

A Randomized Queueless algorithm for Breadth-First Search

K. Subramani

LDCSEE,

West Virginia University,

Morgantown, WV

ksmani@csee.wvu.edu

K. Madduri

College of Computing,

Georgia Institute of Technology,

Atlanta, GA

kamesh@cc.gatech.edu

October 26, 2007

Abstract

First Come First Served is a policy that is accepted for implementing fairness in a number of application domains such as scheduling in Operating Systems [28, 11], scheduling web requests and so on. We also have orthogonal applications of FCFS policies in proving correctness of search algorithms such as Breadth-First Search and the Bellman-Ford FIFO implementation for finding single-source shortest paths [2], program verification [12] and static analysis [25, 24]. The principal data structure used in implementing FCFS policies is the *queue*, which is realized either through a circular array or a linked list. The question of interest then, is whether queues are required to implement FCFS policies; this paper provides empirical evidence answering this question in the negative. The principal contribution of this paper is the development of a randomized algorithm to implement approximate FCFS policies without queues. The

techniques that are developed in this paper find direct applications in program verification, model checking, in the implementation of distributed queues and in the design of incremental algorithms for shortest path problems.

Keywords: FCFS policies, distributed queues, scheduling, Breadth-First Search, randomized algorithms.

1. Introduction

First Come First Served (FCFS) is a policy used to ensure fairness in a number of application domains such as scheduling [34] and Operating Systems [28]. The motivating factor underlying this form of fairness, especially in the servicing of requests, is to preserve order, i.e., if request A has a smaller time-stamp than request B, then request A should be serviced before request B. The only known method of implementing this policy is through the use of a “queue” data structure, in which elements are inserted at the rear and removed from the front. In practice, queues are realized through circular arrays or linked lists; the code for maintaining queues, while conceptually simple, mandates the checking of a number of conditions and is therefore non-trivial [22]. The question of interest then, is whether FCFS fairness (or at least an approximation of FCFS) can be implemented without queues; this paper is devoted towards answering this question. We provide empirical evidence that conclusively demonstrates that queue structures are not necessary to achieve FCFS fairness; indeed, even implicit queues (See Section 2) need not be used. All that is needed is a source for random bits; our algorithm exploits the existence of such a source to effectively simulate a standard queue. Our work

establishes that randomization can serve as an effective substitute to order, insofar as establishing FCFS fairness is concerned. We also point out that our algorithm is the first of its kind to explicitly introduce randomization in reachability problems.

There are a number of advantages to the approach that we propose in this paper, viz.,

(a) **Simplicity** – As per the literature, maintaining queues, especially in distributed applications is a non-trivial task. Accordingly, if it is possible to achieve the FCFS effect without queues then that possibility must be explored.

(b) **Design Paradigms** – We will show in Section 4 that our algorithm is in fact, a complete procedure for the Single Source Shortest Path problem with arbitrary weights. We point out that our algorithm is the first of its kind to explicitly introduce randomization in the vertex-selection procedure. This needs to be contrasted with the traditional Bellman-Ford approach [9] and the approaches in [21, 2].

(c) **Incrementality** – The literature is replete with approaches for incremental algorithms for the Breadth-First Search problem and more generally, Single Source Shortest Path problems. Incremental versions of these traditional graph problems have been studied in their own right, as well as because they arise in domains such as program verification and database query optimization. Our algorithm can easily be converted into an incremental approach to these problems.

(d) **Robustness and Security** – Distributed queues are used in real-world applications such as web crawlers [31] and routing algorithms in VLSI design [17]. FCFS policies and Breadth-First Search are central to security applications analyzing information flow [26, 18] and leaks [16],

parallel program analysis [30], and network localization algorithms [32]. Our algorithm obviates the need for distributed queues, while maintaining approximate FCFS ordering. The data structure is more robust than a distributed queue, and can also be used to implement the security applications detailed above.

The rest of this paper is organized as follows: Section 2 provides a formal statement of the problem under consideration, and argues that it is equivalent to performing a Breadth-First search on a graph. The motivation for our work, as well as related approaches, is detailed in Section 3. In Section 4, we present the Randomized Breadth-First search (RBFS) algorithm and discuss its correctness. Section 5 compares the performance of the RBFS algorithm with respect to traditional BFS on uniprocessor architecture; the implementation details of the algorithm are provided in this section. This is followed by a discussion of the relative performance of the two algorithms on a multiprocessor architecture in Section 6. We conclude in Section 7 by summarizing our results and discussing avenues for future research.

2. Statement of Problem

The problem that we are interested in is as follows: \mathbf{P}_1 : *Given a sequence of requests, $S = \langle s_1, s_2, \dots, s_n \rangle$ which are totally ordered by time, can we service the requests in approximately the same order as their arrival, without using a queue to store the requests?*

Observe that if we are told that the requests must be served in *exactly* the same order as their arrival, then a queue is necessary. It is the relaxation of this requirement to “approximately, in the order of arrival” that permits us to use randomization and eliminate queues.

2.1 Implicit queues

We note that problem P_1 can be solved by the following technique: Assume that the requests enter S over time. At each step, the requests in S are weighted with a probability distribution such that the longer a request is in S , the larger is its weight. Then, the next request is chosen at random, as per this probability distribution, and not uniformly. It is not hard to see that requests will be served roughly in the order that they arrived, in that the expected wait time of a request will be stochastically dominated by the expected wait times of requests arriving after it [38].

However, note that although explicitly defined queues have not been used in the above protocol, we have merely shifted the queuing problem one level in the abstraction process. To see this, observe that that we will need a counter associated with each request to keep track of the time it has spent in S ; these counters then need to be sorted before imposing the probability distribution of the protocol. The sorting requirement makes this protocol more complex than simple queues and defeats the purpose of queue elimination. Queues implemented using this protocol are called *Implicit queues* have been used in page replacement policies in caches [1].

