Accomplishing Approximate FCFS Fairness Without Queues

K. Subramani[1,⋆] and K. Madduri[2,⋆⋆]

[1] LDCSEE,
West Virginia University,
Morgantown, WV
ksmani@csee.wvu.edu
[2] College of Computing,
Georgia Institute of Technology,
Atlanta, GA
kamesh@cc.gatech.edu

Abstract. First Come First Served (FCFS) is a policy that is accepted for implementing fairness in a number of application domains such as scheduling in Operating Systems, 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, the Bellman-Ford FIFO implementation for finding single-source shortest paths, program verification and static analysis. The data structure used to implementing FCFS policies, the *queue*, suffers from two principal drawbacks, viz., non-trivial verifiability and lack of scalability. In case of large distributed networks, maintaining an explicit queue to enforce FCFS is prohibitively expensive. 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 design and analysis of a randomized protocol to implement approximate FCFS policies without queues. From the Software Engineering perspective, the techniques that are developed 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.

1 Introduction

FCFS is a policy used to ensure fairness in a number of application domains such as scheduling [1] and Operating Systems [2]. The motivating factor underlying this form of fairness, especially in the servicing of requests, is to preserve

⋆ 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 NSF CAREER CCF-0611589, NSF DBI-0420513, ITR EF/BIO 03-31654, and NASA (NP-2005-07-375-HQ).

S. Aluru et al. (Eds.): HiPC 2007, LNCS 4873, pp. 540–551, 2007.
© Springer-Verlag Berlin Heidelberg 2007

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 [3]. Additionally, queues do not scale well; in distributed applications, maintaining a FCFS queue is prohibitively expensive. 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.

The main advantage of our approach is 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.

2 Statement of Problem

The problem that we are interested in is as follows: $\mathbf{P_1}$: *Given a sequence of requests,* $S = \langle s_1, s_2, \ldots, s_n \rangle$*, 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 Breadth-First Search

We now argue that the FCFS problem is simulated by performing the Breadth-First Search on an arbitrary graph; note that we use the algorithm described in [4].

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 $\mathbf{Q}$ 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 $\mathbf{Q}$ before v_j. When a vertex is deleted from $\mathbf{Q}$, it is said to be serviced. Observe that $\mathbf{Q}$ implements FCFS fairness in that all requests at a particular level are serviced before requests at higher levels.

Function TRAD-BFS($G =< V, E >$, s)

1: {The output of the Algorithm is the 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}
2: Construct an empty queue Q and initialize it to s.
3: $d[s] = 0$
4: $color[s] = black$
5: **for** ($v \in V$, $v \neq s$) **do**
6: $d[v] = \infty$
7: $color[v] = white$
8: **end for**
9: **while** $|\mathbf{Q}| \neq \emptyset$ **do**
10: Let $v = head(\mathbf{Q})$
11: Delete $head(\mathbf{Q})$
12: **for** (each vertex u adjacent to v) **do**
13: **if** ($color[u] = white$) **then**
14: Add vertex u to $\mathbf{Q}$
15: $color[u] = black$
16: $d[u] = d[v] + 1$
17: **end if**
18: **end for**
19: **end while**

Algorithm 2.1. Traditional Breadth-First Search

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

In the traditional BFS protocol, $\mathcal{R}$ is a queue and each vertex is inserted precisely once in $\mathcal{R}$. 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 interested 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 the following design domains, viz.,

(a) Program Testing - The implementation of a queue mandates the testing of buffer overflow at each insertion and buffer underflow at each deletion, regardless of how the queue is implemented. It has been frequently observed that these bound checks are either ignored or incorrectly implemented (from a logical perspective) leading to program crashes. Our protocol on the other

hand, uses a simple "set" data structure with only membership queries permitted. This structure can be implemented by using a counter and is hence trivial to test. It is well-known that reasoning about heap-allocated structures is challenging; although tools exist for detecting errors in software [5], they are known to be imprecise when dealing with heap-allocated structures. It is to be noted that set semantics can be implemented statically, without utilizing the memory heap at run-time.

(b) 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 [6]. Reachability analysis is also used in model-checking LTL formulae [7], understanding secure information flow, and timed automata [8]. Likewise, reachability analysis is used to test properties of Timed Automata [9]. 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.

(c) Distributed Queues - Consider a web-server A, which is connected to a number of satellite servers $A_1, A_2, \ldots, A_n$. 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 time-stamp. 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 [10] and the practical perspectives. The exact complexity of this problem is unknown, although there have been attempts to categorize it [11]. In program analysis and verification, incremental algorithms for reachability analysis are of paramount importance [12]. The Randomized BFS algorithm is incremental in nature and exploits the fact that the BFS tree is itself constituted of BFS subtrees.

