Maximum Likelihood Analysis of the Ford-Fulkerson Method on Special Graphs

ULRICH LAUBE and MARKUS E. NEBEL, Technische Universität Kaiserslautern

We present original average-case results on the performance of the Ford-Fulkerson maxflow algorithm on grid graphs (sparse) and random geometric graphs (dense). The analysis technique combines experiments with probability generating functions, stochastic context free grammars and an application of the maximum likelihood principle enabling us to make statements about the performance, where a purely theoretical approach has little chance of success. The methods lends itself to automation allowing us to study more variations of the Ford-Fulkerson maxflow algorithm with different graph search strategies and several elementary operations.

A simple depth-first search enhanced with random iterators provides the best performance on grid graphs. For random geometric graphs a simple priority-first search with a maximum-capacity heuristic provides the best performance. Notable is the observation that randomization improves the performance even when the inputs are created from a random process.

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Analysis of algorithms, average case, generating functions, Ford-Fulkerson maxflow, maximum likelihood analysis, random geometric graphs, stochastic context free grammars

Reference Format:
Laube, U., Nebel, M. E. 2013. Maximum Likelihood Analysis of the Ford-Fulkerson Method on Special Graphs.

1. INTRODUCTION

Predicting the performance and making decisions about implementation details based on worst-case bounds is a tricky business. Often there is a large gap between the worst-case bound and the actual performance of the algorithm. Thus the worst-case bounds are of limited use for a practitioner.

The network-flow problem is of interest not only because numerous efficient algorithms for solving this problem have been devised, but also many reductions to it allow a whole range of distribution, matching and cut problems to be solved as well.

In case of the Ford-Fulkerson method the known worst-case results are products of conservative upper bounds for the number of augmenting paths and the work per path. For example on a grid graph with 400 vertices, 760 edges and an edge capacity limit of 100 the bound for the number of augmenting path is as high as $40\,000$ while we actually observe only around 35 augmenting paths with breadth-first-search (bfs) in experiments. For priority-first-search (pfs) the bound is $7\,000$ but we only observe around 5 paths on average. Predicting the performance and making decisions about implementation details based on the bounds is futile.

An alternative would be an average-case analysis, but even for an algorithm of moderate complexity as the augmenting path method, this is a difficult task. Despite being one of the better understood combinatorial optimizations problems the work on the algorithms for computing a maximum flow has mainly concentrated on giving increasingly better worst-case bounds, as summarized in [Goldberg and Rao 1998]. Experimental work on the algorithms for computing the maximum flow has an equally long history, see [Goldfab and Grigoriads 1988] for blocking flows or [Chandran and Hochbaum 2009] for the push-relabel method.

This work is supported by DFG grants NE 1379/2-1 and NE 1379/3-1.
Author's address: Fachbereich Informatik, Technische Universität Kaiserslautern, Gottlieb-Daimler-Straße, 67663 Kaiserslautern, Germany. Email: {laube, nebel}@informatik.uni-kl.de. Tel.: +49 631 205 2509. Fax: +49 631 205 2573.

We will use a hybrid approach, first presented by the authors in [Laube and Nebel 2010], to obtain original average-case results. The *maximum likelihood analysis* involves a formal model of the algorithms behavior derived from the specification of the algorithm, that is subsequently trained using simulation runs on sample inputs. One advantage is that the method lends itself to automation allowing us to study more variations of the algorithm and different sample inputs set. Another advantage is that we can use independent results to crosscheck the derived model and gain confidence that it describes the algorithms behavior adequately. Furthermore the method not just counts the overall number an elementary operation is executed, it provides cost contributions of the different parts of an algorithm not just confirming that one algorithms invokes less elementary operation than another but also hinting *why*. Last but not least, aiming at the average-case performance of an algorithm allows the consideration of randomized variants of an algorithm in a homogeneous way.

This method has been implemented in our tool MaLiJAn[1] which already has successfully been used to investigate the new Java 7's dual pivot quicksort, see [Wild et al.].

The results are not only useful for practitioners as they may guide a theoretician in a common average-case analysis by perhaps ruling out certain cases or giving hints which part of the algorithm influences a certain performance parameter.

We selected two graph types: grid graphs and random geometric graphs. The reason for looking into grid graphs is that they have a rich enough structure that allows to proof at least some results about the properties of the algorithms working on them. They are easily scalable, similar graphs may be found in practice (e.g. a pixel grid in computer graphics) and they are yet sufficiently challenging to be interesting.

The order of the neighbors in an adjacency list representation is a degree of freedom that might be exploited to speed up an algorithm. Due to their regular structure grid graphs provide a nice setup to study this effect.

The reason for looking into random geometric graphs is that the triangle property is often more realistic (e.g. brain cells connected to nearby ones as are communication facilities that are distributed across an area) than the independence of the edges as in the Erdös-Rényi model.

Note that the algorithms we are investigating work on all graphs and they do not use any special property of grid graphs or random geometric graphs to compute the result faster on those graphs.

The main questions addressed here are a comparison of the well known graph search strategies, depth-first search (`dfs`), breath-first search (`bfs`) and priority-first search (`pfs`) in the Ford-Fulkerson method for determining a maxflow. We are also looking into the typically strong dependence of a graph algorithm's performance on the order of the vertecies' adjacency lists by studying the effect that the use of random iterators in the graph search strategies have. They basically turn a graph algorithm into a randomized algorithm (denoted `dfsrnd` and `bfsrnd` in the remainder of this paper).

2. RESULTS

The following results were derived semi-automatically with the maximum likelihood analysis method as described in [Laube and Nebel 2010]. An overview of this analysis method is found in Appendix A.6. The models of the behavior of the Ford-Fulkerson algorithm on the different grids graphs are defined in Section 4 and Section 5. They are denoted by M_d^l where $d \in \{\mathrm{unit}, \mathrm{unif}, \mathrm{gauss}\}$ is one of the edge weight distributions and $l \in \{\mathrm{llur}, \mathrm{lmum}\}$ describes the locations of the source and sink.

[1] The tool can be downloaded at http://wwwagak.cs.uni-kl.de/malijan.html

Table I. Leading terms of the average number of augmenting paths.

	model	`dfs`	`bfs`	`pfs`	`dfsrnd`	`bfsrnd`
grid	$M^{\text{llur}}_{\text{unit}}$	2	2	2	2	2
	$M^{\text{lmum}}_{\text{unit}}$	3	3	3	3	3
	$M^{\text{llur}}_{\text{unif}}$	66.57	$0.81\log^2(n)$	3.53	66.1	$0.78\log^2(n)$
	$M^{\text{lmum}}_{\text{unif}}$	101.32	$0.97\log^2(n)$	5.27	111.4	$0.97\log^2(n)$
	$M^{\text{llur}}_{\text{gauss}}$	$3.11\sqrt{n}$	$4.00\log(n)$	3.24	$26.0\log(n)$	$0.80\log(n)$
	$M^{\text{lmum}}_{\text{gauss}}$	$0.27\sqrt{n}$	$2.61\log(n)$	4.90	$48.9\log(n)$	$2.65\log(n)$
rnd. geom.	$M^{\text{llur}}_{\text{unit}}$	$0.016n$	$0.017n$	$0.016n$	$0.016n$	$0.017n$
	$M^{\text{lmum}}_{\text{unit}}$	$0.034n$	$0.034n$	$0.034n$	$0.034n$	$0.034n$
	$M^{\text{llur}}_{\text{unif}}$	$0.283n$	$0.123n$	$0.019n$	$0.209n$	$0.110n$
	$M^{\text{lmum}}_{\text{unif}}$	$0.563n$	$0.202n$	$0.039n$	$0.373n$	$0.183n$
	$M^{\text{llur}}_{\text{gauss}}$	$0.300n$	$0.096n$	$0.019n$	$0.239n$	$0.064n$
	$M^{\text{lmum}}_{\text{gauss}}$	$0.595n$	$0.166n$	$0.039n$	$0.434n$	$0.114n$

Table II. Average number of elementary operations on the queue in `bfs` and `bfsrnd`.

		`bfs`				`bfsrnd`			
		total		per path		total		per path	
	model	`put`	`get`	`put`	`get`	`put`	`get`	`put`	`get`
grid	$M^{\text{llur}}_{\text{unit}}$	$2n$	$2n$	1.00	1.00	$2.00n$	$2.00n$	1.00	1.00
	$M^{\text{lmum}}_{\text{unit}}$	$2.35n$	$2.31n$	0.78	0.77	$2.35n$	$2.31n$	0.78	0.77
	$M^{\text{llur}}_{\text{unif}}$	$0.74n\log^2 n$	$0.74n\log^2 n$	0.91	0.91	$0.72n\log^2 n$	$0.72n\log^2 n$	0.92	0.92
	$M^{\text{lmum}}_{\text{unif}}$	$0.79n\log^2 n$	$0.78n\log^2 n$	0.81	0.80	$0.79n\log^2 n$	$0.78n\log^2 n$	0.81	0.80
	$M^{\text{llur}}_{\text{gauss}}$	$3.03n\log(n)$	$3.03n\log(n)$	0.76	0.76	$9.93n$	$9.94n$	$\mathcal{O}(1/\log(n))$	
	$M^{\text{lmum}}_{\text{gauss}}$	$2.11n\log(n)$	$2.07n\log(n)$	0.81	0.80	$2.20n$	$1.99n$		
rnd. geom.	$M^{\text{llur}}_{\text{unit}}$	$0.0161n^2$	$0.0163n^2$	0.98	0.97	$0.0160n^2$	$0.0162n^2$	0.96	0.97
	$M^{\text{lmum}}_{\text{unit}}$	$0.0333n^2$	$0.0330n^2$	0.99	0.98	$0.0333n^2$	$0.0329n^2$	0.99	0.98
	$M^{\text{llur}}_{\text{unif}}$	$0.1197n^2$	$0.1190n^2$	0.99	0.98	$0.1090n^2$	$0.1084n^2$	0.99	0.98
	$M^{\text{lmum}}_{\text{unif}}$	$0.2000n^2$	$0.1982n^2$	0.99	0.98	$0.1823n^2$	$0.1802n^2$	1.00	0.98
	$M^{\text{llur}}_{\text{gauss}}$	$0.0943n^2$	$0.0940n^2$	0.98	0.98	$0.0630n^2$	$0.0626n^2$	0.99	0.99
	$M^{\text{lmum}}_{\text{gauss}}$	$0.1641n^2$	$0.1630n^2$	0.99	0.98	$0.1140n^2$	$0.1132n^2$	0.99	1.00

Table I shows the average number of augmenting paths necessary to compute a maxflow with the Ford-Fulkerson method, using one of the five graph search strategies (`dfs`, `bfs`, `pfs`, `dfsrnd`, `bfsrnd`) in the models M^l_d where $d \in \{\text{unit}, \text{unif}, \text{gauss}\}$ and $l \in \{\text{llur}, \text{lmum}\}$.

The main operations on queues used in `bfs` and `bfsrnd` are `put` and `get`. The averages for the different models M^l_d are shown in Table II.

When `dfs` and `dfsrnd` are used as graph search strategies we investigate the average number of elementary operations on the stacks (`push`/`pop`). Table III lists the results when computing a maxflow with the Ford-Fulkerson method using the `dfs` and `dfsrnd` graph search strategies in the models M^l_d.

Tables II–IV include columns that list the average number of operations divided by the number of paths times the number of vertecies n, thus indicating the average fraction of all vertecies of the graph that were stored in the queue or stack per path.

When the `pfs` graph search strategy is used the elementary operations on the priority queue are `getmin`/`lower`; their average counts under the models M^l_d are given in Table IV.

Thus far we have looked at the fraction of vertecies involved in the different operations. It is also instructive to look at the average fraction of edges that were visited during the graph search. We consider an edge visited if both its vertecies are removed from the stack or queue. Thus divding the number of `pops`, `gets` and `getmins` by the number

Table III. Average number of elementary operations on the stack in `dfs` and `dfsrnd`.

		`dfs`				`dfsrnd`			
		total		per path		total		per path	
	model	`push`	`pop`	`push`	`pop`	`push`	`pop`	`push`	`pop`
grid	$M^{\text{llur}}_{\text{unit}}$	$2.00n$	$1.03n$	1.00	0.52	$1.40n$	$1.01n$	0.70	0.51
	$M^{\text{lmum}}_{\text{unit}}$	$0.86n$	$0.76n$	0.29	0.25	$2.05n$	$1.51n$	0.68	0.50
	$M^{\text{llur}}_{\text{unif}}$	$66.70n$	$39.79n$	1.00	0.60	$40.56n$	$29.53n$	0.61	0.45
	$M^{\text{lmum}}_{\text{unif}}$	$83.28n$	$57.00n$	0.82	0.56	$64.82n$	$47.25n$	0.58	0.42
	$M^{\text{llur}}_{\text{gauss}}$	$2.82n^{3/2}$	$1.83n^{3/2}$	0.91	0.59	$102.81n$	$72.64n$	$\mathcal{O}(1/\log(n))$	
	$M^{\text{lmum}}_{\text{gauss}}$	odd-even effect as discussed in Sec. 9				$204.24n$	$145.44n$		
rand. geom.	$M^{\text{llur}}_{\text{unit}}$	$0.016n^2$	$0.004n^2$	1.00	0.27	$0.015n^2$	$0.004n^2$	0.86	0.27
	$M^{\text{lmum}}_{\text{unit}}$	$0.032n^2$	$0.011n^2$	0.94	0.34	$0.031n^2$	$0.012n^2$	0.93	0.36
	$M^{\text{llur}}_{\text{unif}}$	$0.273n^2$	$0.063n^2$	0.97	0.22	$0.180n^2$	$0.052n^2$	0.86	0.25
	$M^{\text{lmum}}_{\text{unif}}$	$0.577n^2$	$0.161n^2$	1.02	0.29	$0.340n^2$	$0.134n^2$	0.91	0.36
	$M^{\text{llur}}_{\text{gauss}}$	$0.293n^2$	$0.067n^2$	0.98	0.23	$0.206n^2$	$0.060n^2$	0.86	0.25
	$M^{\text{lmum}}_{\text{gauss}}$	$0.468n^2$	$0.172n^2$	0.79	0.29	$0.395n^2$	$0.015n^2$	0.91	0.36

Table IV. Average number of elementary operations on the priority queue in `pfs`.