We shall empirically demonstrate that FCFS fairness can be accomplished without queues, explicit or implicit.

2.2 Breadth First Search

We now argue that the FCFS problem is simulated by performing the Breadth First Search on an arbitrary graph. Algorithm 2.1 represents the traditional approach for performing Breadth First Search.

Function *TRAD – BFS*($G = \langle V, E \rangle, s$)

Output : Set of all vertices $v \in V$ reachable from s , and $d[v]$, the shortest path from s to v , given all edges to be unit weight.

1. Construct an empty queue Q and initialize it to s .
2. $d[s] = 0$
3. $color[s] = black$
4. **for** ($v \in V, v \neq s$) **do**
5. $d[v] = \infty$
6. $color[v] = white$
- end for**
7. **while** $|Q| \neq \phi$ **do**
8. Let $v = head(Q)$
9. Delete $head(Q)$
10. **for** (each vertex u adjacent to v) **do**
11. **if** ($color[u] = white$) **then**
12. Add vertex u to Q
13. $color[u] = black$
14. $d[u] = d[v] + 1$
- end if**
- end for**
- end while**

Algorithm 2.1: Traditional Breadth-First Search

Consider an arbitrary level-based labeling of the vertices in G ; with vertices in level i getting a lower label than the vertices in level $i+1$. Vertices in the same level are numbered arbitrarily. We can think of the queue Q as being populated by requests, with the vertices representing the requests; we say that request v_i precedes request v_j , if v_i enters Q before v_j . When a vertex is deleted from Q , it is said to be serviced. Observe that Q implements

FCFS fairness in that all requests at a particular level are serviced before requests at higher levels.

As discussed above, there are other methods through which FCFS fairness can be implemented. For an arbitrary protocol A , using storage structure R , we define the *wait time of a vertex* as the number of times it is inserted in R . Likewise, we define the *wait time of the protocol* as the maximum wait time of any vertex.

In the traditional BFS protocol, R is a queue and each vertex is inserted precisely once in R . Accordingly, the wait time of every vertex is 1 and the wait time of traditional BFS is also 1. The wait time of an arbitrary protocol is a measure of how accurately it implements FCFS, in that larger the wait time, the more it deviates away from FCFS.

For a randomized protocol, the metrics of interest are the *expected wait time of a vertex* and the *expected wait time of the protocol* respectively.

3. Motivation and Related Work

The motivation for our work arises from five different design domains, viz.,

(a) Program Verification – A typical program is represented as Control-Flow graph (CFG); typical questions involving the reachability of unsafe states, non-termination of loops and so on can be answered through BFS [29, 20]. Reachability analysis is also used in model-checking LTL formulae [7], understanding secure information flow [18], and timed automata [4]. Consequently, the technique that we have proposed will find immediate applications in these

domains. We would like to point out that our algorithmic paradigm is not part of the existing literature, to the best of our knowledge.

(b) Shortest Path paradigms – Consider the problem of finding the shortest paths from a specified source in a positively weighted graph (SSSP). SSSP is one of the most well studied problems in Operations Research and theoretical computer science [10]. The first polynomial time algorithm for this problem was proposed by Dijkstra in [15]; this has been followed by a number of incremental improvements on both the theoretical side and the practical side [3, 40, 37]. However, the theoretical advances are impractical from the implementational perspective, while the practical advances have been primarily in the realm of improved data structuring. The approach described in this paper, is the first of its kind to explicitly introduce randomization in the vertex selection process and by doing so, it provides a means to address the demands of the worst case analysis of deterministic algorithms.

(c) Distributed Queues – Consider a web-server A , which is connected to a number of satellite servers A_1, A_2, \dots, A_n . Requests from the external world to A are routed through the satellite servers. Each request comes with a time-stamp and is stored locally at the satellite server. If we were required to serve requests in strict order of time-stamps, we would have to poll all the servers to determine the request with the smallest timestamp. Our strategy here demonstrates that it is sufficient to choose a satellite server at random *and* a request at random from the requests in that server. Indeed, one of the fundamental strengths of our technique is that it can be applied to distributed computing applications.

(d) Incrementality – Incremental algorithms are concerned with maintaining reachability information under edge insertions and deletions. Incremental algorithms for BFS have been

studied from both the theoretical [14] and the practical perspectives [8]. The exact complexity of this problem is unknown, although there have been attempts to categorize it [23]. In program analysis and verification, incremental algorithms for reachability analysis are of paramount importance [36, 35]. The Randomized BFS algorithm is incremental in nature and exploits the fact that the BFS tree is itself constituted of BFS sub-trees.

(e) Models of Computation – Frequently, an algorithm which is well-studied in one computational model (say the RAM model) is analyzed in a weaker model (say the pointer model) [27]; the purpose of such analyses is to enhance our understanding of problem complexity. In similar fashion, this paper asks whether order queries are necessary to accomplish FCFS fairness; we demonstrate that approximate FCFS fairness can be accomplished by using set membership queries only. From our perspective, this is a very surprising observation.

4. The Randomized Breadth First Search Algorithm

Algorithm 4.1 describes the workings of the Randomized Breadth First Search algorithm.

The algorithm is initialized as $d[s]=0$ and $d[v]=\infty, \forall v \neq s$. Further $Q=\{s\}$. It is important to note that the process of picking a vertex from Q involves extracting it from the same, i.e., the vertex picked is no longer in Q .