(e) Constraint Solving - Constraint solving is an integral component of Program Verification [13]. The approach described in this paper can be integrated into program verification tools such as the ones described in [14]. It is to be noted that our approach is easily and very efficiently parallelizable, which is the necessary in modern day constraint solvers.

4 The Randomized Breadth-First Search Algorithm

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

Function RANDOM-BFS($G =< V, E >$, s, $\mathbf{d}$, $\mathbf{Q}$)

1: {The output of the Algorithm is the 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}
2: **if** $|\mathbf{Q}| = \emptyset$ **then**
3:    **return**
4: **else**
5:    Pick a vertex v uniformly and at random from $\mathbf{Q}$
6:    **for** (each vertex u adjacent to v) **do**
7:       **if** $(d[u] > d[v] + 1)$ **then**
8:          $d[u] = d[v] + 1$
9:          **if** $(u \notin \mathbf{Q})$ **then**
10:             Add vertex u to $\mathbf{Q}$
11:          **end if**
12:       **end if**
13:    **end for**
14: **end if**
15: RANDOM-BFS($G =< V, E >$, s, $\mathbf{d}$, $\mathbf{Q}$)

Algorithm 4.1. Randomized Breadth-First Search

The algorithm is initialized as $d[s] = 0$ and $d[v] = \infty$, $\forall v \neq s$. Further, $\mathbf{Q} = \{s\}$.

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 [15].)

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 $\mathbf{Q}$ at most n times. Each time v_i is deleted from $\mathbf{Q}$, 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 $T(n) = \sum_{i=1}^{n} t(i) = \sum_{i=1}^{n} n \cdot degree(i) = O(m \cdot n)$. 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.

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 $\mathbf{Q}$ at most n times.

The correctness of Algorithm (4.1) follows from the following lemma.

Lemma 1. *If $d[v_i] > \delta(v_i)$, and $v_i \notin \mathbf{Q}$, then v_i will be inserted into $\mathbf{Q}$.*

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 $\mathbf{Q}$. 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 $\mathbf{Q}$, 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 $\mathbf{Q}$ for the final time, Lines 7 through 12 of Algorithm 4.1, ensure that v_i is inserted into $\mathbf{Q}$, if it does not already belong there. □

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 $\mathbf{Q}$, more than n times, it follows that when $|\mathbf{Q}| = \emptyset$, $d[v_i] = \delta(v_i)$, $\forall i = 1, 2, \ldots, n$.

It is important to note that Algorithm 4.1 is a complete procedure in that it can be used to detect the Single Source Shortest Paths in an arbitrarily weighted graph (both positive and negative weights on the arcs). The only modification is to lines 9 and 10, 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 and will be presented in an extended version of this paper [16].

Theorem 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. For implementation details, please refer to the extended version of this paper [16].

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 [17]:

- *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 [18] to generate graphs with power-law degree distributions and small-world characteristics.

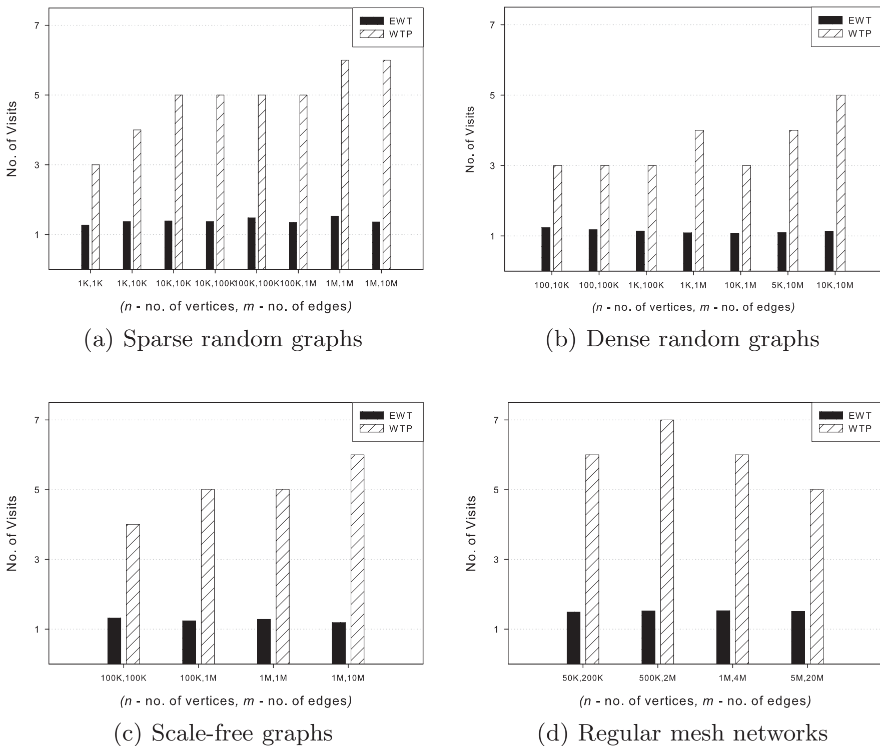
The above graph families are frequently used for the experimental evaluation of graph algorithms [19,20]. Random graphs have a Poisson degree distribution,

low diameter, and low clustering co-efficients. 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.