	`pfs`	total		per path	
	model	`getmin`	`lower`	`getmin`	`lower`
grid	$M^{\text{llur}}_{\text{unit}}$	$1.78n$	$1.90n$	0.89	0.95
	$M^{\text{lmum}}_{\text{unit}}$	$2.63n$	$2.87n$	0.88	0.96
	$M^{\text{llur}}_{\text{unif}}$	$3.84n$	$6.45n$	1.09	1.83
	$M^{\text{lmum}}_{\text{unif}}$	$5.29n$	$9.00n$	1.00	1.71
	$M^{\text{llur}}_{\text{gauss}}$	$3.55n$	$5.96n$	1.10	1.84
	$M^{\text{lmum}}_{\text{gauss}}$	$4.96n$	$8.41n$	1.01	1.72
rand. geom.	$M^{\text{llur}}_{\text{unit}}$	$0.0092n^2$	$0.0195n^2$	0.56	1.19
	$M^{\text{lmum}}_{\text{unit}}$	$0.0189n^2$	$0.0300n^2$	0.56	0.90
	$M^{\text{llur}}_{\text{unif}}$	$0.0190n^2$	$0.0782n^2$	0.99	4.10
	$M^{\text{lmum}}_{\text{unif}}$	$0.0386n^2$	$0.1907n^2$	0.99	4.89
	$M^{\text{llur}}_{\text{gauss}}$	$0.0187n^2$	$0.0843n^2$	1.00	4.49
	$M^{\text{lmum}}_{\text{gauss}}$	$0.0382n^2$	$0.2058n^2$	0.99	5.33

Table V. Average fraction of edges visited during the different graph search strategies. The entry $(*)$ is omitted due to the odd-even effect as discussed in Section 9.

model		`dfs`	`bfs`	`pfs`	`dfsrnd`	`bfsrnd`
grid	$M^{\text{llur}}_{\text{unit}}$	0.26	0.50	0.45	0.25	0.50
	$M^{\text{lmum}}_{\text{unit}}$	0.13	0.39	0.44	0.25	0.39
	$M^{\text{llur}}_{\text{unif}}$	0.30	0.47	0.53	0.22	0.46
	$M^{\text{lmum}}_{\text{unif}}$	0.28	0.40	0.50	0.21	0.40
	$M^{\text{llur}}_{\text{gauss}}$	0.45	0.38	0.54	$\mathcal{O}(1/\log(n))$	
	$M^{\text{lmum}}_{\text{gauss}}$	$(*)$	0.40	0.51		
rand. geom.	$M^{\star}_{\star}$	$\mathcal{O}(1/n)$			$\mathcal{O}(1/n)$	

of paths times the number of edges ($2(n-\sqrt{n})$ for grid graphs and $0.031n^2$ for random geometric graphs (see Corollary 5.3), we can determine the average fraction of edges visited per path as shown in Table V for grid graphs. In case of random geometric graphs we determined that in all the models and graph search strategy combinations the average fraction of edges visited per path is in $\mathcal{O}(1/n)$.

These results not only allow us to select the fastest graph strategies for the input families considered here: A simple depth-first search enhanced with random iterators provides the best performance on grid graphs. And a simple priority-first search with a maximum-capacity heuristic provides the best performance for random geometric graphs.

One of the insights that goes beyond a mere performance characteristic is that it is beneficial to break any regularities in the order that the adjacency lists are build up. And even if the inputs are created from a random process it is a notable observation that randomization still improves the performance.

In the following sections we will present the details on the network-flow problem, the Ford-Fulkerson-Method, the input models and discuss the results. An introduction to the maximum likelihood analysis method is provided in the appendix.

3. NETWORK FLOW

Following [Sedgewick 2003] a *network* is just a weighted digraph where the edge weights are called *capacities*. The *network flow-problem* asks for a second set of edge weights, called the *flow*, that has to satisfy certain constraints. The flow may not exceed the capacities and the *total flow* in and out of a vertex is 0 with the exception of two designated vertecies the *source* and the *sink*. Including an imaginary extra edge that connects source and sink, this flow conservation property holds for all vertecies and the flow on this extra edge is the *value* of the flow. We specifically address the *maxflow-problem* (we are asked to determine a flow that is maximal amongst all feasible flows) and investigate one approach for solving it, the *Ford-Fulkerson method* or *augmenting-path method* introduced in [Ford and Fulkerson 1962].

We selected the Ford-Fulkerson method for a number of reasons. While there are average-case results for other maxflow algorithms, e.g. [Motwani 1994], where results about Dinic's algorithm [Dinic 1970] on random bipartite graphs are given, to the best of our knowledge, we are not aware of average-case results for the classical Ford-Fulkerson method. Also the Ford-Fulkerson method is conceptually simpler than push-relabel, blocking-flow, pseudoflow and Orlin's [Orlin 2013] approach allowing us to verify the results and gain more confidence in the new analysis method.

3.1. Ford-Fulkerson Method

The Ford-Fulkerson method is a rather generic approach, as it does not include a specific rule for finding the augmenting paths. A subroutine `augment(s, t)` is required that finds a non-saturated path (a path whose edges have not reached their capacity limit) during a graph search phase and then saturates the path by pushing a fraction of the flow along that augmenting path.

Various recommendations have been given in the literature such as breadth-first or a greedy maximum-capacity search for finding an augmenting path. We selected three implementations from [Sedgewick 2003] included in the appendix: The general priority-first search `pfs` uses a multiway-heap to implement a priority queue. Choosing the priority to be the remaining capacity of the edges and taking the edge with the largest amount leads to the maximum-capacity-augmenting-path variant from [Edmonds and Karp 1972]. Other choices for the priority could be used to simulate a stack or a simple queue, however this would incur an additional factor of $\mathcal{O}\big(\log(|V|)\big)$ on the priority queue's operations compared to a direct implementation. Therefore the implementations `dfs` and `bfs` use a stack and a simple queue directly. Using `bfs` yields the shortest-augmenting-path heuristic of Edmonds and Karp from [Edmonds and Karp 1972].

Randomized versions of the Ford-Fulkerson method can be devised by using a randomized queue or by randomizing the access to the adjacency lists. This could be done once, when the graph is read into memory for the first time, or upon every access

to the adjacency lists. We use the implementation from [Sedgewick 2003] and add a random iterator which is easily possible. This random iterator is just an adaption of the Durstenfeld variant of the Fisher-Yates-Shuffle [Durstenfeld 1964; Knuth 1998] for the adjacency lists. This adds of course an expensive call to a random number generator to the iterator.

The choice for a Java implementation was made as the tool *MaLiJAn* from [Laube and Nebel 2010] is capable of performing a maximum likelihood analysis semi-automatically on Java bytecode. It works on annotated sources as shown below:

```
while( pfs() ) { // bfs dfs bfsrnd dfsrnd
  produceCost("Path", 1);
  augment(s, t);
}
```

This code fragment shows the main loop of the Ford-Fulkerson method. The annotation "`produceCost("Path", 1)`" indicates, that the performance parameter denoted "Path" is associated with this position in the source code. It allows us to keep track of the number of augmenting paths used while calculating the maxflow. All the elementary operations listed in Section 2 were annotated similarly.

The running time depends on the number of augmenting paths needed to find a maxflow and the time needed to find each augmenting path. The results for the worst-case from [Sedgewick 2003] are:

THEOREM 3.1. *The number of augmenting paths is at most $|V| \cdot M$ for any implementation of the Ford-Fulkerson method, it is at most $|V| \cdot |E|$ for shortest-augmenting-path heuristic of the Ford-Fulkerson method and at most $2|E| \log(M)$ for maximum-capacity-augmenting-path heuristic of the Ford-Fulkerson method. The largest edge weight is denoted by M.*

To illustrate the gap between theses bounds and the actual performance of the implementation from [Sedgewick 2003]. We note that on a grid graph with $|V| = 400$, $|E| = 760$ and $M = 100$ the bound for the number of augmenting path is as high as $40\,000$ while we actually observe only around 35 augmenting paths with bfs. For pfs the bound is $7\,000$ but we only observe around 5 paths on average. As we call graphs *sparse* when $|E| = \mathcal{O}(|V|)$ holds and *dense* when $|E| = \Theta(|V|^2)$ holds we have the following corollary.

COROLLARY 3.2. *Assuming that an augmenting path can be found in $\mathcal{O}(|E|)$ the time to find a maxflow is $\mathcal{O}(|E| \cdot |V| M)$ which is $\mathcal{O}(|V|^2 M)$ for sparse networks. In case of the shortest-augmenting-path heuristic the bounds is $\mathcal{O}(|E| \cdot |V||E|)$ which is $\mathcal{O}(|V|^3)$ for sparse networks. Note that the general bound is sharper for small M and sparse networks. For the maximum-capacity-augmenting-path heuristic we have the bound $\mathcal{O}(|E| \log(|V|) \cdot 2|E| \log(M))$ which is $\mathcal{O}(|V|^2 \log(|V|) \log(M))$ for sparse networks.*

As explained in section A.6.3 the maximum likelihood training runs the analyzed algorithm on sufficient large sets of inputs. In a common analysis a uniform distribution of *all* inputs is assumed for the derivation of an average-case result. With graphs it is a priori not clear how the uniformity assumption should be incorporated. There are at least two choices: Gilbert's [Gilbert 1959] random graph model $G_{n,p}$ picks each edge independently with probability p or the random graph model of Erdős and Rényi $G_{n,m}$ were all graphs with m edges equally probable [Erdos and Rényi 1959].

The maximum likelihood analysis method does not make or need an assumption about the distribution of the inputs. Whether given a sample of real world inputs or a set of generated inputs it allows to derive an average case result for this particular

family of inputs. In this sense the maximum likelihood analysis method supports an average-case analysis for *non-uniform* distributions of the inputs. Moreover we are not required to specific the possibly unknown distribution in mathematical terms. The maximum likelihood training captures the distribution in the trace grammar implicitly.

4. GRID GRAPHS

We now define the square grid graphs and the distributions of the weights on the edges that we use in our experiments for the maximum likelihood training thereby creating models of the behavior of the Ford-Fulkerson method. Formally we have:

Definition 4.1. A *square grid graph* $G_k = (V, E)$ is a graph on $V = \{0, \dots, k-1\}^2$ with the set of edges $E = \big\{\{(i,j),(i',j')\} \mid |i-i'| + |j-j'| = 1\big\}$.

If $G = (W, F)$ denotes a graph *on* W, we write $V(G) = W$ for its set of vertices and $E(G) = F$ for its set of edges. Following [Vizing 1963] we have the following definition:

Definition 4.2. We denote the *cartesian product* of the graphs G and H by $G \,\square\, H$ and define the resulting graph on the vertex set $V(G) \times V(H)$, the cartesian product of the vertex sets $V(G)$ and $V(H)$. Two vertices $(u,u'),(v,v') \in V(G \,\square\, H)$ are adjacent if and only if

(1) $u = v$ and u' is adjacent to v' in H, or
(2) $u' = v'$ and u is adjacent to v in G.

The box symbol of this operation resembles its effect on two path graphs.

Definition 4.3. A *path graph* P_i is a tree without branches on i vertices.

Definition 4.4. A two-dimensional *grid graph* $G_{j,k}$ is defined as $G_{j,k} := P_j \,\square\, P_k$. It is called square if $j = k$.

By defining the two path graphs P_j and P_k on the vertex sets $[1..j]$ and $[1..k]$[2] respectively, the pairs in $V(P_j \,\square\, P_k)$ can serve as coordinates for the vertices they represent, yielding a grid-like embedding in the plane.

LEMMA 4.5. *The grid graph $G_{j,k}$ has the following properties:*

$$|V(G_{j,k})| = j \cdot k, \qquad |E(G_{j,k})| = (j-1)k + j(k-1) = 2j \cdot k - j - k,$$

there are 4 *vertices of degree* 2*,* $2k + 2j - 8$ *vertices of degree* 3 *and* $(k-2)(j-2)$ *vertices of degree* 4*.*

The Proof is not difficult and found in the appendix. As $2 \cdot |E(G_{j,k})|$ is just the total degree sum of the graph, the average or expected degree of a vertex $v \in V(G_{j,k})$ is

$$\mathbb{E}\big[\deg(v)\big] = \frac{2 \cdot |E(G_{j,k})|}{|V(G_{j,k})|} = \frac{4j \cdot k - 2j - 2k}{j \cdot k} = 4 - 2\left(\frac{1}{j} + \frac{1}{k}\right).$$

COROLLARY 4.6. *The square grid graph G_k has the following properties:*

$$|V(G_k)| = k^2, \qquad |E(G_k)| = 2k^2 - 2k,$$

there are 4 *vertices of degree* 2*,* $4k - 8$ *vertices of degree* 3 *and* $(k-2)^2$ *vertices of degree* 4 *and the average or expected degree of some vertex* $v \in V(G_k)$ *is* $\mathbb{E}\big[deg(v)\big] = 4(1 - 1/k)$*.*

With k being the "side-length" of the square grid graph G_k, we let $|V(G_k)| = k^2 = n$ in Corollary 4.6, thus $|E(G_{\sqrt{n}})| = 2n - 2\sqrt{n}$, hence square grid graphs are sparse.

[2] $[a..b]$ with $a, b \in \mathbb{N}$ and $a \leq b$ is the *integer interval*, thus $[a..b] := \{a, a+1, \dots, b\}$

5. RANDOM GEOMETRIC GRAPHS

We now define the random geometric graphs and the distributions of the weights on the edges that we use in the maximum likelihood training thereby creating models of the behavior of the Ford-Fulkerson method. Following [Penrose 2003] we define:

Definition 5.1. A *geometric graph* $G_r(\mathcal{X}) = (\mathcal{X}, E)$ is a undirected graph on $\mathcal{X} \subseteq \mathbb{R}^d$ with the set of edges $E = \big\{\{x, x'\} \mid \|x - x'\| \leq r\big\}$. Where $\|\cdot\|$ is a norm on $\mathbb{R}^d$ and r an additional parameter.

For particular choices of d the graphs have special names, e.g. $d = 1$ *interval graphs*, $d = 2$ *disk graphs* or *proximity graphs*. We restrict ourselves to the case where $d = 2$ and we use the Euclidean norm on $\mathbb{R}^2$. We are not interested in graphs on a specific point set $\mathcal{X}$, we rather use a probabilistic model. Let f be some specified probabilistic density function on $\mathbb{R}^2$ and let $X_1, X_2, \ldots$ denote independent and identically distributed 2-dimensional random variables with common density f. Let $\mathcal{X}_n = \{X_1, X_2, \ldots, X_n\}$. We then call $G_r(\mathcal{X}_n)$ a *random geometric graph*. Again we restrict ourselves to a uniform distribution on $[0, 1]^2$.

If $G = (W, F)$ denotes a graph *on* W, we write $V(G) = W$ for its set of vertices and $E(G) = F$ for its set of edges.

THEOREM 5.2. *The average degree of an arbitrary node in a random geometric graph $v \in V\big(G_r(\mathcal{X}_n)\big)$ on point set $\mathcal{X}_n$ uniformly distributed over the unit square is:*

$$\mathbb{E}\big[\deg(v)\big] = n \cdot p(r) \qquad \textit{with} \qquad p(r) = r^2\left(\pi + \frac{r(3r - 16)}{6}\right).$$

The proof is long but not difficult. We just have to integrate the different contributions over the unit square to get the average size of the neighborhood $p(r)$. The proof is included in the appendix.

COROLLARY 5.3. *The expected number of edges in a random geometric graph $G_r(\mathcal{X}_n)$ is*

$$\mathbb{E}\big[\big|E\big(G_r(\mathcal{X}_n)\big)\big|\big] = \frac{n}{2} \cdot \mathbb{E}\big[\deg(v)\big] = \frac{n^2 p(r)}{2} = \frac{n^2 r^2}{2}\left(\pi + \frac{r(3r - 16)}{6}\right).$$

As there are at most $\binom{n}{2} = \frac{n(n-1)}{2}$ edges in a graph comparing this to the expected number of edges

$$\lim_{n\to\infty} \frac{\frac{n^2 p(r)}{2}}{\frac{n(n-1)}{2}} = \lim_{n\to\infty} \frac{p(r)n}{n-1} = \lim_{n\to\infty} p(r)\frac{1}{1 - \frac{1}{n}} = p(r)$$

shows that asymptotically a fraction $p(r)$ of all possible edges are present. It is clear from the above, that random geometric graphs are dense.