We reiterate that the above algorithm is the first of its kind to explicitly introduce randomization in the vertex selection process and therefore represents a fundamentally distinct design paradigm (cf. the Shortest path algorithms discussed in [2].)

Function *RANDOM – BFS*($G = \langle V, E \rangle, s, d, Q$)

Output : Set of all vertices $v \in V$ reachable from s , and $d[v]$, the shortest path from s to v , given all edges to be unit weight.

```
1. if  $|Q| \neq \emptyset$  then
2.   return
3. else
4.   Pick a vertex  $v$  uniformly and at random from  $Q$ 
5.   for (each vertex  $u$  adjacent to  $v$ ) do
6.     if  $(d[u] > d[v] + 1)$  then
7.        $d[u] = d[v] + 1$ 
8.       if  $(u \notin Q)$  then
9.         Add vertex  $u$  to  $Q$ 
       end if
     end if
   end for
6. end if
10. RANDOM – BFS( $G = \langle V, E \rangle, s, d, Q$ )
```

Algorithm 4.1: Randomized Breadth-First Search

4.1 Worst-case analysis

Let the graph have m edges and n vertices. Let $t(v_i)$ denote the time spent on processing vertex v_i . We know that $1 \leq d[v_i] \leq (n-1)$ for all $v_i \in V$ and hence v_i can be inserted in Q at most n times. Each time v_i is deleted from Q , we spend $degree(v_i)$ time in processing its neighbors. Accordingly, $t(v_i) \leq n \cdot degree(v_i)$ and hence the total time spent in processing all

the vertices is $T(n) = \sum_{i=1}^n t(i) = \sum_{i=1}^n n \cdot \text{degree}(v_i) = O(m \cdot n)$. It must be noted that the above analysis is extremely pessimistic and that our experiments show that $T(n)$ is a linear function of n .

We conjecture that $E[T(n)] \leq c_1 \cdot (n+m)$, for some fixed constant c_1 ; however, a formal proof will require the development of new theoretical techniques.

4.2 Correctness

Let $\delta(v)$ denote the true shortest path distance of vertex v from the source s . As discussed above, Algorithm 4.1 terminates, since each vertex enters Q at most n times.

The correctness of Algorithm 4.1 follows from the following lemma.

Lemma 4.1 If $d[v_i] > \delta(v_i)$, and $v_i \neq Q$, then v_i will be inserted into Q .

Proof: We use induction on the true distance $\delta(v_i)$. If $\delta(v_i) = 1$, the lemma is clearly true, since at the first call, the source s is extracted and all its neighbors are inserted into Q . Assume that the lemma is true, whenever $\delta(v_i) \leq k, k > 1$. Now observe the sequence of events when $\delta(v_i) = k + 1$. By the inductive hypothesis, all vertices v_j , such that $\delta(v_j) \leq k$, will be inserted into Q , till $d[v_j] = \delta(v_j)$. One of these vertices is v_r , the predecessor of v_i in the BFS tree; observe that $\delta(v_r) = k$. If it is the case that $d[v_i] \neq \delta(v_i)$, then $d[v_i] > (k + 1)$ and hence $d[v_i] > d[v_r] + 1$. When v_r is extracted from Q for the final time, Lines 6 through 9 of Algorithm 4.1, ensure that v_i is inserted into Q , if it does not already belong there. \square

Thus, as long as $d[v] > \delta(v)$ for some vertex v , Algorithm 4.1 will continue to recurse; since no vertex can be inserted into Q , more than n times, it follows that when $|Q| = \phi$, $d[v_i] = \delta(v_i), \forall i = 1, 2, \dots, n$.

It is important to note that Algorithm 4.1 is a complete procedure in that it can be used to detect the Single Shortest Paths in an arbitrarily weighted graph (both positive and negative weights on the arcs). The only modification is to lines 6 and 7, which should be replaced by: **if** $(d[u] > d[v] + c(v, u))$ **then** $d[u] = d[v] + c(v, u)$. The proof of this fact is similar to the above proof.

Theorem 4.1 Algorithm 4.1 can be modified to solve the Single Source Shortest Paths problem.

5. Experimental Study: Sequential Performance

This section presents the performance results of the sequential Random-BFS algorithm.

Our reference platform for evaluating sequential performance is a 3.2 GHz 64-bit Intel Xeon machine with 6GB memory and 1MB L2 cache.

5.1 Experimental Setup

We test our Random-BFS implementation on a variety of synthetic graph families. These generators and graph instances are part of the DIMACS Shortest Path Implementation Challenge network collection [13]:

- *random graphs*: Random graphs are generated by first constructing a Hamiltonian cycle, and then adding $m - n$ edges to the graph at random. The generator may produce parallel

edges as well as self-loops. By varying the parameters m and n , we can generate both sparse as well as dense random graphs.

- *mesh networks*: This synthetic generator produces regular two-dimensional square meshes, where $m = 4n$.
- *scale-free graphs*: We use the R-MAT graph model [6] to generate graphs with power-law degree distributions and small-world characteristics.

The above graph families are frequently used for the experimental evaluation of graph algorithms [33, 5]. Random graphs have a Poisson degree distribution, low diameter, and low clustering coefficients. BFS on sparse random graphs has poor cache locality, and so this is a hard test instance for performance comparison on cache-based architectures. Scale-free networks are sparse graphs characterized by low average distance, high local density, and heavy-tailed power-law degree distributions. In contrast, the two-dimensional mesh network is a regular graph with a high diameter. These three families differ in the number of BFS phases, as well as the average number of vertices in each phase.

