[59] G.S. Shedler, C. Tung. Locality in page reference strings. *SIAM Journal on Computing*, Vol. 1, pages 218–241, 1972.

[60] D.D. Sleator, R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, Vol. 28, pages 202–208, February 1985.

[61] D.D. Sleator, R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Vol. 32, No. 3, pages 652–686, July 1985.

[62] J.R. Spirn. *Program Behavior: Models and Measurements*. Elsevier Computer Science Library. Elsevier, Amsterdam, 1977.

[63] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, Vol. 22, pages 215–225, 1975.

[64] R.E. Tarjan. A class of algorithms that require nonlinear time to maintaing disjoint sets. *Journal of Computer and System Sciences* , 18:110-227. (1979).

[65] R.E. Tarjan. Amortized copmutational complexity. *SIAM J. Alg. Discrete Methods*, Vol. 6, No.2, April 1985.

[66] S. Vishwanathan. Randomized online coloring of graphs. *31st Annual Symposium on the Foundations of Computer Science*, 1990.

[67] H. Whitney. Non-seperable and planar graphs. *Transactions of the American Mathematical Society*, 34:339-362.

[68] A. C-C. Yao. Probabilistic computations: Towards a unified measure of complexity. *17th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.

[69] N. Young. Competitive paging as cache-size varies. *Proc. Second ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1991.

[47] L. Lovasz, M. Naor, I. Newman, A. Wigderson. Search problems in the decision tree model. To appear in *Proc.32nd Symposium on the Foundations of Computing Science*, 1991.

[48] L. Lovasz, M.E. Saks, W.A. Trotter. An on-Line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, Special Volume on Graph Theory and Combinatorics , pages 319-326, 1988.

[49] M.S. Manasse, L.A. McGeoch, D.D. Sleator. Competitive algorithms for on-line problems. *Journal of Algorithms*, Vol. 11, pages 208–230, 1990.

[50] L.A. McGeoch, D.D. Sleator. A strongly competitive randomized paging algorithm. Technical Report CMU–CS–89–122, Carnegie-Mellon University, Pittsburgh, PA, 1989. Submitted to *Algorithmica*.

[51] L.A. McGeoch, D.D. Sleator, C. Tomasi. Decision procedures for competitive algorithms. In preparation.

[52] D. Maier, S.C. Salveter. Hysterical B-trees. *Information Processing Letters*, Vol. 12, pages 199–202, 1981.

[53] D.A. Patterson. Reduced instruction set computers. *Communications of the ACM*, Vol. 28, No. 1, January 1985.

[54] P. Raghavan, M. Snir. Memory versus randomization in on-line algorithms. *16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 687–703. Springer-Verlag, July 1989. Revised version available as an IBM Research Report.

[55] N. Reingold, J. Westbrook, D. Sleator. Randomized competitive algorithms for the list update problem. To appear in *Algorithmica*.

[56] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, Vol. 19, pages 63-67, 1976.

[57] M. Saks. Personal communication.

[58] M. Szegedy. Personal communication, transmitted through [48].

[35] A.R. Karlin, M.S. Manasse, L. Rudolph, D.D. Sleator. Competitive snoopy caching. *Algorithmica*, Vol. 3, No. 1, pages 70–119, 1988.

[36] H. Karloff. Personal communication.

[37] R.M. Karp. Reducibility amoung combinatorial problems. R.E. Miller and J.W. Thatcher (ed.), *Complexity of Computer Computations*, Plenum Press, New York, pages 85–103.

[38] R.M. Karp, U.V. Vazirani, V.V. Vazirani. An optimal algorithm for on-line bipartite matching *Proc of the 22nd Symposium on the Theory of Computing*, pages 352–358, 1990.

[39] S. Khuller, S. Mitchell, V. Vazirani. Online algorithms for weighed matching and stable marriages. Technical Report TR 90-1143, Department of Computer Science, Cornell University,1990.

[40] H.A. Kierstead. The linearity if first-fit coloring of interval graphs. *SIAM Journal of Discrete Mathematics*, Vol. 1, No. 4, November 1988, pages 526-530.

[41] H.A. Kierstead, W.A. Trotter. An extremal problem in recursive combinatorics *Congressus Numerantium 33*, pages 143–153, 1981.

[42] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. One-level storage system. *IRE Trans. Elect. Computers*, Vol. 37, pages 223–235, 1962.