6. SAMPLE INPUTS FOR EXPERIMENTS

For the sample inputs sets of square grid graphs G_k with "side-lengths" $k \in \{5, 10, 15, \ldots, 50, 51, 52, \ldots, 60\}$ are used. We call the four possible directions of traversal in the grid graph as north, east, west and south. Our graph generator ensures that the edges are always inserted into adjacency lists in the same order, which is east, west, north, south for no special reason.

Numbering the vertecies in row-major order starting with 0 in the lower left corner of the grid two locations for the *source* s and the *sink* t were considered:

(1) (llur) the lower left $s = 0$ and the upper right corner $t = k^2 - 1$,

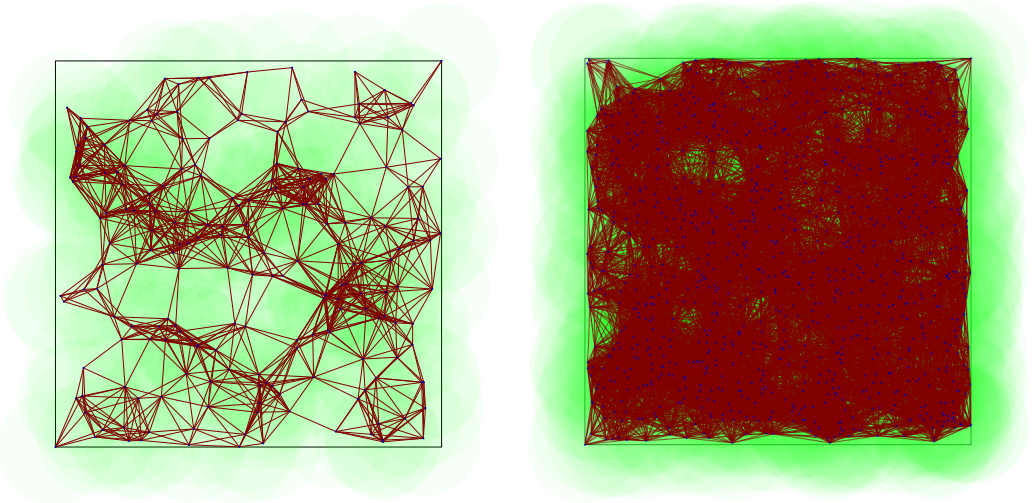

Fig. 1. Two random geometric graphs draw in the unit square. On the left with $|V| = 200$ vertices and $|V| = 1000$ vertices on the right. The shaded disks (green) with radius $r = 15/100$ represent the neighborhood of the vertices.

(2) (lmum) the lower middle $s = \lfloor \frac{k}{2} \rfloor$ and the upper middle vertex $t = k^2 - 1 - \lfloor \frac{k}{2} \rfloor$.

When k is even there is of course no vertex in the middle. Note that with $k = 2i$ the source s is in column i and the sink t is in column $4i^2 - 1 - i \bmod 2i = i - 1$.

In case of the random geometric graphs $G_r(\mathcal{X}_n)$ the sample inputs sets are built over random point sets $\mathcal{X}_n$ uniformly distributed in the unit square with sizes $n \in \{100, 200, \ldots, 2100\}$. The parameter r was arbitrarily set to $15/100$, to avoid crossing the distance between source and sink with too few edges. Computing $p(15/100) \approx 0.062$ shows that asymptotically 6.2% of all possible edges are present. Figure 1 shows two examples graphs of this sample input set.

Two settings for the *source* s and the *sink* t were considered:

(1) (llur) the lower left $s = (0, 0)$ and the upper right corner $t = (1, 1)$,
(2) (umlm) the upper middle $s = (1/2, 1)$ and the lower middle vertex $t = (1/2, 0)$.

They are always part of the point set. But as the random point sets may put the source and sink in different connected components, only point sets resulting in a single connected component with n vertices where used.

7. DISTRIBUTIONS

As the Ford-Fulkerson method may fail to terminate when real numbers are used as edge weights, we use three distributions of integers, namely:

(1) (unit) were all edges have capacity 1,
(2) (unif) the uniform discrete distribution were the capacities are independently drawn from the integer interval $[0..100]$ and
(3) (gauss) a shifted and truncated normal distribution were the capacity of an edge is chosen according to $250 - \lfloor 50 \cdot X \rfloor$ where $X \sim \mathcal{N}(0, 1)$.

Of course the Z-transformation $Z = \mu \pm \sigma X$ turns a standard normal random variable X into one with mean $\mu = 250$ and variance $\sigma^2 = 50^2$. As negative capacities are pointless we discard them and draw again whenever $250 - \lfloor 50 \cdot X \rfloor < 0$ is true. As shown below this shifts the average edge capacity slightly over $\mu = 250$.

LEMMA 7.1. *The average capacities of an edge $e \in E(G_k)$ are:*

$$\mathbb{E}\big[\mathrm{cap}_{\mathrm{unit}}(e)\big] = 1 \qquad \mathbb{E}\big[\mathrm{cap}_{\mathrm{unif}}(e)\big] = 50 \qquad \mathbb{E}\big[\mathrm{cap}_{\mathrm{gauss}}(e)\big] = \frac{501}{2} + \epsilon$$

with $\epsilon \ll 1$.

PROOF. The unit case is obvious and the uniform case follows quickly by calculating

$$\mathbb{E}\big[\mathrm{cap}_{\mathrm{unif}}(e)\big] = \frac{\sum_{i=0}^{100} i}{101} = \frac{1}{101}\frac{100(100+1)}{2} = \frac{100}{2} = 50.$$

The gauss case is not difficult but involves tedious calculations (included in the appendix) with truncated normal distributions resulting in

$$\mathbb{E}\big[\mathrm{cap}_{\mathrm{gauss}}(e)\big] = \frac{1}{\Phi(251/50)} \sum_{a=0}^{\infty} \Phi\left(\frac{250-a}{50}\right) = \frac{501}{2} + \epsilon$$

with $\epsilon \ll 1$ and $\Phi(x)$ denoting the cdf of the standard normal distribution. □

Let the *total capacity* of a vertex be the sum of the capacities of the edges incident with the vertex. Then we find

LEMMA 7.2. *The average total capacities of source s and sink t are:*

$\mathbb{E}\big[\mathrm{cap}(\cdot)\big]$	*unit*	*unif*	*gauss*
llur	$2 \cdot 1$	$2 \cdot 50$	$2 \cdot \frac{501}{2}$
lmum	$3 \cdot 1$	$3 \cdot 50$	$3 \cdot \frac{501}{2}$

PROOF. As the edges are independently weighted we have $\mathbb{E}[\mathrm{cap}(v)] = \deg(v) \cdot \mathbb{E}[\mathrm{cap}(e)]$ for $v \in V(G_k)$ and $e \in E(G_k)$. Corollary 4.6 and Lemma 7.1 provide the results. □

As the total flow originating at the source must "fit" into the sink, we expect the average maxflow to be lower than the average total capacity of the source. In fact we have the following lemma.

LEMMA 7.3. *The average maxflows in a square grid graph G_k are bounded by:*

$\mathbb{E}\big[\mathrm{mf}(\cdot)\big]$	*unit*	*unif*	*gauss*
llur	$2 \cdot 1$	$2 \cdot \frac{3350}{101}$	$2 \cdot 222.3$
lmum	$3 \cdot 1$	$3 \cdot \frac{3350}{101}$	$3 \cdot 222.3$

The proof of the lemma is omitted here and included in the appendix.

8. VERIFICATION

As explained in Section A.6.3 using the various sets of graphs as sample inputs for the maximum likelihood training of the grammar provides us with a number of models for the behavior of the Ford-Fulkerson method in the different settings. We denote the models by M_d^l where $d \in \{\mathrm{unit}, \mathrm{unif}, \mathrm{gauss}\}$ and $l \in \{\mathrm{llur}, \mathrm{lmum}\}$. The results are listed in Section 2.

As already pointed out many algorithms – and the same holds for those discussed in this paper – are not amenable to a theoretical average-case analysis. In particular when non-uniform distributions of the inputs are involved. The development of the maximum likelihood analysis technique in [Laube and Nebel 2010] was partly motivated by the hope to find methods for the analysis of algorithms, that allow us to make statements

Table VI. Observed and expected average parameters of the sample input sets for grid graphs.

	observed			expected		
average	unit	unif	gauss	unit	unif	gauss
edge capacity	1	49.9904	250.51	1	50	250.5
llur source/sink capacity	2	99.5	501.1	2	100	501
lmum source/sink capacity	3	150.0	751.9	3	150	751.5
llur maxflow	2	65.1	462.0	2	66.3	444.6
lmum maxflow	3	108.8	701.4	3	99.5	666.9

about the performance, in cases where a purely theoretical approach has little chance of success.

The maximum likelihood training provides us with a model for the algorithms behavior on the sample set of inputs. This is done by tuning the probabilities of a stochastic context free grammar as explained in Section A.6. This model is subsequently analyzed by generating function techniques which yield true results. The independent verification allows us to gain confidence that the model describes the algorithms behavior adequately.

By verifying simpler properties of grids graphs independently from the maximum likelihood analysis, as shown in the lemmata 7.1, 7.2 and 7.3 for grid graphs we are able to crosscheck the derived model and gain confidence in the results for other parameters obtained by the maximum likelihood analysis technique.

The average parameters noted in Table VI were observed over $79\,360\,000$ edges of the grid graph sample inputs. The expectations are from Lemma 7.1, 7.2 and 7.3 respectively.

For random geometric graphs we don't have similar results of simple parameters for crosschecking, as the random processes involved in the generation of random geometric graphs make a mathematical treatment considerably harder.

9. DISCUSSION

The performance of the strategies `dfs` and `bfs` with respect to the number of augmenting paths needed in the computation of the maxflow is mixed. As evident from Table I neither one clearly outperforms the other while both are beaten by `pfs`, that only needs a constant number of augmenting paths on grid graphs with the given capacity distributions. The introduction of random iterators to `dfs` and `bfs` reduces the number of augmenting paths in some cases however not greatly as indicated in the two rightmost column of Table I.

Regarding random geometric graphs the first thing to notice when looking at the results in Tables I–IV is that randomization has a stabilizing effect on the algorithms performance. Across the different models, graph search strategies and parameters we observe the same asymptotic complexity only the constants vary.

As explained in Section 2 the Tables II–IV give the average fraction of all vertices of the graph that were stored in the queue or stack per path, by dividing the total number of operations by the number of paths times the number of vertices n.

This allows the comparison of `dfsrnd` and `bfsrnd` to the strategies without random iterators (`dfs`, `bfs`). The observation is, that using random iterators leads to a minor improvement on the work per path for random geometric graphs only. Hence when the inputs are already formed by an underlying random process, as is the case with the random geometric graphs, introducing randomization a second time, via random iterators, has only a limited effect on the per path workload as expected.

For the average number of paths however the use of random iterators can lead to significant reductions, up to a third in the case of `dfs` for random geometric graphs as shown in Table I. This effect is easily explained by the fact that the saturation of the

edges changes after each augmenting path is added to the total flow. Without random iterators the graph search visits the neighbors in the same order. It thus explores the same part of the graph where the remaining capacities were lowered by the previous path. With every further path the average flow per path decreases until the whole area is saturated. This happens repeatedly until the maxflow is found. Overall more paths are necessary.

With random iterators every graph search explores different parts of the graph. Thus avoiding the previously visited areas where only a smaller remaining capacity is available. This allows more flow per path and thus fewer paths are necessary to reach the maximum flow.

This also indicates why the `pfs` graph search strategy has the lowest overall average number of augmenting paths. First it greedily chooses the maximum remaining capacity edge, thus aiming for a high flow per path and second our random process used in the generation of the graph placed the edges randomly in the adjacency lists. Searching the maximum remaining capacity edge is implicitly like using a random iterator. It exploits the randomness already present in the input.

We want to check further effects to gain confidence in the models created. In case of grid graphs Table II gives the average numbers of queue operations when `bfs` is used as a graph search strategy. In case of unit edge capacities ($M_{\text{unit}}^{\text{llur}}$) the whole graph is placed in the queue during every graph search phase. The intuitive explanation is that by the time it reaches the sink in the opposite corner of the square grid graph all other vertecies where closer and thus already in the queue. Changing the location of source and sink to the midpoints on opposite sides of the grid ($M_{\text{unit}}^{\text{lmum}}$) only causes about $3/4$ of all vertecies being pushed on the stack. This is due to the top left and top right corners regions of the grid, containing an $1/8$ of all vertecies each, being not pushed on the stack as the sink is reached before them.

Considering that `bfs` visits the vertecies in ever growing “circles”, centered in the source, hints why randomized iterators have little effect. They just change locally the order in which the neighbors are visited. Globally this only has limited effect since the closest neighbors are still visited first, only in different orders.

The influence of the choice of the starting locations is also evident for random geometric graphs from Table I when comparing the rows for models with the same distribution. Moving source and sink from the corners (llur) to the middle of the boundary (lumu) increases the area in which edges to neighbors are created from a quarter disk to half a disk. This factor 2 is present in the average number of paths as well.

Thus despite their inherent simplifications of the models they reflect the true behavior of the algorithms in many details.

Comparing the Tables II and III with respect to the number of vertices remaining on the stack (`dfs`) or in the queue (`bfs`) one first notices that on average the whole random geometric graph is pushed on the stack or put in the queue during every graph search phase. Considering that `bfs` visits the vertices in ever growing “circles”, centered in the source, it is not surprising, that it reaches all closer vertices before the sink on the other corner (llur) or edge (lmum) of the graph.

In case of `dfs` all vertices are pushed on the stack on average. The left graph in Figure 2 shows the spanning tree of a single `dfs` graph search illustrating this observation. This is due to the random geometric graphs being dense, as all vertices are placed in the unit square, and the way the graph search strategies from [Sedgewick 2003] are implemented. They always place all adjacent vertices in the data structure upon visiting a new vertex while avoiding to insert the same vertex twice.