The Random-BFS implementation differs significantly from that of the conventional Trad-BFS algorithm. In Trad-BFS, we maintain a queue of visited vertices and *enqueue* and *dequeue* are unit-cost operations. Each vertex enters the queue only once. However, in Random-BFS, the vertices are stored in a *set*, and a vertex is randomly picked on each iteration. Clearly, the worst case computational complexity and running time are higher than the conventional approach, as a vertex may enter the set of visited vertices multiple times.

The choice of the data structure to store the elements of S also determines the complexity. Note that the data structure has to support the *insert* and *remove* operations. Array-based bit vector set representations are easy to implement, but have significant problems – they are not space efficient, and the *randChoose* operation (randomly choosing an element from the set) has a worst case asymptotic cost of $O(n)$ (where n denotes the number of vertices). Similarly, a linked list is another data structure we could use, but irregular memory accesses on cache-based machines and the $O(n)$ worst-case running time would lead to skewed running time results. On the other hand, we could use specialized priority-queue data structures like Fibonacci heaps [19] or pairing heaps [39], which support expected-case unit-cost insertions and deletions. These are however tough to implement and do not lead to good results in practice on cache-based architectures.

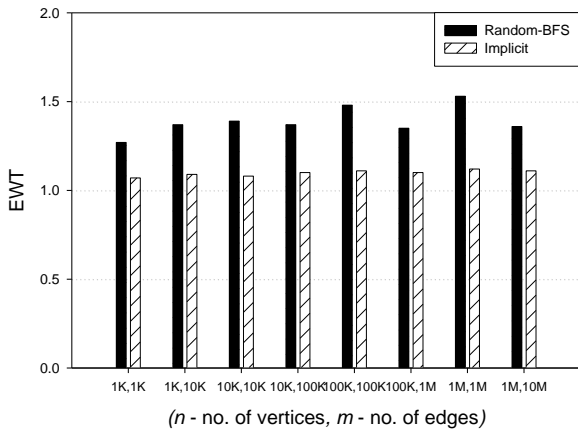
For running time comparison in this paper, we use an AVL-tree to represent the set S . The AVL-tree data structure is easy to implement and supports $O(\log n)$ insertions and deletions. To ensure that a vertex is randomly picked on each iteration, we associate each vertex with a secondary key, which is a unique uniformly-generated random number bounded by the current size of the set. So, on each iteration, a random integer in $[0, |S|]$ is picked, and the vertex corresponding to this key value is extracted from S . Note that we do not time the random number generation step while reporting the running time. It should be noted that we are not replacing queues with AVL trees; instead of AVL trees, we could have used linked lists and the results on expected wait time would have been identical. The purpose of this paper is to detail an empirical study of the number of times a vertex has to enter the set Q , when ordered queries are not available.

We report results averaged over ten runs, excluding the best and worst values and any outliers. Our first set of experiments estimate the randomization overhead in Algorithm 2.1. We do this by calculating the following metrics defined in Section 2.2:

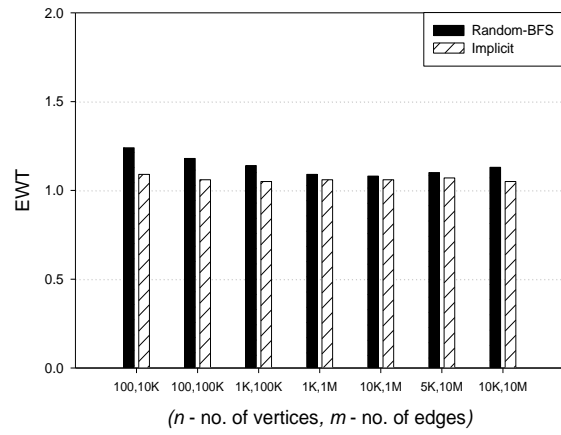
- *Expected wait time of a vertex* (EWT): the average number of times a vertex in the graph is inserted into the queue.
- *Expected wait time of protocol* (WTP): the maximum number of times any vertex in the graph is added to the queue.

Note that the EWT and WTP for a FCFS queue are 1. To evaluate Random-BFS performance, we also compute the metrics for an implicit queue representation, in which the requests are weighted according to the time spent in the queue. Figure 1 plots EWT for various graph instances, and the corresponding WTP values are reported in Figure 2.

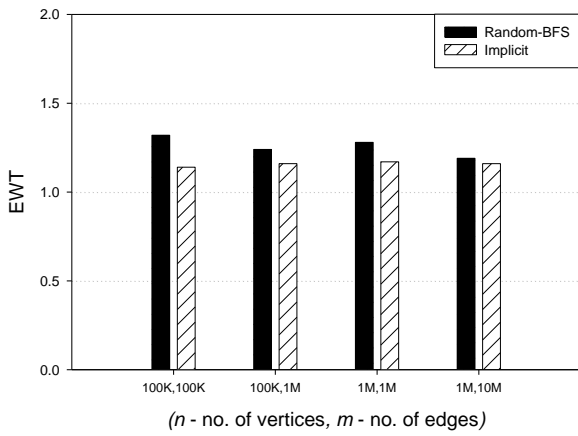
For performance comparison, we also implemented the traditional queue-based BFS (Alg. 4.1). Figure 3 plots the execution time of Trad-BFS and Random-BFS for synthetic graphs of different problem sizes.