[43] D. J. Kleitman, D.B. West. Spanning trees with many leaves. *SIAM Journal of Discrete Mathematics*, Vol. 4, No. 1, pages 99–106, 1991.

[44] D.E. Knuth. *The Art of Computer Programming Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[45] A. Lempel, S. Even, I. Cederbaum. An algorithm for planarity testing of graphs. *Proc. Int. Symposium on Theory of Graphs; P. Rosenstiehl Ed.*, pages 215–232. Gordon and Breach, 1967.

[46] P.A.W. Lewis, G.S. Shedler. Empirically derived models for sequences of page exceptions. *IBM Journal of Research and Development*, Vol. 17, pages 86–100, 1973.

[23] E. Grove. The harmonic online k-server algorithm is competitive *Proc of the 23nd Symposium on the Theory of Computing*, pages 260-266, 1991.

[24] A. Gyarfas, J. Lehel. On-line and first fit colorings of graphs. *Journal of Graph Theory*, Vol. 12, No. 2, pages 217-227, 1988.

[25] W.C. Hobart, Jr., H.G. Cragon. Locality characteristics of symbolic programs. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 508–511, 1989.

[26] D. Hatfield, J. Gerald. Program restructuring for virtual memory. *IBM J. Sys. and Tech.*, Vol 10, pages 168–192, 1971.

[27] S. Huddleston, K. Melhorn. Robust balancing B-trees. *Proc. 5th GI-Conference on Theoretical Computer Science, Lecture Notes in Computer Science 104*, Springer-Verlag, New York, pages 234-244, 1981.

[28] S. Huddleston, K. Melhorn. A new data structure for representing sorted lists. *Acta Inform.*, 17:157-184, 1982.

[29] S. Irani. Coloring inductive graphs on-line. *the Proceedings for for the 31st Symposium on the Foundations of Computer Science*, pages 470–479, 1990, and invited to appear in a special issue of *Algorithmica* on on-line algorithms.

[30] S. Irani. Two results on the list update problem. To appear in Information Processing Letters.

[31] S. Irani, A. Karlin, S. Phillips. More on paging with locality of reference. Manuscript.

[32] S. Irani, N. Reingold, J. Westbrook, D. Sleator. Randomized competitive algorithms for the list update problem. *Proc. of the 2nd Annual Symposium on Discrete Algorithms*, pages 251–260, 1991.

[33] S. Irani, R. Rubinfeld. A competitive 2-server algorithm. To appear in *Information Processing Letters.*

[34] B. Kalyanasundaram, K. Pruhs. On-line weighted matching. *Proc. Second ACM-SIAM Symposium on Discrete Algorithms*, pages 234–240, 1991.

[10] M. Chrobak, H.J. Karloff, T. Payne, S. Viswanathan. New results on server problems. *Proc. First ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1990.

[11] M. Chrobak, L.L. Larmore. An optimal online algorithm for k servers on trees. *SIAM Journal of Computing*, Vol. 20, pages 144-148, 1991.

[12] M. Chrobak, L.L. Larmore. On fast algorithms for two servers. To appear in *Journal of Algorithms.*

[13] D. Coppersmith, P. Doyle, P. Raghavan, M. Snir. Random walks on weighted graphs, and applications to on-line algorithms. *Proc. of the 22nd ACM Symposium on Theory of Computing*, pages 369–377, 1990.

[14] F.K. Chung, R. Graham, M.E. Saks. A dynamic location problem for graphs. *Combinatorica*, Vol. 9, No. 2, pages 111-131, 1989.

[15] X. Deng, S. Mahajan. Infinite games, randomization, computability and applications to online problems *Proc of the 23nd Symposium on the Theory of Computing*, pages 289–298, 1991.

[16] P.J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering.*, Vol. 6, pages 64–84, 1980.

[17] U. Faigle, W. Kern, G. Turan. On the performance of on-line algorithms for partitioning problems. *Acta Cybernetica* Vol. 9, pages 107–119, 1989.

[18] D. Ferrari. The improvement of program behavior. *IEEE Computer*, Vol. 9, pages 39–47, November 1976.

[19] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, N. Young. On competitive algorithms for paging problems. To appear in *Journal of Algorithms*, 1991.

[20] A. Fiat, Y. Rabani, Y. Ravid. Competitive k-server algorithms. *Proceedings 31st Symposium on the Foundations of Comp. Sci. Vol. II*, pages 454-469, October 1990.