This is not problematic for `bfs` as it visits all its neighbors, removing them, before proceeding. As indicated by the number of `gets`, that is only slightly less than the

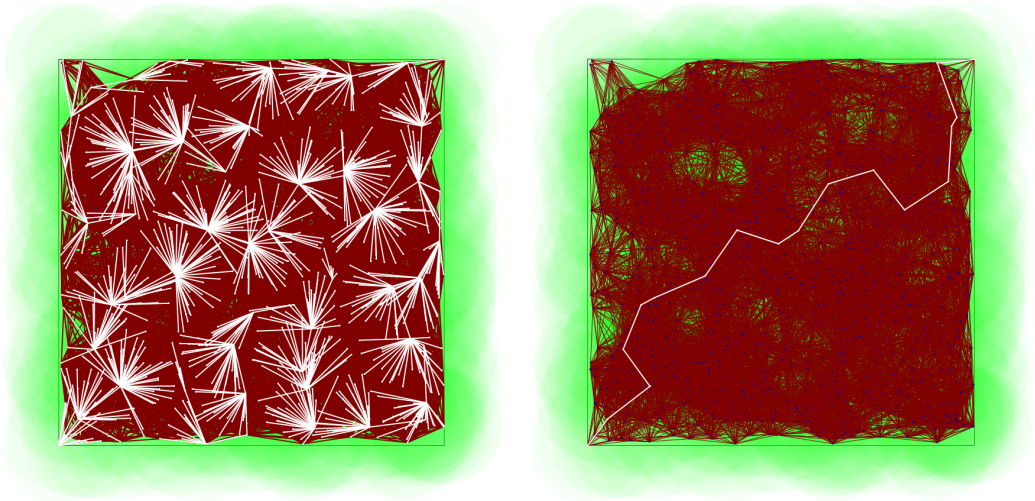

Fig. 2. The same graph as in Figure 1 is shown. On the left with the spanning tree of a single `dfs` graph search in white and on the right the resulting augmenting path is shown in white.

number of `puts`, therefore the `bfs` strategy removes almost all vertices from the queue while searching for an augmenting paths.

In case of `dfs` however a larger fraction remains on the stack when the augmenting paths is found. Only about 27% of the vertices have been removed from the stack. This too is not surprising, once again it is a check that the models represents correctly the true behavior of the algorithm. But it leads to an immediate suggestion for improving the `dfs` implementation, by changing what is stored on the stack. Instead of placing all the adjacent vertices on the stack a reference to the last entry in the adjacency list that was looked at would reduce the number of pushes considerably, thereby saving time and space, by avoid the needless pushes.

Looking at Table III we find that `dfs` pushes the whole grid graph on the stack in model $M_{\text{unit}}^{\text{llur}}$ too. The reason is the order of the edges in the adjacency list, that is east, west, north and south. However as they are placed on a stack the order is reversed when they are removed from the stack. In the $M_{\text{unit}}^{\text{llur}}$ model `dfs` must connect the lower left corner with the upper right corner while preferring the directions north and south over east and west. This causes the path to meander up and down through the whole grid graph as it proceeds from left to right. This pattern has a high potential for the randomized iterators to break the problematic order of the neighbors. Comparing the fractions of `pushes` in Table III reductions between 30% to 39% are possible by replacing `dfs` with `dfsrnd`.

This shows how important the order of the neighbors is. Imagine the order being permuted every time a graph is read into memory and saved back due to careless data management. The runtime would fluctuate for no apparent reason for the same graph.

However the order of the neighbors is not always problematic. If the source and sink are in the same column, as in the $M_{\text{unit}}^{\text{lmum}}$ model, the preference for going north leads directly to the sink thus in this case only 29% of the vertecies are pushed on the stack on average. Here the use of randomized iterators actually increases the number of `pushes` by a factor of slightly less than 2.4.

Comparing the numbers of operations for `dfs` and `bfs` shows that the number of `gets` is only slightly less than the number of `puts`, therefore the `bfs` strategy removes almost all vertecies from the queue while searching for an augmenting paths, compared to `dfs`

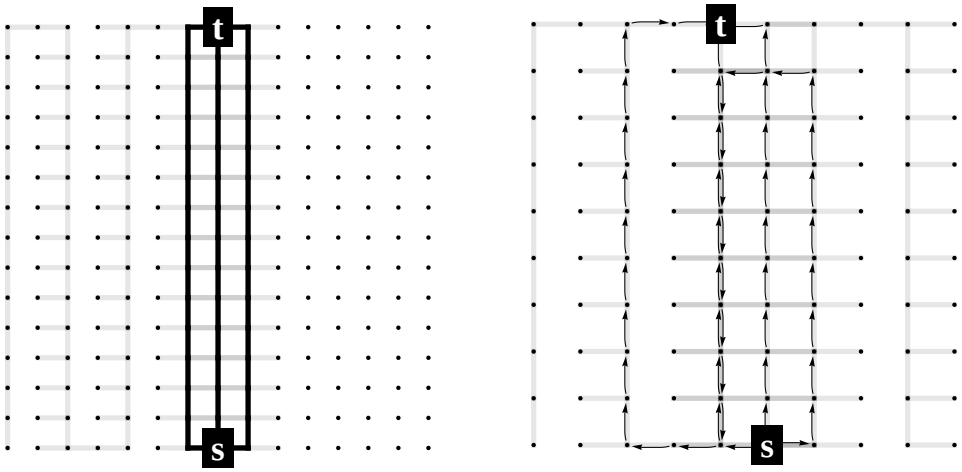

Fig. 3. The model $M^{\text{lmum}}_{\text{unit}}$ with $k = 15$ on the left and $k = 10$ and on the right. Grey edges were visited during the search for the three augmenting paths shown as solid black lines on the left and as black arrows on the right.

where a larger fraction remains on the stack when the augmenting paths is found. This too is not surprising, once again it is a check that the models represents correctly the true behavior of the algorithm.

As announced in Section 4 the choice of the location of source and sink has a drastic effect on the performance of the graph search strategies, dfs in particular. An oversight in the calculation of the location of source and sink vertecies in the `lmum` setting caused the source and sink to be placed in two adjacent "columns" in the grid when the "side-length" k of the grid graph G_k is even, as explained in the previous section. In the case of unit edge capacities $M^{\text{lmum}}_{\text{unit}}$ Figure 3 illustrates the odd-even effect.

On the left the three augmenting paths head straight north. The grey edges are the ones that where explored during the graph search, leading to slightly under 59% of the vertecies being pushed on the stack. On the right the three augmenting paths are shown using arrows because the third path redirects some of the flow of the second one. Here the whole graph is explored. As the behavior of the algorithm alternates between the two cases for odd and even side-lengths k used to create the $M^{\text{lmum}}_{\text{unit}}$ model the average is roughly $(1 + 0.59)/2 \approx 0.8$. As the augmenting paths are determined independently of each other some vertecies are pushed more often than others we have overall the $0.86n$ reported in Table III. The odd-even effect is present in the $M^{\text{lmum}}_{\text{unit}}$ model too as can be seen in Figure 4 for the total number of paths and pushes. The two functions plotted on the right are $0.297n \log^2(n)$ and $0.175n \log^2(n)$ indicating that odd inputs incur roughly 60% more pushes.

The results for `pfs` in Table IV show that the models are not sufficiently trained to reflect seldom events properly. Table I lists only around 3 to 5 paths on average for `pfs` independently of the size of the graph. Diving by those small constants yields fractions larger than 1 in Table IV, which are of course pointless as vertecies are only removed after the priority queue is initialized with all vertecies in every search phase. For the `lower` operation fractions larger than 1 are possible. This indicates that on average the priority was updated more than once.

Table V presents a different point of view. As explained in Section 2 an edge is considered visited if both its vertecies are removed from the stack or queue. The average fraction of edges visited per path as listed in Table V shows nicely the stabilizing effect

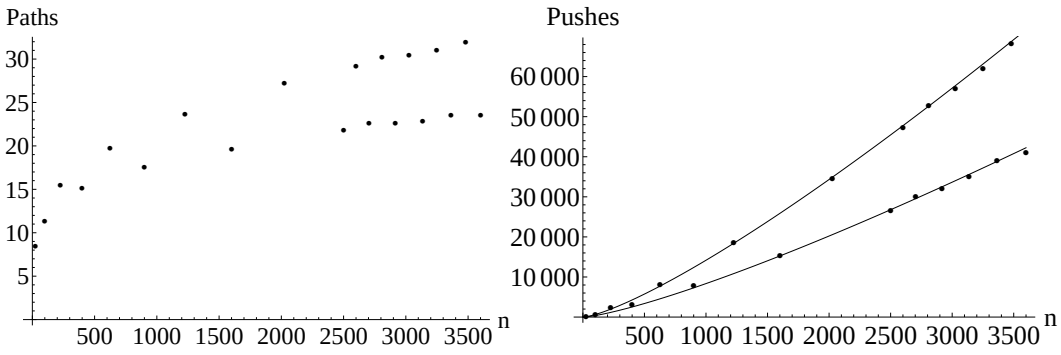

Fig. 4. Total number of paths and pushes for dfs in the $M_{\text{unit}}^{\text{lmum}}$ model.

on the performance that the random iterators in case of dfsrnd have. The fraction for dfsrnd is around 0.25 across the different models. As discussed above bfs does not profit from the use of randomized iterators, this is confirmed from the visited edges point of view as well.

Considering the above, our recommendation for a practitioner would be to use the pfs search strategy on random geometric graphs. The average number of paths is the least of the search strategies considered here, as it benefits from the randomness in the inputs. As the implementation of dfs is clearly simpler than the implementation of pfs it would be interesting to measure how big the impact of the suggested improvements from above is.

And for grid graphs the dfsrnd search strategy is a good choice. On average the smallest number of edges is involved and the number of paths is low, sometimes even a constant albeit not as low as for pfs. However the implementation of dfs is clearly simpler than the implementation of pfs.

As there is no shortage of other graph classes like random graphs, random geometric graphs, small-world graphs, complete graphs, etc. extending the study to include different inputs is one idea for further research, as well as other graph search strategies (e.g. the A*-heuristic if a suitable distance function can be provided). Perhaps replacing the queue in bfs by a randomized queue is equally effective as the use of the randomized iterators in dfs? Meta graph search strategies are used by practitioners, where a dfs run is started in the source and a second dfs run begins in the sink and the two runs are interleaved until they meet. We believe that a traditional average-case analysis of such a meta graph search strategy would be challenging. With the maximum likelihood method used in this study models for the average-case behavior of such a meta graph search strategy could be obtained and analyzed semi-automatically.

The maximum likelihood method itself is subject to further research as well. The performance of graphs is typically measured in $|V|$ and $|E|$. The grid graphs allowed us to set $|V| = n$ and use $|E| = 2(n - \sqrt{n})$. Extending the maximum likelihood method to allow multiple size measures in a single analysis would be a good enhancement. The average-case analysis of string matching algorithms where the length of the text and the length of the pattern has an influence on the performance immediately comes to ones mind.

10. CONCLUSION

In conclusion original results for multiple Ford-Fulkerson implementations were obtained by a maximum likelihood analysis. Being able to rediscover the stabilizing effect of randomization for average-case results as well as checking simple cases of

the algorithm's behavior by inspection verifies that the probability model derived from experiments reflects the true behavior of the algorithms well enough to provide predictions that are useful for practitioners and guide further theoretical work. For example knowing that the number of paths is a constant and therefore does not depend on the size of the input is a valuable insight when starting an analysis.

APPENDIX

A.1. Grid graph properties – Proof of Lemma 4.5

The grid graph $G_{j,k}$ has the following properties:

$$|V(G_{j,k})| = j \cdot k, \qquad |E(G_{j,k})| = (j-1)k + j(k-1) = 2j \cdot k - j - k,$$

there are 4 vertices of degree 2, $2k + 2j - 8$ vertices of degree 3 and $(k-2)(j-2)$ vertices of degree 4.

PROOF. Assume without loss of generality, that the underlying path graphs P_j and P_k of the grid graph $G_{j,k} = P_j \,\square\, P_k$ are not only defined on $[1..j]$ and $[1..k]$ respectively, but the elements of the edge sets have the form $(i, i+1)$. According to the definition of the grid graph its edges then have the form $\big\{(u,i),(u,i+1)\big\}$ or $\big\{(i,v),(i+1,v)\big\}$, $u \in [1..k], v \in [1..j]$, intuitively the vertical and horizontal edges of the grid graph.

The $2\cdot(k-1+j-1)$ vertices on the "edges" of the grid are of the form $(1,\bullet),(k,\bullet),(\bullet,j)$ and $(\bullet,1)$. Four of them are on two "edges" of the grid: $(1,1),(1,j),(k,j)$ and $(k,1)$ thus in the "corners" of the grid and have degree 2, as there are only two ways to increase or decrease each element of the pairs and stay within the ranges $[1..k]$ and $[1..j]$, the other have 3 degrees of freedom. The remaining $(k-2)(j-2)$ inner vertices have 4 degrees of freedom.

Thus $|V(G_{j,k})| = 4+2\cdot(k-2+j-2)+(k-2)(j-2) = 4+2k+2j-8+jk-2j-2k+4 = j\cdot k$ and counting each edge from both ends yields

$$2 \cdot |E(G_{j,k})| = 4 \cdot 2 + 3 \cdot 2 \cdot (k-2+j-2) + 4 \cdot (k-2)(j-2) = 4j \cdot k - 2j - 2k.$$

□

A.2. Random geometric graph properties – Proof of Theorem 5.2

The average degree of an arbitrary node in a random geometric graph $v \in V\big(G_r(\mathcal{X}_n)\big)$ on point set $\mathcal{X}_n$ uniformly distributed over the unit square is:

$$\mathbb{E}\big[\deg(v)\big] = n \cdot p(r) \qquad \textit{with} \qquad p(r) = r^2\left(\pi + \frac{r(3r-16)}{6}\right).$$

The expected degree of a vertex v depends on its position in the unit square. In the center $[r, 1-r]^2$ square we have with $r = 15/100$

$$\mathbb{E}[\deg(v)] = n\frac{\pi r^2}{1} \approx 0.071n$$

as this fraction of all n nodes in the disk around node v are connected to v. This is less near the boundary as no full disk with radius $r = 15/100$ fits in there. On the boundary we have

$$\mathbb{E}[\deg(v)] = n\frac{\pi r^2}{2} \approx 0.035n$$

as there is only half a disk possible. In the corner only a quarter disk fits, thus

$$\mathbb{E}[\deg(v)] = n\frac{\pi r^2}{4} \approx 0.018n.$$

Thus to proof the theorem we have to determine the average size of the neighborhood including the boundary effects when the vertex v is near the boundaries of the unit square.

PROOF. To do this we integrate over the unit square to determine the average size of the neighborhood $p(r)$. Exploiting the symmetry of the unit square, integrating over a single quadrant $[0, 1/2]^2$ is enough:

$$p(r) = 4 \cdot \int_0^{\frac{1}{2}} \int_0^{\frac{1}{2}} A(r, x, y)\, dxdy.$$

The contribution $A(x, y, r)$ obviously varies with the location (x, y) and the radius r of the neighborhood. We can distinguish four rectangles in the quadrant by splitting it horizontally and vertically in the point (r, r). The large square $[r, 1/2]^2$ allows for a full disk at all positions. In the two rectangles $[0, r] \times [r, 1/2]$ and $[r, 1/2] \times [0, r]$ a segment of the disk is cut off by the boundary. And on the small square $[0, r]^2$ this happens on both boundaries. Thus we arrive at

$$p(r) = 4 \cdot \left(\int_r^{\frac{1}{2}} \int_r^{\frac{1}{2}} A(r)\, dxdy + 2 \cdot \int_0^r \int_r^{\frac{1}{2}} A(r, y)\, dxdy + \int_0^r \int_0^r A(r, x, y)\, dxdy. \right) \quad (1)$$

The first integral is easily solved as the contribution does not depend on the location (x, y).

$$\int_r^{\frac{1}{2}} \int_r^{\frac{1}{2}} A(r)\, dxdy = A(r) \int_r^{\frac{1}{2}} \int_r^{\frac{1}{2}} 1\, dxdy = \pi r^2 \int_r^{\frac{1}{2}} \int_r^{\frac{1}{2}} 1\, dxdy = \pi r^2 \cdot \left(\frac{1}{2} - r \right)^2.$$

Thus the contribution πr^2 is weighted with the area of the larger square $(1/2 - r)^2$.