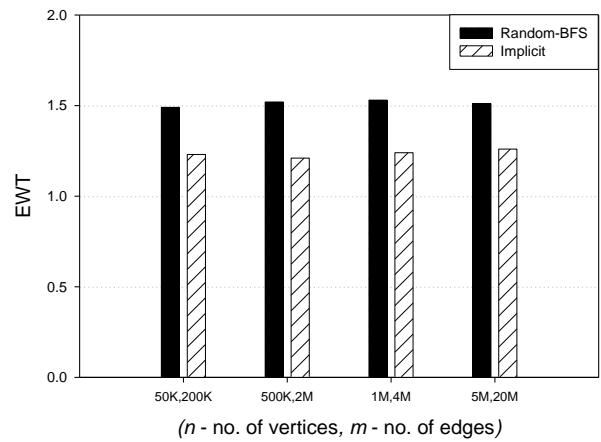
(a) Sparse random graphs



(a) Dense random graphs

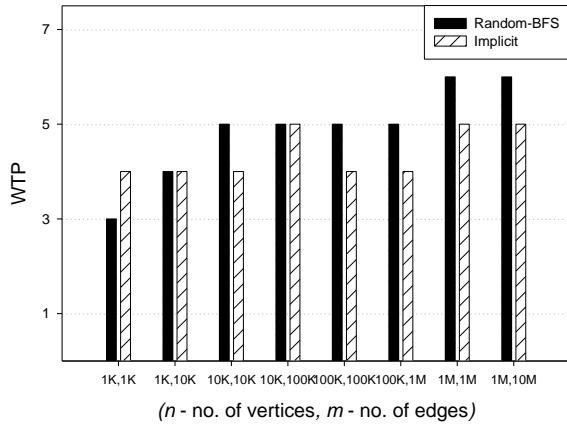


(c) Scale-free graphs

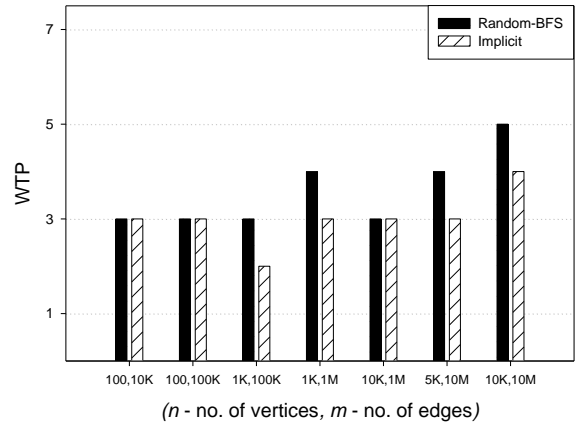


(d) Regular mesh networks

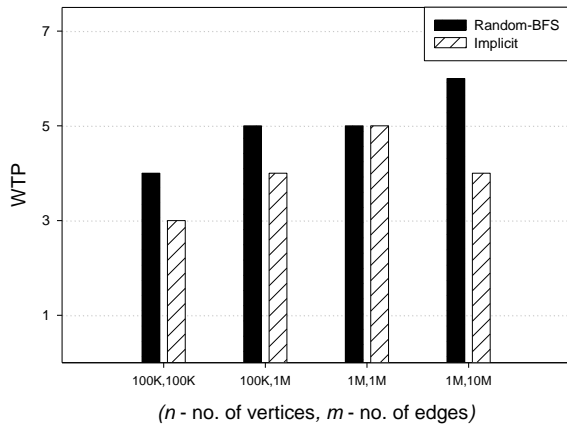
Figure 1: Expected Wait Time of a vertex (EWT) for Random-BFS and an Implicit queue representation.