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 Alg. 2.1. We do this by calculating the following metrics defined in Sec. 2.1:

- *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.

Figure 1 plots EWT and WTP for various sparse and dense graph instances.



(a) Sparse random graphs

(b) Dense random graphs

(c) Scale-free graphs

(d) Regular mesh networks

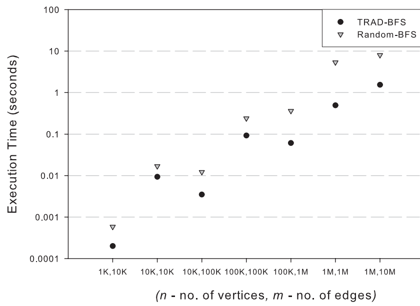
Fig. 1. RANDOM-BFS performance counts for various graph instances

For performance comparison, we also implemented the traditional queue-based BFS (Alg. 4.1). Figure 2 plots the execution time of Trad-BFS and Random-BFS for synthetic graphs of different problem sizes.

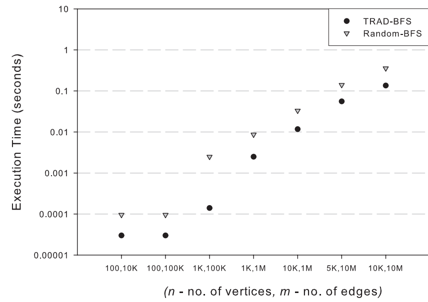
5.1 Observations

FCFS policy metrics. For random graph instances in Figure 1, the *expected wait time* value (EWT) varies from 1.13 to 1.56. 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. 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).

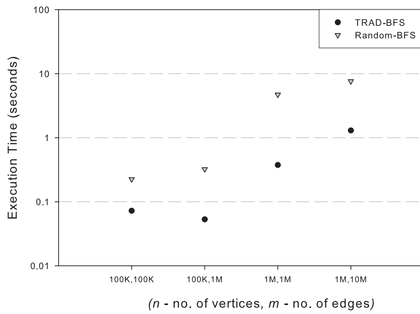
Execution Times. Figure 2 gives the running times of Trad-BFS and Random-BFS on the reference sequential platform. While Trad-BFS is faster than



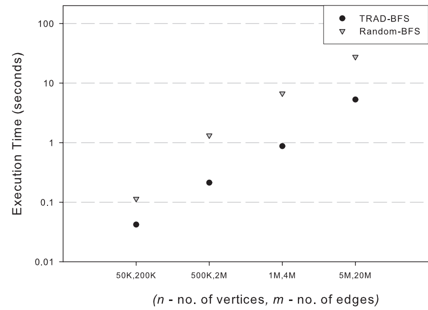
(a) Sparse random graphs



(b) Dense random graphs



(c) Scale-free graphs



(d) Regular mesh networks

Fig. 2. Trad-BFS and Random-BFS execution time comparison for various graph instances

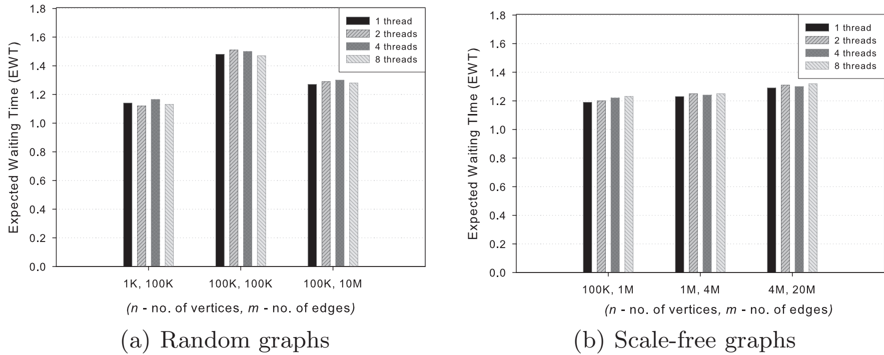
Random-BFS in all cases, the running times only differ by a constant factor. We observe that Trad-BFS is on an average six times faster than Random-BFS for sparse graphs, and thrice as fast for dense graphs. The running times are also directly correlated to the EWT and WTP values, which are both greater 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.