The two rectangles are easy too, as the contribution only depends on the change of the location in one dimension.

$$\begin{aligned} 2 \cdot \int_0^r \int_r^{\frac{1}{2}} A(r, y)\, dxdy &= 2 \cdot \int_0^r A(r, y) \int_r^{\frac{1}{2}} 1\, dxdy \\ &= 2 \cdot \int_0^r A(r, y) \left(\frac{1}{2} - r \right) dy = (1 - 2r) \int_0^r A(r, y)\, dy. \end{aligned}$$

The contribution $A(r, y)$ is just the area of disk minus the segment that protrudes over the boundary by $r - y$.

$$A(r, y) = A_\circ(r) - A_{SG}(r, y) = \pi r^2 - A_{SG}(r, y).$$

The area of the segment $A_{SG}(r, y)$ is just the area of the sector of the disk less the equal sided triangle formed by the chord and the radii whose height is y when the segment protrudes by $r - y$. Thus

$$A_{SG}(r, y) = A_{SK}(r, y) - A_\Delta(r, y),$$

$$A_\Delta(r, y) = \frac{sy}{2}.$$

The length of half the chord ist just

$$\frac{s}{2} = \sqrt{r^2 - y^2}$$

by Pythagoras' theorem. Accordingly

$$A_\Delta(r, y) = y \cdot \sqrt{r^2 - y^2}.$$

The area of the sector $A_{SK}(r,y)$ depends on its apex angle α:

$$A_{SK}(r,y) = \pi r^2 \frac{\alpha}{2\pi} = \frac{\alpha}{2} r^2.$$

We have to express this angle by y and r. Another application of Pythagoras' theorem shows that half the angle is just $\frac{\alpha}{2} = \arccos\left(1 - \frac{r-y}{r}\right)$ yielding

$$\begin{aligned} A_{SG}(r,y) &= r^2 \arccos\left(1 - \frac{r-y}{r}\right) - y \cdot \sqrt{r^2 - y^2} \\ &= r^2 \arccos\left(\frac{y}{r}\right) - y \cdot \sqrt{r^2 - y^2}. \end{aligned}$$

The final contribution is

$$\begin{aligned} A(r,y) &= \pi r^2 - \left(r^2 \arccos\left(\frac{y}{r}\right) - y \cdot \sqrt{r^2 - y^2}\,\right) \\ &= r^2 \left(\pi - \arccos\left(\frac{y}{r}\right)\right) + y \cdot \sqrt{r^2 - y^2}, \end{aligned}$$

and integrating over it yields

$$\begin{aligned} (1-2r)\int_0^r A(r,y)\,dy &= (1-2r)\left[\int_0^r r^2\left(\pi - \arccos\left(\frac{y}{r}\right)\right) + y \cdot \sqrt{r^2 - y^2}\,dy\right] \\ &= (1-2r)\left[r^2 \int_0^r \pi - \arccos\left(\frac{y}{r}\right)\,dy + \int_0^r y \cdot \sqrt{r^2 - y^2}\,dy\right] \\ &= (1-2r)\left[r^2\big(\pi r - r\big) + \frac{r^3}{3}\right] \\ &= (1-2r)\left[r^3(\pi - 1) + \frac{r^3}{3}\right] \\ &= (1-2r) r^3 \left((\pi - 1) + \frac{1}{3}\right) \\ &= (1-2r) r^3 \left(\pi - \frac{2}{3}\right) \\ &= 2\left(\frac{1}{2} - r\right) r \cdot r^2 \left(\pi - \frac{2}{3}\right). \end{aligned}$$

This is again of the form where the contribution $r^2(\pi - 2/3)$ is weighted with area of the two rectangles $r(1/2 - r)$. The antiderivatives used above appear again in the next cases and are derived afterwards.

The small square is more difficult to solve. The easier case is encountered when the relation $\sqrt{x^2 + y^2} \geq r$ is true. The segments of the disk protruding over two boundaries do not overlap and they can simply be subtracted as done before. However when $\sqrt{x^2 + y^2} < r$ holds the overlapping part would be subtracted twice. So we add a correction term. Separating the two cases gives

$$\int_0^r \int_0^r A(r,x,y)\,dydx = \int_0^r \left[\int_0^{\sqrt{r^2 - x^2}} A_<(r,x,y)\,dy + \int_{\sqrt{r^2 - x^2}}^r A_\geq(r,x,y)\,dy\right] dx. \quad (2)$$

The integrands are

$$\begin{aligned} A_\geq(r,x,y) &= \pi r^2 - A_{SG}(r,x) - A_{SG}(r,y) \\ A_<(r,x,y) &= A_\geq(r,x,y) + K(r,x,y) \end{aligned} \quad (3)$$

$$= \pi r^2 - A_{SG}(r,x) - A_{SG}(r,y) + K(r,x,y).$$

We can use (3) to rewrite (2)

$$\begin{aligned}\int_0^r \int_0^r A(r,x,y)\,dydx &= \int_0^r \left[\int_0^{\sqrt{r^2-x^2}} A_{\geq}(r,x,y) + K(r,x,y)\,dy \right.\\ &\quad \left. + \int_{\sqrt{r^2-x^2}}^r A_{\geq}(r,x,y)\,dy\right] dx\\ &= \int_0^r \left[\int_0^{\sqrt{r^2-x^2}} K(r,x,y)\,dy + \int_0^r A_{\geq}(r,x,y)\,dy\right] dx.\end{aligned}$$

To determine $A_{\geq}(r,x,y)$ we already know all the necessary expressions and get:

$$\begin{aligned}A_{\geq}(r,x,y) &= \pi r^2 - r^2 \arccos\left(\frac{y}{r}\right) + y\cdot\sqrt{r^2-y^2} - r^2\arccos\left(\frac{x}{r}\right) + x\cdot\sqrt{r^2-x^2}\\ &= r^2\left(\pi - \arccos\left(\frac{y}{r}\right) - \arccos\left(\frac{x}{r}\right)\right) + y\cdot\sqrt{r^2-y^2} + x\cdot\sqrt{r^2-x^2}\\ &= r^2\left(\arcsin\left(\frac{y}{r}\right) + \arcsin\left(\frac{x}{r}\right)\right) + y\cdot\sqrt{r^2-y^2} + x\cdot\sqrt{r^2-x^2}.\end{aligned}$$

Thus it remains to determine the correction term $K(r,x,y)$. It describes a part of the area of a disk that looks like a segment offset from the center. We start the derivation with a full disk. In Cartesian coordinates we have

$$\int_{-r}^r \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} 1\,dydx = \pi r^2,$$

because when moving along the diameter $[-r,r]$ we can only move as far as $[-\sqrt{r^2-x^2}, \sqrt{r^2-x^2}]$ in the perpendicular direction to stay with in the disk. Halving that interval gives a semi disk

$$\int_{-r}^r \int_0^{\sqrt{r^2-x^2}} 1\,dydx = \frac{\pi r^2}{2}.$$

Halving the other direction too yields a quarter disk

$$\int_0^r \int_0^{\sqrt{r^2-x^2}} 1\,dydx = \frac{\pi r^2}{4}.$$

Limiting the integral further by cutting of stripes with the width to $0 < a < r$ in one and $0 < b < r$ in the other direction we get:

$$K(r,a,b) = \int_a^{\sqrt{r^2-b^2}} \int_b^{\sqrt{r^2-x^2}} 1\,dydx.$$

Obviously $r > \sqrt{a^2+b^2}$ has to be true otherwise the area would be empty. Thus

$$\begin{aligned}K(r,a,b) &= \int_a^{\sqrt{r^2-b^2}} \sqrt{r^2-x^2} - b\,dx\\ &= \left[\frac{r^2}{2}\arcsin(x/r) + \frac{x}{2}\sqrt{r^2-x^2} - bx\right]_a^{\sqrt{r^2-b^2}}\\ &= \left(\frac{r^2}{2}\arcsin\left(\sqrt{r^2-b^2}/r\right) + \frac{\sqrt{r^2-b^2}}{2}\sqrt{r^2-\sqrt{r^2-b^2}^2} - b\sqrt{r^2-b^2}\right)\end{aligned}$$

$$
\begin{aligned}
&- \left(\frac{r^2}{2} \arcsin(a/r) + \frac{a}{2}\sqrt{r^2-a^2} - ba \right) \\
&= \frac{r^2}{2} \arcsin\left(\frac{\sqrt{r^2-b^2}}{r} \right) + \frac{1}{2}\sqrt{r^2-b^2}\sqrt{r^2-r^2+b^2} - b\sqrt{r^2-b^2} \\
&- \frac{r^2}{2} \arcsin\left(\frac{a}{r}\right) - \frac{a}{2}\sqrt{r^2-a^2} + ba \\
&= ba - \frac{a}{2}\sqrt{r^2-a^2} + \sqrt{r^2-b^2}\frac{\sqrt{b^2}}{2} - b\sqrt{r^2-b^2} \\
&+ \frac{r^2}{2} \arcsin\left(\frac{\sqrt{r^2-b^2}}{r} \right) - \frac{r^2}{2} \arcsin\left(\frac{a}{r}\right) \\
&= ba - \frac{a}{2}\sqrt{r^2-a^2} + \frac{b}{2}\sqrt{r^2-b^2} - b\sqrt{r^2-b^2} \\
&+ \frac{r^2}{2} \left(\arcsin\left(\frac{\sqrt{r^2-b^2}}{r} \right) - \arcsin\left(\frac{a}{r}\right) \right) \\
&= ba - \frac{a}{2}\sqrt{r^2-a^2} - \frac{b}{2}\sqrt{r^2-b^2} + \frac{r^2}{2} \left(\arcsin\left(\frac{\sqrt{r^2-b^2}}{r} \right) - \arcsin\left(\frac{a}{r}\right) \right).
\end{aligned}
$$

Solving the remaining integrals

$$
\begin{aligned}
\int_0^r A_{\geq}(r,x,y)\,dy &= \int_0^r r^2 \left(\arcsin\left(\frac{y}{r}\right) + \arcsin\left(\frac{x}{r}\right) \right) + y\cdot\sqrt{r^2-y^2} + x\cdot\sqrt{r^2-x^2}\,dy \\
&= r^2 \arcsin\left(\frac{x}{r}\right)[y]_0^r + r^2 \int_0^r \arcsin\left(\frac{y}{r}\right)\,dy + x\sqrt{r^2-x^2}[y]_0^r \\
&\quad + \int_0^r y\sqrt{r^2-y^2}\,dy \\
&= r^3 \arcsin\left(\frac{x}{r}\right) + r^2 \left[y \arcsin\left(\frac{y}{r}\right) + \sqrt{r^2-y^2} \right]_0^r + rx\sqrt{r^2-x^2} \\
&\quad + \left[-\frac{1}{3}(r^2-y^2)^{3/2} \right]_0^r \\
&= r^3 \arcsin\left(\frac{x}{r}\right) + \frac{r^3\pi}{2} - r^3 + rx\sqrt{r^2-x^2} + \frac{r^3}{3} \\
&= r^3 \arcsin\left(\frac{x}{r}\right) + r^3\left(\frac{\pi}{2} - \frac{2}{3}\right) + rx\sqrt{r^2-x^2}
\end{aligned}
$$

and

$$
\begin{aligned}
\int_0^{\sqrt{r^2-x^2}} K(r,x,y)\,dy &= \int_0^{\sqrt{r^2-x^2}} xy - \frac{x}{2}\sqrt{r^2-x^2} - \frac{y}{2}\sqrt{r^2-y^2} \\
&\quad + \frac{r^2}{2}\left(\arcsin\left(\frac{\sqrt{r^2-y^2}}{r} \right) - \arcsin\left(\frac{x}{r}\right) \right) dy \\
&= x\left[y^2/2\right]_0^{\sqrt{r^2-x^2}} - \frac{x}{2}\sqrt{r^2-x^2}[y]_0^{\sqrt{r^2-x^2}} - \frac{1}{2}\int_0^{\sqrt{r^2-x^2}} y\sqrt{r^2-y^2}\,dy
\end{aligned}
$$

$$
\begin{aligned}
&+ \frac{r^2}{2}\int_0^{\sqrt{r^2-x^2}} \arcsin\left(\frac{\sqrt{r^2-y^2}}{r}\right) dy - \frac{r^2}{2}\arcsin\left(\frac{x}{r}\right)[y]_0^{\sqrt{r^2-x^2}} \\
&= \frac{x}{2}(r^2-x^2) - \frac{x}{2}(r^2-x^2) - \frac{1}{2}\left[-\frac{1}{3}(r^2-y^2)^{3/2}\right]_0^{\sqrt{r^2-x^2}} \\
&- \frac{r^2}{2}\sqrt{r^2-x^2}\arcsin\left(\frac{x}{r}\right) \\
&+ \frac{r^2}{2}\left[y\arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right) + \sqrt{r^2-y^2}\right]_0^{\sqrt{r^2-x^2}} \\
&= \frac{x^3}{6} - \frac{r^3}{6} - \frac{r^2}{2}\sqrt{r^2-x^2}\arcsin\left(\frac{x}{r}\right) \\
&+ \frac{r^2}{2}\sqrt{r^2-x^2}\arcsin\left(\sqrt{1-\frac{\sqrt{r^2-x^{2^2}}}{r^2}}\right) \\
&- \sqrt{r^2-\sqrt{r^2-x^{2^2}}^2} + \frac{r^2}{2}\sqrt{r^2} \\
&= \frac{x^3}{6} - \frac{r^3}{6} - \frac{r^2}{2}\sqrt{r^2-x^2}\arcsin\left(\frac{x}{r}\right) \\
&+ \frac{r^2}{2}\sqrt{r^2-x^2}\arcsin\left(\frac{x}{r}\right) - \frac{xr^2}{2} + \frac{r^3}{2} \\
&= \frac{x^3}{6} + \frac{r^3}{3} - \frac{xr^2}{2}
\end{aligned}
$$

and combining the above, we get

$$
\begin{aligned}
\int_0^r\int_0^r A(r,x,y)\,dydx &= \int_0^r r^3\arcsin\left(\frac{x}{r}\right) + r^3\left(\frac{\pi}{2}-\frac{2}{3}\right) \\
&+ rx\sqrt{r^2-x^2} + \frac{x^3}{6} + \frac{r^3}{3} - \frac{xr^2}{2}\,dx \\
&= r^3\int_0^r \arcsin\left(\frac{x}{r}\right)\,dx + r^3\left(\frac{\pi}{2}-\frac{2}{3}\right)[x]_0^r + r\int_0^r x\sqrt{r^2-x^2}\,dx \\
&+ \frac{1}{6}\left[\frac{x^4}{4}\right]_0^r + \frac{r^3}{3}[x]_0^r - \frac{r^2}{2}\left[\frac{x^2}{2}\right]_0^r \\
&= r^3\left[x\arcsin\left(\frac{x}{r}\right) + \sqrt{r^2-x^2}\right]_0^r + r^4\left(\frac{\pi}{2}-\frac{2}{3}\right) \\
&+ r\left[-\frac{1}{3}(r^2-x^2)^{3/2}\right]_0^r + \frac{r^4}{24} + \frac{r^4}{3} - \frac{r^4}{4} \\
&= \frac{\pi}{2}r^4 - r^4 + r^4\left(\frac{\pi}{2}-\frac{1}{3}\right) + \frac{r^4}{3} + \frac{r^4}{24} - \frac{r^4}{4} \\
&= \pi r^4 - \frac{24}{24}r^4 + \frac{1}{24}r^4 - \frac{6}{24}r^4 = r^2\cdot r^2\left(\pi - \frac{29}{24}\right).
\end{aligned}
$$

Thus once more the contribution $r^2(\pi - 29/24)$ is weighted by the area r^2. With all integrals solved we can combine the results in (1) and arrive at

$$\begin{aligned}
p(r) &= 4 \cdot \left(\pi r^2 \cdot \left(\frac{1}{2} - r \right)^2 + (1-2r) r^3 \left(\pi - \frac{2}{3} \right) + r^4 \left(\pi - \frac{29}{24} \right) \right) \\
&= \pi r^2 (1-2r)^2 + (1-2r) r^3 \left(4\pi - \frac{8}{3} \right) + r^4 \left(4\pi - \frac{29}{6} \right) \\
&= \pi r^2 - 4\pi r^3 + 4\pi r^4 + r^3 \left(4\pi - \frac{8}{3} \right) - r^4 \left(8\pi - \frac{16}{3} \right) + r^4 \left(4\pi - \frac{29}{6} \right) \\
&= \pi r^2 - \frac{16}{6} r^3 + \frac{32}{6} r^4 - \frac{29}{6} r^4 \\
&= r^2 \left(\pi + \frac{3r^2 - 16r}{6} \right) = r^2 \left(\pi + \frac{r(3r-16)}{6} \right)
\end{aligned}$$

for the average size of the neighborhood. As expected with $r = 15/100$ it is $p(r) \approx 0.062$, which is a little less than a full disk $\pi r^2 \approx 0.071$ due to the boundary effects. □

The following five integrals have been used in the previous proof.