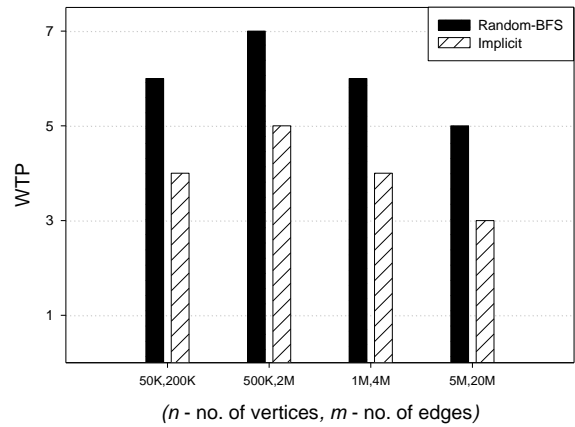
(a) Sparse random graphs



(a) Dense random graphs

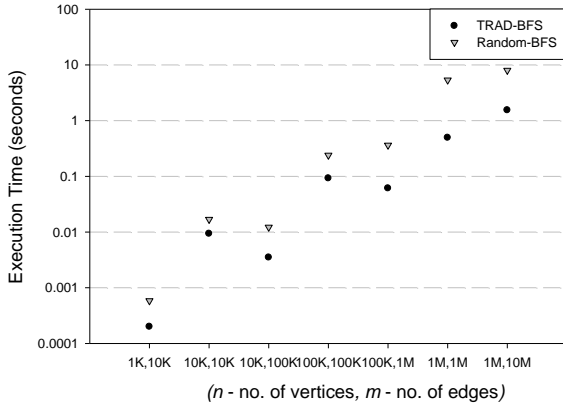


(c) Scale-free graphs

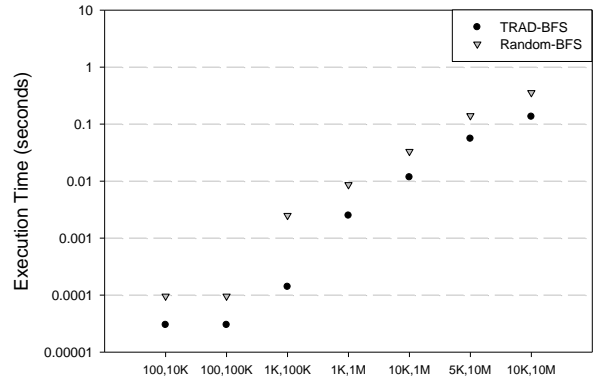


(d) Regular mesh networks

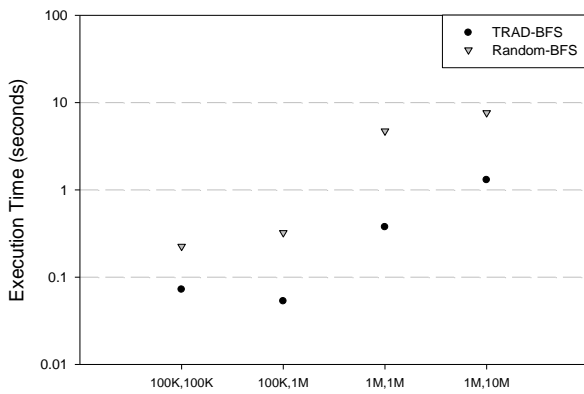
Figure 2: Expected Wait Time of the protocol (WTP) for Random-BFS and an Implicit queue representation.



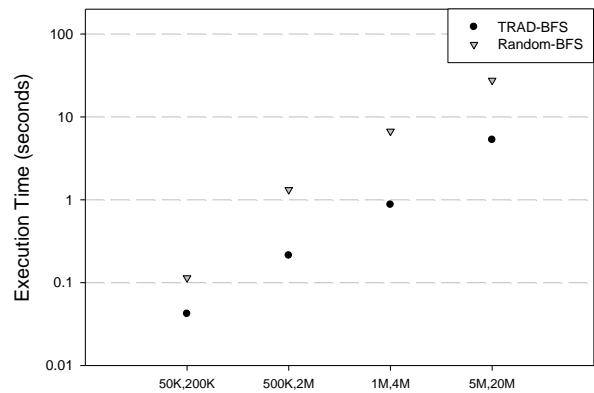
(a) Sparse random graphs



(a) Dense random graphs



(c) Scale-free graphs



(d) Regular mesh networks

Figure 3: Trad-BFS and Random-BFS execution time comparison for various graph instances.

5.2 Observations

5.2.1 FCFS policy metrics

For random graph instances in Figures 1 and 2, the expected wait time value (EWT) for Random-BFS varies from 1.13 to 1.56. This is slightly higher than the average EWT value for implicit queues. Also, note that EWT appears to be independent of the problem size for the graphs we considered. The expected wait time for the protocol (WTP) value varies from 3 to 6, with a slight increase for large instances. The WTP values for Random-BFS are comparable to the computed values for implicit queues. We observe a similar behavior in case of the mesh and scale-free networks (Figures 1(d) and 1(c) respectively) also. The EWT value for sparse instances (about 1.35 on an average) is higher than the value for dense graphs (averaging 1.15).

5.2.2 Execution Times

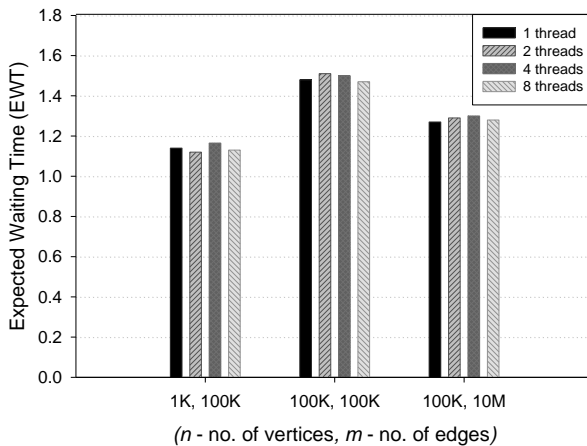
Figure 3 gives the running times of Trad-BFS and Random-BFS on the reference sequential platform. While Trad-BFS is faster than Random-BFS in all cases, the running times only differ by a constant factor. We observe that Trad-BFS is on an average six times faster than Random-BFS for sparse graphs, and thrice as fast for dense graphs. The running times are also directly correlated to the EWT and WTP values, which are both greater than 1. The execution time trends are similar for all three graph families.

6 Experimental Study: Parallel Performance

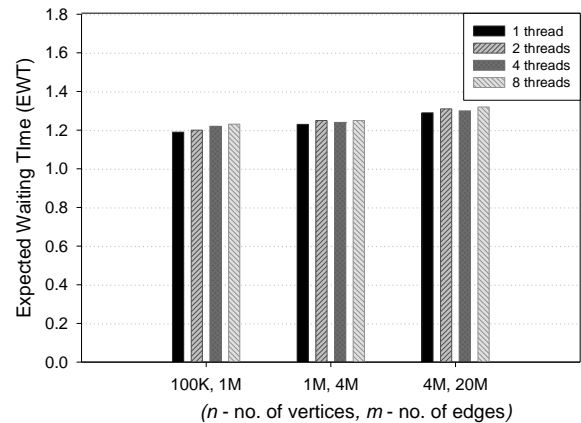
We also implement parallel shared-memory versions of Trad-BFS and Random-BFS. In case of Trad-BFS, we employ a level-synchronized approach to parallelization that exploits concurrency at two key steps:

1. All vertices at a given level (distance from source vertex) in the graph can be processed simultaneously, instead of just picking the vertex at the head of the queue.
2. Adjacencies of each vertex can be inspected in parallel.

In case of Random-BFS, we further assume that any vertex in the visited set can be picked.