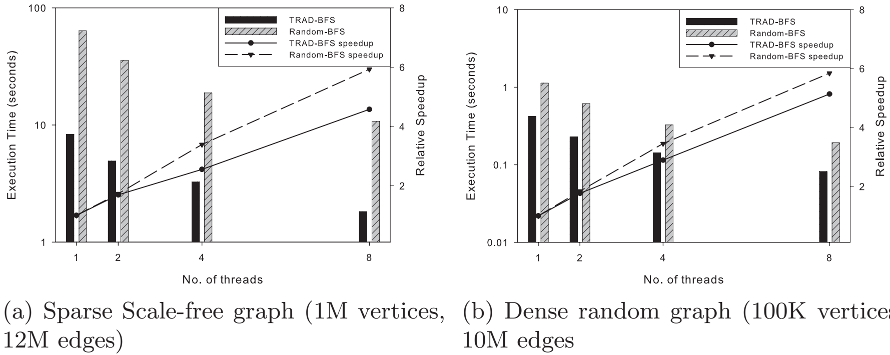


(a) Random graphs (b) Scale-free graphs

Fig. 3. Parallel Expected Waiting Time (EWT) performance counts for various sparse and dense graph instances

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 1.0 GHz, 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. Fig. 3(a) depicts the value of EWT for parallel Random-BFS, as the number of processors is varied from 1 to 8. Figure 4 compares execution times of Trad-BFS and Random-BFS for various problem sizes, as the number of processors is varied from 1 to 8.



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

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

Fig. 4. Parallel Execution time and Speedup comparison

6.1 Observations

FCFS policy metrics. Figure 3 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.

Running Time. Figure 4 gives the execution time and speedup acheived 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 Conclusion

The primary objective of this paper was to investigate whether FCFS fairness can be accomplished without queues, for the reasons described in previous sections. 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 direct 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 times of different BFS algorithms, but to determine whether membership queries can simulate order queries. Our implementation 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. Even on the sequential front, it is important to note that sets and membership queries are easier to implement and test when contrasted with queues and their associated operations.

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 [21].
(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.

References

1. Pinedo, M.: Scheduling: theory, algorithms, and systems. Prentice-Hall, Englewood Cliffs (1995)
2. Levi, S.T., Tripathi, S.K., Carson, S.D., Agrawala, A.K.: The `Maruti` Hard Real-Time Operating System. ACM Special Interest Group on Operating Systems 23(3), 90–106 (1989)
3. Goodrich, M.T., Tamassia, R.: Algorithm Design: Foundations, Analysis and Internet Examples. John Wiley & Sons, Chichester (2002)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE, pp. 82–87 (2005)
6. Louden, K.C.: Programming Languages: Principles and Practice. Brooks/Cole (2002)
7. Clarke, E.M.: Automatic verification of sequential circuit designs. In: Agnew, D., Claesen, L., Camposano, R. (eds.) Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL 1993), Amsterdam, The Netherlands, North-Holland. IFIP Transactions A: Computer Science and Technology, vol. 32, pp. 163–166 (1993)
8. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science (Fundamental Study) 126(2), 183–235 (1994)

9. Aceto, L., Bouyer, P., Burgueño, A., Larsen, K.G.: The power of reachability testing for timed automata. Theor. Comput. Sci. 300(1-3), 411–475 (2003)
10. Demtrescu, C.: A new approach to dynamic all pairs shortest paths. Journal of the ACM 51(6), 968–992 (2004)
11. Hesse, W.: The dynamic complexity of transitive closure is in dyntc^0. Theor. Comput. Sci. 3(296), 473–485 (2003)
12. Ramalingam, G., Reps, T.W.: On the computational complexity of dynamic graph problems. Theor. Comput. Sci. 158(1&2), 233–277 (1996)
13. Revesz, P.: Safe query languages for constraint databases. ACM Transactions on Database Systems 23(1), 58–99 (1998)
14. Revesz, P.: Introduction to Constraint Databases. Springer, Heidelberg (2002)
15. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms and Applications. Prentice-Hall, Englewood Cliffs (1993)
16. Subramani, K., Madduri, K.: A randomized, queueless algorithm for breadth-first search. International Journal of Computers and their Applications (accepted, 2007)
17. Demetrescu, C., Goldberg, A., Johnson, D.: 9th DIMACS implementation challenge – Shortest Paths (2005), http://www.dis.uniroma1.it/~challenge9/
18. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proc. 4th SIAM Intl. Conf. on Data Mining, Florida, USA (2004)
19. Pettie, S., Ramachandran, V., Sridhar, S.: Experimental evaluation of a new shortest path algorithm. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 126–142. Springer, Heidelberg (2002)
20. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). J. Parallel & Distributed Comput. 65(9), 994–1006 (2005)
21. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Computer-Aided Verification, pp. 449–461 (2005)