LEMMA A.1.

$$\int \arcsin(x)\, dx = x \arcsin(x) + \sqrt{1-x^2} + C$$

PROOF. Starting with integration by parts

$$\int u'v = uv - \int uv'$$

with

$$u' = 1 \quad \rightsquigarrow \quad u = x \qquad \text{and} \qquad v = \arcsin(x) \quad \rightsquigarrow \quad v' = \frac{d}{dx} \arcsin(x)$$

yields an easier integral. We must first determine v'. Let $y = \arcsin(x)$ then $x = \sin(y)$ and use implicit differentiation. Then solve for dy/dx and apply Pythagoras' Theorem and substitute $x = \sin(y)$.

$$\begin{aligned}
\frac{d}{dx} x &= \frac{d}{dx} \sin(y), \\
1 &= \frac{d}{dx} y \cos(y), \\
\frac{d}{dx} \arcsin(x) = \frac{d}{dx} y &= \frac{1}{\cos(y)} = \frac{1}{\sqrt{1 - \sin^2(y)}} = \frac{1}{\sqrt{1-x^2}}.
\end{aligned}$$

Hence we must now solve the integral on the right-hand side of

$$\int \arcsin(x)\, dx = x \arcsin(x) - \int \frac{x}{\sqrt{1-x^2}}\, dx. \tag{4}$$

Subsituting with $k = 1 - x^2$ thus

$$\frac{d}{dx} k = \frac{d}{dx} 1 - x^2 = -2x \qquad \rightsquigarrow \qquad \frac{dk}{-2x} = dx,$$

$$\int \frac{x}{\sqrt{1-x^2}}\,dx = \int \frac{x}{\sqrt{k}}\frac{dk}{-2x} = -\frac{1}{2}\int \frac{1}{\sqrt{k}}\,dk = -\frac{1}{2}\int k^{-1/2}\,dk = -\frac{1}{2}\frac{k^{1/2}}{1/2} = -\sqrt{k}.$$

Backsubstitution yields

$$\int \frac{x}{\sqrt{1-x^2}}\,dx = -\sqrt{1-x^2}.$$

Inserting this into (4) gives the result. □

LEMMA A.2.

$$\int y\sqrt{r^2-y^2}\,dy = -\frac{1}{3}\left(r^2-y^2\right)^{3/2} + C$$

PROOF. Substituting with $k = r^2 - y^2$ hence

$$\frac{d}{dy}k = \frac{d}{dy}r^2 - y^2 = -2y \qquad \rightsquigarrow \qquad \frac{dk}{-2y} = dy,$$

$$\int y\sqrt{r^2-y^2}\,dy = \int y\sqrt{k}\frac{dk}{-2y} = -\frac{1}{2}\int \sqrt{k}\,dk = -\frac{1}{2}\frac{k^{3/2}}{3/2} = -\frac{1}{3}(r^2-y^2)^{3/2} + C.$$

□

LEMMA A.3.

$$\int \arcsin(x/r)\,dx = x\arcsin(x/r) + \sqrt{r^2-x^2} + C$$

PROOF. Starting with integration by parts

$$\int u'v = uv - \int uv'$$

with

$$u' = 1 \quad \rightsquigarrow \quad u = x \qquad \text{and} \qquad v = \arcsin(x/r) \quad \rightsquigarrow \quad v' = \frac{d}{dx}\arcsin(x/r)$$

yields an easier integral. We must first determine v'. Let $y = \arcsin(x/r)$ then $x/r = \sin(y)$ and use implicit differentiation. Then solve for dy/dx and apply Pythagoras' Theorem and substitute $x/r = \sin(y)$.

$$\begin{aligned}
\frac{d}{dx}\frac{x}{r} &= \frac{d}{dx}\sin(y),\\
\frac{1}{r} &= \frac{d}{dx}y\cos(y),\\
\frac{d}{dx}\arcsin(x/r) = \frac{d}{dx}y = \frac{1}{r\cos(y)} &= \frac{1}{r\sqrt{1-\sin^2(y)}} = \frac{1}{r\sqrt{1-(x/r)^2}} \overset{r\geq 0}{=} \frac{1}{\sqrt{r^2-x^2}}.
\end{aligned}$$

Or with Lemma A.1 from before the chain rule gives us

$$\frac{d}{dx}\arcsin(x/r) = \frac{1}{\sqrt{1-\frac{x^2}{r^2}}}\cdot\left(\frac{x}{r}\right)' \overset{r\geq 0}{=} \frac{1}{\sqrt{r^2-x^2}}.$$

Hence we must now solve the integral on the right-hand side of

$$\int \arcsin(x/r)\,dx = x\arcsin(x/r) - \int \frac{x}{\sqrt{r^2-x^2}}\,dx. \tag{5}$$

Subsituting with $k = r^2 - x^2$ thus

$$\frac{d}{dx}k = \frac{d}{dx}r^2 - x^2 = -2x \qquad \rightsquigarrow \qquad \frac{dk}{-2x} = dx,$$

$$\int \frac{x}{\sqrt{r^2-x^2}}\,dx = \int \frac{x}{\sqrt{k}}\frac{dk}{-2x} = -\frac{1}{2}\int \frac{1}{\sqrt{k}}\,dk = -\frac{1}{2}\int k^{-1/2}\,dk = -\frac{1}{2}\frac{k^{1/2}}{1/2} = -\sqrt{k}.$$

Backsubstitution yields

$$\int \frac{x}{\sqrt{r^2-x^2}}\,dx = -\sqrt{r^2-x^2}.$$

Inserting this into (5) gives the result. □

LEMMA A.4.

$$\int \arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right)\,dy = y\arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right) - \sqrt{r^2-x^2} + C$$

PROOF. Starting with integration by parts

$$\int u'v = uv - \int uv'$$

with

$$u' = 1 \quad \rightsquigarrow \quad u = x \quad \text{and} \quad v = \arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right) \quad \rightsquigarrow \quad v' = \frac{d}{dy}\arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right)$$

yields an easier integral. We must first determine v'. Using the chain rule we find

$$\begin{aligned} v' &= \frac{1}{\sqrt{1-\sqrt{1-\frac{y^2}{r^2}}^2}} \cdot \left(\sqrt{1-\frac{y^2}{r^2}}\right)' \\ &= \frac{1}{\sqrt{1-1+\frac{y^2}{r^2}}} \cdot \frac{1}{2}\left(1-\frac{y^2}{r^2}\right)^{-\frac{1}{2}} \cdot \left(\frac{-2}{r^2}y\right) \\ &= \frac{1}{y/r} \cdot \frac{-1}{\sqrt{1-\frac{y^2}{r^2}}}\frac{y}{r^2} \\ &= \frac{-1}{\sqrt{r^2-y^2}}. \end{aligned}$$

Hence we must now solve the integral on the right-hand side of

$$\int \arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right)\,dy = y\arcsin\left(\sqrt{1-\frac{y^2}{r^2}}\right) - \int y\frac{-1}{\sqrt{r^2-y^2}}\,dy. \tag{6}$$

Substituting $k = r^2 - x^2$ thus

$$\frac{d}{dx}k = \frac{d}{dx}r^2 - x^2 = -2x \quad \rightsquigarrow \quad \frac{dk}{-2x} = dx,$$

$$\int \frac{-y}{\sqrt{r^2-y^2}}\,dy = \int \frac{-y}{\sqrt{k}}\frac{1}{-2y}\,dk = \frac{1}{2}\int k^{-1/2}\,dk = \frac{1}{2}2k^{1/2} = \sqrt{k}.$$

Backsubstitution yields

$$\int \frac{-y}{\sqrt{r^2-y^2}}\,dy = \sqrt{r^2-x^2}.$$

Inserting this in (6) gives the result. □

LEMMA A.5.

$$\int \sqrt{r^2-x^2}-b\,dx = \frac{r^2}{2}\arcsin\left(\frac{x}{r}\right)+\frac{x}{2}\sqrt{r^2-x^2}-bx+C$$

PROOF. By linearity we get

$$\int \sqrt{r^2-x^2}-b\,dx = \int \sqrt{r^2-x^2}\,dx - b\int 1\,dx = r\int \sqrt{1-x^2/r^2}\,dx - bx. \tag{7}$$

And we can continue with the standard trigonometric substitution $x/r = \sin(y)$, thus $y = \arcsin(x/r)$ and we continue with:

$$\frac{d}{dx}\frac{x}{r} = \frac{d}{dx}\sin(y) \qquad \rightsquigarrow \qquad \frac{1}{r} = \frac{d}{dx}y\cos(y) \qquad \rightsquigarrow \qquad dx = dyr\cos(y),$$

$$r\int \sqrt{1-\sin^2(y)}r\cos(y)\,dy = r^2\int \sqrt{\cos^2(y)}\cos(y)\,dy = r^2\int \cos^2(y)\,dy.$$

Using the cosine double angle formula

$$\cos(2x) = 2\cos^2(x)-1 \qquad \rightsquigarrow \qquad \frac{1+\cos(2x)}{2} = \cos^2(x),$$

we remove the squared cosine

$$r^2\int \frac{1+\cos(2y)}{2}\,dy = \frac{r^2}{2}\int 1+\cos(2y)\,dy = \frac{r^2}{2}y+\frac{r^2}{2}\int \cos(2y)\,dy = \frac{r^2}{2}y+\frac{r^2}{4}\sin(2y), \tag{8}$$

Using the sine double angle formula $\sin(2y) = 2\sin(y)\cos(y)$, and Pythagoras' Theorem to replace the cosine

$$\begin{aligned}\left(\frac{x}{r}\right)^2 &= \sin^2(y) = 1-\cos^2(y),\\ \frac{r^2-x^2}{r^2} &= \cos^2(y),\\ \frac{\sqrt{r^2-x^2}}{r} &= \cos(y)\end{aligned}$$

we can rewrite the right-hand side of (8)

$$\begin{aligned}\frac{r^2}{2}y+\frac{r^2}{4}\sin(2y) &= \frac{r^2}{2}y+\frac{r^2}{4}2\sin(y)\cos(y)\\ &= \frac{r^2}{2}y+\frac{r^2}{2}\sin(y)\frac{\sqrt{r^2-x^2}}{r}.\end{aligned}$$

Finally reverting the substitution with $y = \arcsin(x/r)$

$$\begin{aligned}&= \frac{r^2}{2}\arcsin(x/r)+\frac{r^2}{2}\sin(\arcsin(x/r))\frac{\sqrt{r^2-x^2}}{r}\\ &= \frac{r^2}{2}\arcsin(x/r)+\frac{r^2}{2}\frac{x}{r}\frac{\sqrt{r^2-x^2}}{r}\end{aligned}$$

$$= \frac{r^2}{2}\arcsin(x/r) + \frac{x}{2}\sqrt{r^2 - x^2}.$$

Inserting this in (8) and its result in (7) completes the proof. □

A.3. Average edge capacities — Proof of Lemma 7.1

The average capacity of an edge $e \in E(G_k)$ is

$$\mathbb{E}\big[\mathrm{cap}_{\mathit{gauss}}(e)\big] = \frac{1}{\Phi(251/50)} \sum_{a=0}^{\infty} \Phi\left(\frac{250-a}{50}\right) = \frac{501}{2} + \epsilon$$

with $\epsilon \ll 1$ and $\Phi(x)$ denoting the cdf of the standard normal distribution.

PROOF. The capacity of an edge is chosen in the interval $[0..\infty]$ according to a truncated normal distribution. $250 - \lfloor 50 \cdot X \rfloor$ where $X \sim N(0,1)$. Truncation is due to the fact, that negative capacities are pointless. Whenever $250 - \lfloor 50 \cdot X \rfloor < 0$ is true we discard the result and draw again. Due to

$$250 < \lfloor 50 \cdot \frac{251}{50} \rfloor = \lfloor 251 \rfloor = 251$$

we accept only draws within the half-open interval $X \in [-\infty, 251/50)$, consequently the normal distribution is truncated to this interval. The pdf of the truncated normal distribution is

$$f(x) = \frac{g(x)}{\Phi(251/50)} \qquad g(x) = \begin{cases} 0 & x \geq \frac{251}{50} \\ \phi(x) & x < \frac{251}{50} \end{cases}$$

With $\phi(x)$ being the pdf and $\Phi(x)$ denoting the cdf of the standard normal distribution $N(0,1)$

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \qquad \Phi(z) = \int_{-\infty}^{z} \phi(t)dt \qquad \frac{d}{dz}\Phi(z) = \phi(z).$$

The pdf and cdf of a random variable X normal distributed with mean μ and variance σ^2, that is $X \sim N(\mu, \sigma^2)$, can be written as

$$f_X(x) = \frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right) \qquad F_X(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

in terms of the standard normal distribution. This is due to the standard normalization $Z = \frac{X-\mu}{\sigma}$, that turns an RV X with mean μ and variance σ^2 into a RV Z with mean 0 and variance 1. Conversely $X = \mu \pm \sigma Z$ can be used to transform a standard normal random variable into one with mean μ and variance σ^2.