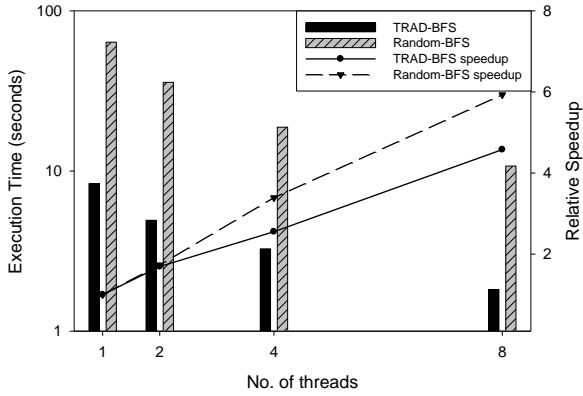


(a) Random graphs

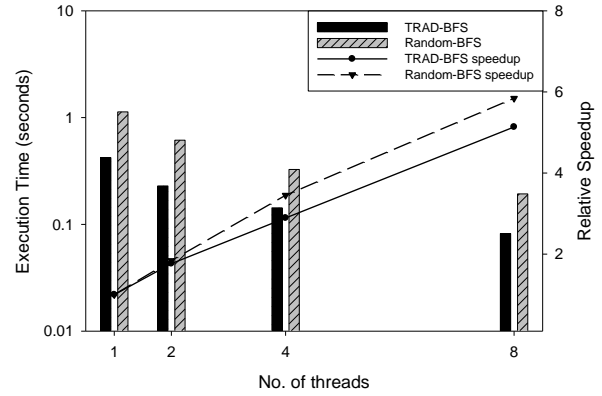


(b) Scale-free graphs

Figure 4: Parallel Expected Waiting Time (EWT) performance counts for various sparse and dense graph instances.



(a) Sparse scale-free graph
(1M vertices, 12M edges)



(b) Dense random graph
(100K vertices, 10M edges)

Figure 5: Parallel Execution time and Speedup comparison.

We report performance results on the Sun Fire T2000 multi-core server, with the Sun UltraSPARC T1 (Niagara) processor. This system has eight cores running at 1.0GHz, each of which is four-way multithreaded. The cores share a 3 MB L2 cache, and the system has a main memory of 16 GB. We use the same set of graph families discussed in the previous section.

As in the previous case, we first calculate EWT and WTP for different graph instances. Figure 4(a) depicts the value of EWT for parallel Random-BFS, as the number of processors is varied from 1 to 8. Figure 5 compares execution times of Trad-BFS and Random-BFS for various problem sizes, as the number of processors is varied from 1 to 8.

6.1 Observations

6.1.1 FCFS policy metrics

Figure 4 gives EWT for graph instances of different sizes from the random and scale-free families. We again note that EWT is higher for sparse graphs. But the key observation here is that there is no performance drop for parallel Random-BFS. There is very little variation in the EWT value across parallel runs.

6.1.2 Execution Times

Figure 5 gives the execution time and speedup achieved for multiprocessor runs. We observe that Trad-BFS is faster than Random-BFS up to 8 threads. The speedup on random graphs is lower than that on dense graphs, in case of both BFS and Random-BFS. The parallel performance on sparse random graphs is similar to the scale-free graph performance reported here. An important observation is that the relative speedup of Random-BFS is greater than Trad-BFS in both the cases. This is expected, as there is more concurrency in parallel Random-BFS than in parallel Trad-BFS. On larger multiprocessor systems, we expect the performance of Random-BFS to match Trad-BFS performance. From these results, we can also expect that Random-BFS would perform favorably on distributed memory systems, as there is no overhead of maintaining the FCFS queue.

7 Conclusions

The primary objective of this paper was to investigate whether FCFS fairness can be accomplished without queues. This is a fundamental problem in Program Verification, inasmuch as verifying the correctness of programs using queue structures is non-trivial. We have succeeded in showing (empirically) that randomization does indeed achieve the “queue” effect in that service requests are met on an almost First Come First Served basis. Our Randomized BFS algorithm is the first of its kind, in that it explicitly introduces randomization in the selection of requests to be serviced. This approach finds direction application in the implementation of distributed queues as discussed in Section 3. It is important to reiterate that the goal of this study is not to compare running time of different BFS algorithms, but whether membership queries can simulate order queries. Our implementational results indicate that such is indeed the case with a bearable loss in efficiency. However, in distributed applications, order queries will be significantly more expensive and therefore, our technique will have an immediate impact in that domain.

A number of interesting research problems have arisen out of this work:

- (a) As argued in Section 4, the RBFS algorithm is a complete procedure for finding Single Source shortest paths on arbitrarily weighted graphs. It would be instructive to study the performance profile of this algorithm for instances of the Single Source Shortest path problem.
- (b) We would like to implement this technique in actual program verification tools, such as the ones discussed in [8].

(c) We are currently engaged in developing an expected case analysis of the RBFS algorithm. Our goal is to analytically establish that the expected number of times that an arbitrarily chosen vertex is inserted into the set S is constant.

(d) We are also studying the performance of our algorithm in distributed applications, wherein the simplicity of our approach will lead to performance gains over the more traditional algorithms for implementing FCFS fairness.

Acknowledgments

The research of the first author was supported by the Air-Force Office of Scientific Research under contract FA9550-06-1-0050. The research of the second author was supported in part by NSF CAREER CCF-0611589, NSF DBI-0420513, ITR EF/BIO 03-31654, and NASA (NP-2005-07-375-HQ).

References

- [1] P.B.G.A. Silberschatz and G. Gagne, “Operating System Concepts,” John Wiley & Sons, 2005.
- [2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, “Network Flows: Theory, Algorithms and Applications,” Prentice-Hall, 1993.
- [3] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, and R.E. Tarjan, “Faster algorithms for the shortest path problem,” *Journal of the ACM*, 37(2):213–223, April 1990.

