discs on the same peg, except for 15 discs. In this case the optimal solution length is 129 moves, but there are states that are 130 moves away from the canonical initial state (Korf 2008). For 20 through 26 discs, the eccentricity of this canonical initial state is also greater than the optimal solution length (Korf 2008; Hinz, Klavzar, and Petr 2018), with the difference increasing with increasing numbers of discs. We conjecture that the diameter of the graph always equals the eccentricity of the canonical state with all discs on the same peg, but no proof of this conjecture is known. This work and that of Hinz and Petr independently confirm this conjecture for up to 18 discs.

[1]Ciril Petr and Andreas M. Hinz, personal communication, December 2020

An Additional Optimization?

The pruning power of each BFS is based on the maximum tree depth. For each node, we add its distance to the root to the maximum tree depth, and if the sum is no greater than the current lower bound on the diameter, then we prune that node. As described in the related work section, computing a new tree from each BFS but rooted in every other node is too expensive, but we can take one small step in this direction.

Consider a BFS tree rooted at node r, with a longest path of length e, with node n being the immediate neighbor of r on that path. We can prune node n if $1 + e$ is no greater than the current lower bound on the diameter. However, the endpoint of this longest path can be reached from node n in $e - 1$ moves, since there is no need to go from n to r, and back to n again. The maximum eccentricity of n is therefore the maximum of $e-1$ and $1+f$, where f is the longest path from any other neighbor of the root. More generally, for all nodes descendent from n, and at a distance d from the root, their maximum eccentricity is the maximum of $e+d-2$ and $d + f$. This can result in more pruning among those nodes. This idea can be extended by considering the descendants of the nodes at depths two, three, or more from the root.

To implement the one-step version of this, for each state generated we keep track of which neighbor of the root it is descended from. We also keep track of the maximum depth of the nodes descended from each neighbor. In experiments with 13 disks, this reduced the number of searches required, resulting in a 12.75% reduction in running time. For 14 discs, this resulted in only a 1.23% reduction in time. For 15 disks, the number of searches was reduced, but the additional overhead of this method resulted in a 4.24% slowdown in running time. Thus, this optimization doesn't seem to pay for itself on larger problems in this domain.

Extension to Non-Uniform Edge Costs

So far, we have assumed unit edge costs. The algorithms described generalize to the case of weighted graphs, where different edges have different costs. Instead of the length of a path, we use the cost of a path, which is the sum of its edge costs. The diameter becomes the largest cost of any lowest-cost path between two nodes. In place of breadth-first search, we use uniform-cost search, or equivalently Dijkstra's single-source shortest path algorithm (Dijkstra 1959). We can still terminate these searches when all active nodes have been generated, rather than completing them. A node can be pruned when the cost of reaching the root from that node, plus the largest cost of any path from the root is no greater than the current lower bound on the diameter. The algorithm does require that the graph be undirected, in that the cost of an edge must be the same in either direction.

Summary and Conclusions

The diameter is a fundamental property of a graph with many applications. Despite this, there has been very little empirical work on computing exact graph diameters. Even the complete BFS algorithm of (Takes and Kosters 2011) performs fewer than .05% of the searches of the textbook algorithm on the 4-peg Tower of Hanoi problem. While all previous algorithms perform complete BFSs, the primary contribution of this paper is terminating these searches early, once all active nodes have been generated. The algorithm is readily parallelized to run on multiple cores.

The performance of this partial BFS algorithm was compared to the complete BFS algorithm for computing the exact diameter of 4-peg Tower of Hanoi problem-space graphs. Improvements ranged from almost a factor of two to a factor of 5.88. The 18-disc graph has over 68 billion nodes, which is more than four orders of magnitude larger than any other graph types whose exact diameter computations have been reported in the literature.

While unit edge costs were assumed, this work can be generalized to the case of undirected graphs with weighted edge costs, by substituting uniform-cost search for BFS.

Acknowledgements

I would like to thank Ciril Petr and Andreas M. Hinz for helpful correspondence regarding this problem. Their work computing the eccentricities of every node in the 4-peg Tower of Hanoi problem-space graphs with up to 15 discs originally inspired me to compute the diameters of larger graphs. I recently informed them of my latest results, and they have computed the diameters of the 16 through 18-disc problems, using the complete BFS algorithm on a large supercomputer.

References

Borassi, M.; Crescenzi, P.; Habib, M.; Kosters, W. A.; Marino, A.; and Takes, F. W. 2015. Fast diameter and radius BFS-based computation in (weekly connected) real-world graphs with an application to the six degrees of separation games. *Theoretical Computer Science* 586: 59–80.

Chechik, S.; Larkin, D. H.; Roditty, L.; Shoenebeck, G.; Tarjan, R. E.; and Williams, V. V. 2014. Better approximation algorithms for the graph diameter. In *Proceedings of the 25th Annual ACM-SIAM Syposium on Discrete Algorithms (SODA '14)*, 1041–1052. Portland, OR.

Crescenzi, P.; Grossi, R.; Habib, M.; Lanzi, L.; and Marino, A. 2013. On computing the diameter of real-world undirected graphs. *Theoretical Computer Science* 514: 84–95.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerishe Mathematik* 1: 269–71.

Hinz, A. M.; Klavzar, S.; and Petr, C. 2018. *The Tower of Hanoi—Myths and Maths, 2nd ed.* Cham, Switzerland: Springer Birkhauser.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the Association for Computing Machinery (JACM)* 55(6): 26–1:26–40.

Pennycuff, C.; and Weninger, T. 2015. Fast, Exact Graph Diameter Computation with Vertex Programming. In *1st High Performance Graph Mining Workshop (HPGM '15)*. Sydney, Australia.

Petr, C. 2020. Personal communication.

Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2010. God's number is 20. https:www.cube20.org. Accessed:2021-05-28.

Sagharichian, M.; Langouri, M. A.; and Naderi, H. 2016. Calculating exact diameter metric of large static graphs. *Journal of Universal Computer Science* 23(3): 302–318.

Takes, F. W.; and Kosters, W. A. 2011. Determining the diameter of small world networks. In *Conference on Information and Knowledge Management (CIKM-11)*, 1191–1196. Glasgow, Scotland.

Yuster, R. 2011. Computing the diameter polynomially faster than APSP. arXiv:1011.6181v2.