Integrating

$$\int_{-\infty}^{\infty} f(x)dx = \frac{1}{\Phi(251/50)} \int_{-\infty}^{\infty} g(x)dx = \frac{1}{\Phi(251/50)} \int_{-\infty}^{\frac{251}{50}} \phi(x)dx = \frac{\Phi(251/50)}{\Phi(251/50)} = 1$$

confirms that we have a probability distribution again.

Now for the expectation

$$\mathbb{E}\big[250 - \lfloor 50 \cdot X \rfloor\big] = 250 - \int_{-\infty}^{\infty} \lfloor 50 \cdot x \rfloor f(x)dx$$

we deal with the floor by treating it as what it is a piecewise constant function. Creating a step function, allows to collect all floating-point values that are mapped by the floor

function to the same integer.

$$250 - \lfloor 50 \cdot x \rfloor = a \in \mathbb{N}_0$$
$$250 - a = \lfloor 50 \cdot x \rfloor \Leftrightarrow x \in \left[\frac{250-a}{50}, \frac{250-a+1}{50} \right)$$

Thus we integrate

$$h(a) = \int_{\frac{250-a}{50}}^{\frac{250-a+1}{50}} f(x)dx = \frac{1}{\Phi(251/50)} \int_{\frac{250-a}{50}}^{\frac{250-a+1}{50}} g(x)dx = \frac{\Phi\left(\frac{250-a+1}{50}\right) - \Phi\left(\frac{250-a}{50}\right)}{\Phi(251/50)}$$

Checking that we have a distribution

$$\sum_{a=0}^{\infty} h(a) = \sum_{a=0}^{\infty} \frac{\Phi\left(\frac{250-a+1}{50}\right) - \Phi\left(\frac{250-a}{50}\right)}{\Phi(251/50)} = \frac{\sum_{a=0}^{\infty} \Phi\left(\frac{250-a+1}{50}\right) - \Phi\left(\frac{250-a}{50}\right)}{\Phi(251/50)}$$

clearly telescopes to

$$\sum_{a=0}^{\infty} h(a) = \frac{\Phi\left(\frac{250-0+1}{50}\right)}{\Phi(251/50)} = 1$$

Thus the expectation now simply is

$$\begin{aligned}
\sum_{a=0}^{\infty} a \cdot h(a) &= \sum_{a=0}^{\infty} a \cdot \frac{\Phi\left(\frac{250-a+1}{50}\right) - \Phi\left(\frac{250-a}{50}\right)}{\Phi(251/50)} \\
&= \frac{1}{\Phi(251/50)} \left(\sum_{a=0}^{\infty} a \cdot \Phi\left(\frac{251-a}{50}\right) - \sum_{a=0}^{\infty} a \cdot \Phi\left(\frac{250-a}{50}\right) \right) \\
&= \frac{1}{\Phi(251/50)} \left(\sum_{a=1}^{\infty} a \cdot \Phi\left(\frac{251-a}{50}\right) - \sum_{a=1}^{\infty} a \cdot \Phi\left(\frac{250-a}{50}\right) \right) \\
&= \frac{1}{\Phi(251/50)} \left(\sum_{a=0}^{\infty} (a+1) \cdot \Phi\left(\frac{251-(a+1)}{50}\right) - \sum_{a=1}^{\infty} a \cdot \Phi\left(\frac{250-a}{50}\right) \right) \\
&= \frac{1}{\Phi(251/50)} \left(\Phi\left(\frac{250}{50}\right) + \sum_{a=1}^{\infty} (a+1) \cdot \Phi\left(\frac{250-a}{50}\right) - a \cdot \Phi\left(\frac{250-a}{50}\right) \right) \\
&= \frac{1}{\Phi(251/50)} \left(\Phi\left(\frac{250}{50}\right) + \sum_{a=1}^{\infty} \Phi\left(\frac{250-a}{50}\right) \right) \\
&= \frac{\sum_{a=0}^{\infty} \Phi\left(\frac{250-a}{50}\right)}{\Phi(251/50)} \approx 250.500067...
\end{aligned}$$

□

A.4. Average maxflow in a square grid graph — Proof of Lemma 7.3

The average maxflows in a square grid graph G_k are bounded by:

$\mathbb{E}[\mathrm{mf}(\cdot)]$	*unit*	*unif*	*gauss*
llur	$2 \cdot 1$	$2 \cdot \frac{3350}{101}$	$2 \cdot 222.3$
lmum	$3 \cdot 1$	$3 \cdot \frac{3350}{101}$	$3 \cdot 222.3$

PROOF. There is no more flow possible than the minimum of the total capacities of source and sink. Due to the random edge weights there may not be enough edges in between with a high enough total capacity. As these bottlenecks may only lower the maxflow we ignore them and proceed with:

$$\mathbb{E}[\text{maxflow(s, t)}] \leq \mathbb{E}\big[\min\big\{\text{cap}(s), \text{cap}(t)\big\}\big] = c \cdot \mathbb{E}\big[\min\big\{\text{cap}(e), \text{cap}(e)\big\}\big]$$

with $c = \deg(s) = \deg(t)$. This expectation is the first moment of the first order statistics of two iid discrete random variables, it can be expressed via the "tail" of the cdf of the discrete random variables, see [David and Nagaraja 2003]:

$$\mathbb{E}\big[X_{(1)}\big] = \mathbb{E}\big[\min\{X_1, X_2\}\big] = \sum_{x=0}^{\infty} \big(1 - F_X(x)\big)^2.$$

In case of the discrete uniform distribution $\mathcal{U}$ on $[0..100]$ we find:

$$\begin{aligned}\mathbb{E}\big[U_{(1)}\big] &= \sum_{u=0}^{100} \left(1 - \frac{1+u}{101}\right)^2 = \sum_{u=0}^{100} \left(\frac{100-u}{101}\right)^2 = \sum_{u=1}^{100} \left(\frac{u}{101}\right)^2 = \frac{1}{101^2} \sum_{u=1}^{100} u^2 \\ &= \frac{1}{101^2} \frac{(200+1)(100+1)100}{6} = \frac{3350}{101} = 33.1683.\end{aligned}$$

In case of the gauss distribution $X \sim \mathcal{N}(0, 1)$ we redraw whenever $250 - \lfloor 50 \cdot X \rfloor < 0$ is true. Due to

$$250 < \left\lfloor 50 \cdot \frac{251}{50} \right\rfloor = 251$$

we accept only draws within the half-open interval $X \in [-\infty, 251/50)$. The truncated pdf $f_X(x)$ of the standard normal distribution is then:

$$f_X(x) = \frac{g(x)}{\Phi(251/50)} \qquad g(x) = \begin{cases} 0 & x \geq 251/50 \\ \phi(x) & x < 251/50 \end{cases},$$

therefore its cdf is

$$F_X(x) = \begin{cases} 1 & x > 251/50 \\ \Phi(x)/\Phi(215/50) & x \leq 251/50 \end{cases}.$$

Transforming with $Z = 250 - 50X$ yields:

$$F_Z(z) = \begin{cases} 0 & z < -1 \\ 1 - \Phi\big((250 - z)/50\big)/\Phi(251/50) & z \geq -1 \end{cases}.$$

Thus

$$\mathbb{E}\big[Z_{(1)}\big] = \sum_{z=0}^{\infty} \big(1 - F_Z(z)\big)^2 = \sum_{z=0}^{\infty} \left(\frac{\Phi\left(\frac{250-z}{50}\right)}{\Phi(251/50)}\right)^2 \approx 222.3$$

where $\phi(x)$ and $\Phi(x)$ are the pdf and cdf of the standard normal distribution respectively. □

A.5. DFS, BFS and PFS implementations of the graph search

```
cEdge[] st; // spanning tree parent pointer

void augment(int s, int t) {
  int d = st[t].capRto(t);  // init with remaining cap into the sink
```

```
  for (int v = ST(t); v != s; v = ST(v)) { // search sink to source
    if (st[v].capRto(v) < d) {             // for the bottleneck
      d = st[v].capRto(v);
  } }
  st[t].addflowRto(t, d);
  for (int v = ST(t); v != s; v = ST(v)) { // push the flow along
    st[v].addflowRto(v, d);                // the augmenting path
} }
```

The annotation `produceCost("EdgeFlowChanged", 1)` inside the method `addflowRto()` is used to keep track of the updates made to the graph data structure. The next code fragments show the similarity of the `dfs()` and `bfs()` routines, just the stack is replaced by a queue. The part where the iterator visits the incident edges of a vertex is shown in both versions. First we present the code for depth-first-search:

```
int[] wt; // flag for already visited nodes
cEdge[] st; // spanning tree parent pointer

boolean dfs() {
  intStack iS = new intStack(G.V());
  for (int v = 0; v < G.V(); v++) { // init
    wt[v] = 0;
    st[v] = null;
  }
  produceCost("dfsPush", 1);
  iS.push(s);
  wt[s] = 1; // mark s as visited
  while (!iS.isEmpty()) {
    produceCost("dfsPop", 1);
    int v = iS.pop(); // current node
    if (v == t) { return true; } // stop on sink
    cGraph.AdjList A = G.getAdjList(v);
    for (cEdge e = A.beg(); !A.end(); e = A.nxt()) {       // dfs
    for (cEdge e = A.begrnd(); !A.end(); e = A.rndnxt()) { // dfsrnd
      produceCost("dfsTouch", 1);
      int w = e.other(v); // the other end of the edge
      if (e.capRto(w) > 0) { // any cap left?
        if (wt[w] == 0) { // check if already visited
          st[w] = e; // move the edge from fringe to the DFS-spanning tree
          wt[w] = 1; // mark w as visited
          produceCost("dfsPush", 1);
          iS.push(w);
  } } } }
  return false;
}
```

Next the code for breadth-first-search is given:

```
int[] wt; // flag for already visited nodes
cEdge[] st; // spanning tree parent pointer

boolean bfs() {
  intQueue iQ = new intQueue(G.V());
  for (int v = 0; v < G.V(); v++) { // init
```

```
    wt[v] = 0;
    st[v] = null;
  }
  produceCost("bfsPut", 1);
  iQ.put(s);
  wt[s] = 1; // mark s as visited
  while (!iQ.empty()) {
    produceCost("bfsGet", 1);
    int v = iQ.get(); // current node
    if (v == t) { return true; } // stop on sink
    cGraph.AdjList A = G.getAdjList(v);
    for (cEdge e = A.beg(); !A.end(); e = A.nxt()) {       // bfs
    for (cEdge e = A.begrnd(); !A.end(); e = A.rndnxt()) { // bfsrnd
      produceCost("bfsTouch", 1);
      int w = e.other(v); // the other end of the edge
      if (e.capRto(w) > 0) { // any cap left?
        if (wt[w] == 0) { // check if already visited
          st[w] = e; // move the edge from fringe to the BFS-spanning tree
          wt[w] = 1; // mark w as visited
          produceCost("bfsPut", 1);
          iQ.put(w);
  } } } }
  return false;
}
```

The methods `push()`/`put()` and `pop()`/`get()` are annotated to be able to track them.

The maximum capacity augmenting path heuristic is implemented via a priority queue that uses the leftover capacity as the priority.

```
int[] wt; // remaining negative cap along the PFS spanning tree edge
cEdge[] st; // spanning tree parent pointer

boolean pfs() {
  intPQi pQ = new intPQi(G.V(), wt); // capacities are the PQs priority
  for (int v = 0; v < G.V(); v++) {  // init
    wt[v] = 0;
    st[v] = null;
    pQ.insert(v);  // like Dijkstra insert all vertecies
  }
  wt[s] = -5000;   // set to largest minimum (unlimited inflow at source)
                   // -> move source to front of PQ
  pQ.lower(s);     // priority has changed fix PQ-heap
  while (!pQ.empty()) {
    produceCost("pfsGetmin", 1);
    int v = pQ.getmin();
    wt[v] = -5000; // mark visited and assume unlimited inflow we are just
                   // searching for a path not yet sending flow along it
    if (v == t) { break; } // stop on sink
    if (v != s && st[v] == null) { break; } // node not part of the PFS
    cGraph.AdjList A = G.getAdjList(v);                    //  spanning tree
    for (cEdge e = A.beg(); !A.end(); e = A.nxt()) { // check the neighbors
      produceCost("pfsTouch", 1);
      int w = e.other(v);    // the other end of the edge
```

```
      int cap = e.capRto(w); // leftover cap towards w
      int P = cap < -wt[v] ? cap : -wt[v]; // inflow from v is capped by
      if (cap > 0 && -P < wt[w]) {                  // the cap constraint
        wt[w] = -P;
        pQ.lower(w);
        produceCost("pfsEdgesConsidered", 1);
        st[w] = e; // move the edge from fringe to the PFS-spanning tree
  } } }
  return st[t] != null; // check if the sink is in the PFS-spanning tree
}
```

A.6. Maximum Likelihood Analysis

This section contains a brief introduction to the maximum likelihood analysis method introduced in [Laube and Nebel 2010]. There are three main ideas to this analysis method:

First, we know from probability theory that the partial derivative of a probability generating function (pgf) evaluated at 1 gives the expectation of a discrete random variable. To find the average of a performance parameter of an algorithm, we regard it as a random variable that depends on the distribution of the inputs. Our goal is then to derive the pgf from the algorithm to be analyzed and a sufficient sample of its inputs using methods from statistical inference.

Second, in analytic combinatorics [Flajolet and Sedgewick 2009] an idea from [Chomsky and Schützenberger 1963] is regularly used to solve counting problems by encoding the structure of combinatorial objects into words of formal languages and translating a grammar of the language into ordinary generating functions. An example how the structure of tries can be encoded with Motzkin words and used in an average-case analysis can be found in [Laube and Nebel 2010]. Modifying this approach to incorporate stochastic context free languages, to allow non-uniform settings, enables us to translate the problem of finding the pgf into constructing an appropriate stochastic context free grammar, that is a context free grammar together with probabilities associated with its rules.

And third, for the purpose of analyzing algorithms a regular language, that can be described by a right-linear grammar derived automatically from the source code of the algorithm, suffices. The grammar describes the traces of the algorithm's execution and is augmented with probabilities for each rule that are determined from a "typical" sample set of inputs. Here R.A. Fisher's maximum likelihood principle [Aldrich 1997] is used to tune the probabilities of this stochastic grammar to capture the non-uniform probability distribution induced by the set of inputs. We call this the *maximum likelihood training* of the grammar.

In the following subsections we explain how the grammar is obtained and trained and how it is transformed into a probability generating function.

A.6.1. A General Regular Grammar for Analyzing Algorithms. For the purpose of analyzing algorithms a regular language, that can be described by a right-linear grammar derived automatically from the source code of the algorithm, is used to describe the traces of the algorithm's execution (a trace is a sequence of line numbers of the executed instructions).

Definition A.6 (*trace, trace language*). Given an algorithm $\mathcal{A}$ specified in an low-level language (like Knuth's MMIX assembly code or Java bytecode), we let $L_{\mathcal{A}}$ be the set of line numbers appearing in the specification of algorithm $\mathcal{A}$. The set of line numbers $L_{\mathcal{A}}$ will serve as alphabet. The words from $L_{\mathcal{A}}^{\star}$ (sequences of line numbers) that correspond to the flow of control, that can be observed when algorithm $\mathcal{A}$ runs on

the input v, is called the *trace of $\mathcal{A}$ on input v*, denoted by $T_{\mathcal{A}}(v)$. The set of all words $T_{\mathcal{A}}(v)$ is the *trace language* of $\mathcal{A}$ (the set of all control flows), denoted by $\mathcal{L}_{\mathcal{A}}$.