- [4] R. Alur and D.L. Dill, "A theory of timed automata," *Theoretical Computer Science*, 126(2):183–235, 25 April 1994.
- [5] D.A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)," *J. Parallel & Distributed Computing*, 65(9):994–1006, 2005.
- [6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," In *Proc. 4th SIAM International Conference on Data Mining*, Florida, USA, April 2004.
- [7] E.M. Clarke, "Automatic verification of sequential circuit designs," In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 163–166, Amsterdam, The Netherlands, April 1993, North-Holland.
- [8] C.L. Conway, K.S. Namjoshi, D. Dams, and S.A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," In *Computer-Aided Verification*, pages 449–461, 2005.
- [9] W. Cook, W.H. Cunningham, W. Pulleyblank, and A. Schrijver, "Combinatorial Optimization," John Wiley and Sons, 1998.
- [10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2001.
- [11] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz, "The Real-Time Operating System of MARS," *ACM Special Interest Group on Operating Systems*, 23(3):141–157, July 1989.
- [12] D. Dams and K.S. Namjoshi, "Automata as abstractions," In *VMCAI*, pages 216–232, 2005.

- [13] C. Demetrescu, A.V. Goldberg, and D. Johnson, “9th DIMACS Implementation Challenge: Shortest Paths,” 2005, [http://www.dis.uniroma1.it/~ challenge9/](http://www.dis.uniroma1.it/~challenge9/).
- [14] C. Demtrescu, “A new approach to dynamic all pairs shortest paths,” *Journal of the ACM*, 51(6):968–992, 2004.
- [15] E.W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, 1:269–271, 1959.
- [16] K.G. Doh and S.C. Shin, “Detection of information leak by data flow analysis,” *SIGPLAN Not.*, 37(8):66–71, 2002.
- [17] V.J. Fazio and R.D. Pose, “Distributed route initialization algorithms for the Monash secure RISC multiprocessor,” *HICSS*, 05:24, 1997.
- [18] N. DeFrancesco, A. Santone, and L. Tesei, “Abstract interpretation and model checking for checking secure information flow in concurrent systems,” *Fundam. Inf.*, 54(2-3):195–211, 2003.
- [19] M.L. Fredman and R.E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *J. ACM*, 34(3):596–615, 1987.
- [20] C. Ghezzi and M. Jazayeri, “Programming Language Concepts,” John Wiley & Sons, 3rd edition, 1997.
- [21] A.V. Goldberg, “Scaling algorithms for the shortest paths problem,” *SIAM Journal on Computing*, 24(3):494–504, June 1995.
- [22] M.T. Goodrich and R. Tamassia, “Algorithm Design: Foundations, Analysis and Internet Examples,” John Wiley & Sons, 2002.

- [23] W. Hesse, "The dynamic complexity of transitive closure is in $\text{dync}0$," *Theoretical Computer Science*, 3(296):473–485, 2003.
- [24] Y-W. Huang, F. Yu, C. Hang, C-H. Tsai, D-T. Lee, and S-Y. Kuo, "Securing web application code by static analysis and runtime protection," In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004, ACM Press.
- [25] C. Hymans and E. Upton, "Static analysis of gated data dependence graphs," In *SAS*, pages 197–211, 2004.
- [26] R. Joshi and K.R.M. Leino, "A semantic approach to secure information flow," *Science of Computer Programming*, 37(1-3):113–138, 2000.
- [27] D.R. Karger, P.N. Klein, and R.E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, 42(2):321–328, 1995.
- [28] S.T. Levi, S.K. Tripathi, S.D. Carson, and A.K. Agrawala, "The Maruti Hard Real-Time Operating System," *ACM Special Interest Group on Operating Systems*, 23(3):90– 106, July 1989.
- [29] K.C. Loudon, "Programming Languages: Principles and Practice," Brooks/Cole, 2002.
- [30] H. Mantel and A. Sabelfeld, "A unifying approach to the security of distributed and multi-threaded programs," *J. Computer Security*, 11(4):615–676, 2004.
- [31] R.C. Miller and K. Bharat, "Sphinx: a framework for creating personal, site-specific web crawlers," In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 119-130, Amsterdam, The Netherlands, The Netherlands, 1998, Elsevier Science Publishers B. V.

- [32] D. Moore, J. Leonard, D. Rus, and S. Teller, “Robust distributed network localization with noisy range measurements,” In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 50–61, New York, NY, USA, 2004, ACM Press.
- [33] S. Pettie, V. Ramachandran, and S. Sridhar, “Experimental evaluation of a new shortest path algorithm,” In *Proceedings 4th Workshop on Algorithm Engineering and Experiments (ALENEX02)*, pages 126–142, London, UK, 2002, Springer-Verlag.
- [34] M. Pinedo, “Scheduling: theory, algorithms, and systems,” Prentice-Hall, Englewood Cliffs, 1995.
- [35] G. Ramalingam and T.W. Reps, “An incremental algorithm for a generalization of the shortest-path problem,” *J. Algorithms*, 21(2):267–305, 1996.
- [36] G. Ramalingam and T.W. Reps, “On the computational complexity of dynamic graph problems,” *Theoretical Computer Science*, 158(1&2):233–277, 1996.
- [37] R. Raman, “Recent results in single-source shortest paths problem,” *SIGACT news*, 28:81–87, 1997.
- [38] S.M. Ross, “Probability Models,” Academic Press, Inc., 7th edition, 2000.
- [39] J.T. Stasko and J.S. Vitter, “Pairing heaps: experiments and analysis,” *Communications of the ACM*, 30(3):234–249, 1987.
- [40] M. Thorup, “Undirected single source shortest path in linear time,” In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS-97)*, pages 12–21, Los Alamitos, October 20–22 1997. IEEE Computer Society Press.