The reasons for choosing a low-level representation (here Java bytecode) of the algorithm are: First, the low-level representation serves as a kind of normal form with respect to the control flow, as we only have to distinguish between unconditional and conditional jumps and all other instructions. Second, for parameters that do not depend on the particular value of the input or where it is stored in memory the trace language provides all information for the analysis of algorithm $\mathcal{A}$. While we can analyze parameters like the running time in clock cycles we usually use coarser units, like the number of comparison, exchanges, visited nodes, arithmetic operations, etc., that is, the typical elementary operations considered in the analysis of algorithms.

We only have to assign the various "costs" to the operations in the specification of algorithm $\mathcal{A}$. Formally this is expressed through a parameter.

Definition A.7 (*parameter*). A *parameter* is given by a homomorphism $\lambda : L^\star \to \mathbb{N}_0$ with $\lambda(\epsilon) = 0$ and $\lambda(u \circ w) = \lambda(u) + \lambda(w)$ where $\circ$ denotes concatenation and ϵ the empty word.

If we let $p(v)$ be the probability for input v and $\mathcal{I}_n$ the set of all inputs of size n, then the expected cost of λ for all inputs of size n is simply

$$\mathbb{E}[\lambda] := \sum_{v \in \mathcal{I}_n} p(v) \cdot \lambda(T_{\mathcal{A}}(v)). \tag{9}$$

Unlike the situation in analytic combinatorics, were a new formal language – a new encoding – has to be found for every new class of combinatorial objects to be analyzed, we define a general grammar that describes the traces of the algorithm's execution. Each rule is augmented with probabilities, this stochastic grammar is then a model for the algorithm's behavior. The rule's probabilities will be determined from a "typical" sample set of inputs.

Definition A.8 (*trace grammar*). A *trace grammar* $G_{\mathcal{A}}$ for algorithm $\mathcal{A}$ with line numbers $L_{\mathcal{A}} = \{1, \ldots, m\}$ is a stochastic context free grammar (scfg) $G_{\mathcal{A}} = (N, T, R, P, S)$, where $N = \{L_1 = S, L_2, \ldots, L_{m+1}\}$ is a set of variables or non-terminal symbols. $T = \{\ell_1, \ldots, \ell_m\}$ is a set of terminal symbols disjoint from N. The set of rules $R = \{r_1, \ldots, r_{m+1}\}$ is a subset of $N \times (TN \cup \{\epsilon\})$. Its elements $r_j = (L_i, \omega_j)$ are also called productions. Additionally we denote by $R_i = \{r_j \mid r_j = (L_i, \omega_j) \in R\}$ the set of rules with the same left-hand side L_i. The mapping P from R into $(0, 1]$ assigns each rule r_j its probability p_j. We also write $L_i \to p_j : \omega_j$ for a rule $r_j = (L_i, \omega_j) \in R$ with $P(r_j) = p_j$. We require that

$$\sum_{\{j \mid r_j \in R_i\}} p_j = 1, \quad i = 1, 2, \ldots, m+1$$

holds. This ensures that we have a probability distribution on the rules with the same left-hand side. The symbol $S = L_1 \in N$ is a distinguished variable called the *axiom* or start symbol. All sets in this definition are finite.

Let $\mathcal{L}(G_{\mathcal{A}})$ be the language generated by the trace grammar $G_{\mathcal{A}}$, that is the set of all terminal strings or words which can be generated by successively substituting all non-terminals according to the productions starting with the axiom S. Such a successive substitution is called a *derivation* and is written as $\alpha \Rightarrow^\star \beta$, if there is a possibly empty sequence of rules $r_{i_1}, \ldots, r_{i_k}$ that leads from α to β. The probabilities of the rules now induce probabilities on the derivations by multiplying the involved rule's probabilities

$p = p_{i_1} \cdot \ldots \cdot p_{i_k}$. This can be extended to the whole set of terminal strings, that is the whole language, by summing up all probabilities of the different left-most derivations of each terminal string.

For the trace grammar $G_{\mathcal{A}}$ to actually generate the trace language $\mathcal{L}_{\mathcal{A}} \subseteq \mathcal{L}(G_{\mathcal{A}})$ we have to describe how the low-level specification of the algorithm $\mathcal{A}$ is translated into the set of grammar rules. Each line of the low-level code is translated according to the following three simple rules:

(1) An unconditional jump from line i to line j is expressed by the rule $L_i \to 1 : \ell_i L_j$.
(2) A conditional jump in line i that may jump to line j is expressed by the rule $L_i \to p_i : \ell_i L_j \mid 1 - p_i : \ell_i L_{i+1}$.
(3) All other instructions yield the rule $L_i \to 1 : \ell_i L_{i+1}$.

A last production $L_{m+1} \to 1 : \epsilon$ is added to allow the grammar to generate terminal strings, that are obviously sequences of line numbers of the executed instructions. The parameter homomorphism will reduce the amount of information further and focus on the chosen elementary operations, but we are free to use all the details from these terminal strings to count for example the number of memory access' if required. By design we ignore the contents of registers or memory locations. Consequently we have the following lemma.

LEMMA A.9. *A trace grammar $G_{\mathcal{A}}$ build according to the three rules presented above generates the trace language of a given algorithm $\mathcal{A}$, that is $\mathcal{L}_{\mathcal{A}} \subseteq \mathcal{L}(G_{\mathcal{A}})$. Furthermore the grammar can be efficiently constructed in a single pass over the specification of the algorithm.*

Relaxing our definition of the rule set of our grammar to be a subset of $N \times (T^+ N \cup \{\epsilon\})$ allows us to combine many linear rules like $L_i \to \ell_i L_j$ and thus reduces the number of rules to the number of conditional jumps in the low-level specification of the algorithm.

A.6.2. Obtaining a Generating Function from the Grammar. Having constructed the grammar as discussed in the previous section we can apply the idea of [Chomsky and Schützenberger 1963] and derive a probability generating function (pgf), whose partial derivative we want to evaluate at 1 to get the average performance of a parameter of an algorithm as in Eq. (9). We obtain the pgf by translating the grammar rules into linear equations using the following two transformations:

$$\begin{aligned} L_i \to 1 : \ell_i L_{i+1} \quad &\rightsquigarrow \quad L_i(z,y) = y^{\lambda(i)} z^{\gamma(i)} L_{i+1}(z,y), \\ L_i \to p_i : \ell_i L_j \mid q_i : \ell_i L_{i+1} \quad &\rightsquigarrow \\ & L_i(z,y) = p_i y^{\lambda(i)+2} z^{\gamma(i)} L_j(z,y) + q_i y^{\lambda(i)} z^{\gamma(i)} L_{i+1}(z,y), \end{aligned} \tag{10}$$

where $q_i = 1 - p_i$ and γ is a second homomorphism, that assigns each trace $T_{\mathcal{A}}(v)$ the size of the input v, that is for $v \in \mathcal{I}_n$ we have $\gamma(T_{\mathcal{A}}(v)) = n$. The homomorphism γ is chosen in such a way, that it assigns a line i a contribution of 1 to the total size of the input, if one element of the input is accessed by that line for the last time, because algorithms do not work on the input as a whole. For example if the size of the input is measured by the number of nodes in a graph, the part of the algorithm that works on a node for the last time gets a contribution of 1. Thus whenever this part of the algorithm finishes working on a node it contributes to the total size, which amounts to the correct size when all nodes are processed.

The set of linear equations obtained from the set of grammar rules can now be solved for $L_1(z, y)$, leading to the pgf

$$L_1(z,y) = \sum_{w \in \mathcal{L}(G_\mathcal{A})} p_w y^{\lambda(w)} z^{\gamma(w)} = \sum_{n \geq 0} \sum_{k \geq 0} \sum_{\substack{v \in \mathcal{I}_n \\ \lambda(T_\mathcal{A}(v))=k}} p(v) y^k z^n. \tag{11}$$

Next we can find symbolic, due to the yet unspecified probabilities, expressions for the expectation of the parameter λ by conditioning the partial derivative of the pgf to a fixed size of the inputs:

$$\mathbb{E}[\lambda] = \frac{[z^n] \frac{\partial}{\partial y} L_1(z,y) \mid_{y=1}}{[z^n] L_1(z,1)} = \frac{\sum_{w \in \mathcal{L}_n(G_\mathcal{A})} p_w \lambda(w)}{\sum_{w \in \mathcal{L}_n(G_\mathcal{A})} p_w} = \frac{\sum_{v \in \mathcal{I}_n} p(v) \cdot \lambda(T_\mathcal{A}(v))}{\sum_{v \in \mathcal{I}_n} p(v)}, \tag{12}$$

where $\mathcal{L}_n(G_\mathcal{A})$ is a subset of $\mathcal{L}(G_\mathcal{A})$ containing all traces $T_\mathcal{A}(v)$ with $\gamma(T_\mathcal{A}(v)) = n$. Conditioning is necessary, as the probabilities of the grammar rules induce a probability distribution on the entire trace language and we are only looking at the subset of all traces $T_\mathcal{A}(v)$ with $\gamma(T_\mathcal{A}(v)) = n$.

A.6.3. Maximum Likelihood Training of the Grammar Probabilities. The still unspecified probabilities p_i of the rules in the trace grammar correspond to the conditional jumps in the low-level description of the algorithm and are determined by observing the algorithm working on sample inputs. This allows us to make statements about the performance parameters behavior, assuming a probability model derived from a specific family of inputs.

We call this step the *maximum likelihood training* because the probabilities of the trace grammar rules can be interpreted as the free parameters of that probability model. Then the maximum likelihood principle instructs us to adjust the free parameters of the probability model in such a way that any other choice would make the same observation less likely. From the work of [Chi and Geman 1998; Nederhof and Satta 2006] we know, that assigning relative frequencies, observed on a sample, yields maximum likelihood estimates of the unknown probabilities and moreover induces a probability distribution on the entire trace language, which is not the case for an arbitrary choice. Thus a trained grammar allows us to generate words with are distributed very much the same as the traces observed on the sample inputs. Consequently we obtain a model for the behavior of the algorithm for a specific family of inputs. In [Laube and Nebel 2010] we have been able to prove that D. Knuth's classical approach towards an average-case analysis, see [Knuth 1997], and a maximum likelihood analysis yield precisely the same expectations when they are provided with the same probability model.

Once the model is trained we treat it as given. Like in other sciences when the scientific method is applied, we accept the inherent simplifications made while building the model and use it nevertheless to make predictions based on it. This is a compromise we are willing to make as a complete and true representation of the average-case behavior of a complex algorithm maybe impossible.

Obviously such a training requires an implementation of the algorithm $\mathcal{A}$ to record the relative frequencies of the jumps for every input v in the sample set. The details of how these counts are obtained and processed were presented in [Laube and Nebel 2010]. The result of such a maximum likelihood training is a set of probability functions in dependence of the input size n.

ACKNOWLEDGMENTS

This work was inspired by the slides of two talks of R. Sedgewick on "The Role of the Scientific Method in Programing" [Sedgewick 2010b] and "Putting the Science back into Computer Science" [Sedgewick 2010a].

The authors would like to thank Sebastian Wild, Raphael Reitzig, Michael Holzhauser, Vasil Tenev and Florian Furbach for their ongoing effort in the development of MaLiJAn, the tool to do the maximum likelihood average case analysis of Java bytecode programs semi-automatically.

References

ALDRICH, J. 1997. R. a. fisher and the making of maximum likelihood 1912–1922. *Statistical Science 12,* 3, 162–176.

CHANDRAN, B. AND HOCHBAUM, D. 2009. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Op. Res. 57*, 358–376.

CHI, T. AND GEMAN, S. 1998. Estimation of probabilistic context-free grammars. *Computational Linguistics 24,* 2, 299–308.

CHOMSKY, N. AND SCHÜTZENBERGER, M.-P. 1963. The algebraic theory of context-free languages. In *Computer Programming and Formal Languages*, P. Braffort and D. Hirschberg, Eds. North Holland, 118–161.

DAVID, H. A. AND NAGARAJA, H. N. 2003. *Order Statistics* Third Ed. Wiley.

DINIC, E. A. 1970. An algorithm for the solution of the max-flow problem with the polynomial estimation. *Soviet Math. Dokl 11*, 1277–1280. Dokl. Akad. Nauk SSSR 194, 1970, no.4 (in Russian).

DURSTENFELD, R. 1964. Algorithm 235: Random permutation. *Communications of the ACM 7,* 7, 420.

EDMONDS, J. AND KARP, R. M. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19,* 2, 248–264.

ERDOS, P. AND RÉNYI, A. 1959. On random graphs i. *Publ. Math. Debrecen 6*, 290–297.

FLAJOLET, P. AND SEDGEWICK, R. 2009. *Analytic Combinatorics*. Camb. Univ. Press.

FORD, L. R. AND FULKERSON, D. R. 1962. *Flows in Networks*. Princeton University Press.

GILBERT, E. N. 1959. Random graphs. *Annals of Mathematical Statistics 30*, 1141–1144.

GOLDBERG, A. V. AND RAO, S. 1998. Beyond the flow decomposition barrier. *J. Assoc. Comput. Mach. 45*, 753–782.

GOLDFAB, D. AND GRIGORIADS, M. D. 1988. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Oper. Res. 13*, 83–123.

KNUTH, D. E. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms* Third Ed. Addison Wesley.

KNUTH, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* Third Ed. Addison Wesley.

LAUBE, U. AND NEBEL, M. E. 2010. Maximum likelihood analysis of algorithms and data structures. *Theoretical Computer Science 411,* 1, 188–212.

MOTWANI, R. 1994. Average-case analysis of algorithms for matchings and related problems. *Journal of the ACM 41,* 6, 1329–1356.

NEDERHOF, M.-J. AND SATTA, G. 2006. Estimation of consistent probabilistic context-free grammars. In *HLT-NAACL*. New York, USA, 343–350.

ORLIN, J. B. 2013. Max flows in $O(nm)$ time, or better. In *STOC '13: Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*. ACM, New York, NY, USA, 765–774.

PENROSE, M. 2003. *Random Geometric Graphs*. Oxford University Press.

SEDGEWICK, R. 2003. *Algorithms in Java, 3rd. Ed., Part 5 Graph Algorithms*. Addison Wesley.

SEDGEWICK, R. 2010a. Putting the science back into computer science. www.cs.princeton.edu/rs~/talks/ScienceCS10.pdf.

SEDGEWICK, R. 2010b. The role of the scientific method in programming. www.cs.princeton.edu/~rs/talks/ScienceCS.pdf.

VIZING, V. G. 1963. The cartesian product of graphs. *Vyčisl. Sistemy 9*, 30–43.

WILD, S., NEBEL, M., REITZIG, R., AND LAUBE, U. Engineering java 7's dual pivot quicksort using malijan. In *ALENEX 2013: Proceedings of the Meeting on Algorithm Engineering & Experiments*. New Orleans, USA, 55–70.