[21] P.A. Franaszek, T.J. Wagner. Some distribution-free aspects of paging performance. *Journal of the ACM*, Vol. 21, pages 31–39, 1974.

[22] B.A. Galler, M.J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, Vol. 7, pages 301–303, 1964.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, Vol. 5, pages 78–101, 1966.

[3] S. Ben-David, A. Borodin. A New measure for the study of on-line algorithms. Manuscript.

[4] S. Ben-David, A. Borodin, R.M. Karp, G. Tárdos, A. Wigderson. On the power of randomization in on-line algorithms. *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 379–388, 1990. To appear in *Algorithmica.*

[5] J.R. Bitner. Heuristics that dynamically organize data-structures for representing sorted lists. *SIAM Journal on Computing*, Vol. 8, pages 82–110, 1979.

[6] A. Borodin, S. Irani, P. Raghavan, B. Schieber. Competitive paging with locality of reference. *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 249–259, 1991.

[7] A. Borodin, N. Linial, M. Saks. An optimal online algorithm for metrical task systems. *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[8] J.L. Bentley, C. McGeogh. Worst-case analysis of self-orginizing sequential search heuristics. *Proc. 20th Allerton Conference on Communication, Control, and Computing.*

[9] C.G. Chaitin. Register allocation and spilling via graph coloring. *Proceedings of Sigplan Symposium on Computer Construction*, Sigplan, Note 17, 6, pages 98–105, June 1982.

5.3 Computational Restrictions

Most of the work on on-line algorithms has been information-theoretic. Traditionally the important question has been, given the fact that some part of the output must be produced without complete information about the input, what kind of solution quality can one hope to obtain? Although most of the on-line algorithms developed have been efficient, there are no explicit computational restrictions. Raghavan and Snir examine the question of limiting the space complexity of on-line algorithms [54], but what about time complexity? Can the computation time available to the algorithm affect the competitiveness that it can achieve? Is there a (natural) on-line problem for which (under some complexity assumptions) an algorithm with unlimited computation time can achieve a competitive ratio strictly better than one that must run in polynomial time?

rescheduling a process may greatly enhance the performance of a scheduling algorithm. Can one quantify the improvement in performance as a function of the amount of output an algorithm is allowed to alter?

5.2 New Measures

Chapter 3 dealt with some of the shortcomings of competitive analysis. Because competitive analysis is a worst-case analysis, the results are often more pessimistic that what is observed in practice. We use access graphs to restrict the adversary in a way that characterizes the typical input an algorithm would receive. Paging is not the only scenario in which this could be done. There may be other natural means to restrict the input in order to better represent the type of input one observes and to enhance the performance an on-line algorithm can achieve.

There is also some question as to whether competitive analysis is measuring the right quantity in the first place. There may be useful alternative measures to competitive analysis that evaluate how "good" an on-line algorithm is. The measure chosen to evaluate algorithms influences what kind of algorithms are developed. Ben-David and Borodin have suggested alternative measures [3], but there are interesting unresolved issues that remain. They present the following example that illustrates a weakness of competitive analysis: Consider an auto insurance plan with an annual premium of p, and let the cost of repairing a car be c. An algorithm must decide every year whether to buy auto insurance or not. The (c/p)-competitive algorithm that never buys insurance is optimal. This contradicts our intuition that insurance is good to have if cars break down frequently enough.

In some situations it seems unnatural to compare an on-line algorithm to the optimal off-line algorithm. If a problem to be solved is inherently on-line, we want to use the "best" on-line algorithm. Thus, the important comparison to be made is between on-line algorithms. Is there a way to compare two on-line algorithms without comparing each one to the best off-line algorithm? For most problems, it is not likely to be the case that there is an algorithm that performs better than all the others on every input. However, even if it is not true that an algorithm incurs a lower cost than another algorithm on every sequence, there may be some meaningful way of comparing the two algorithms.

Chapter 5

Conclusions and Future Directions

Aside from designing algorithms in the many areas where on-line problems arise, much of the work that remains in the field of on-line algorithms is to examine more general issues. The following are some suggestions for new directions in the study of on-line algorithms.

5.1 Lookahead

Chapter 4 examines the question of what kind of solution quality can be obtained when the on-line model is relaxed. Specifically, how much can an on-line algorithm benefit by seeing some portion of the future? The general question of lookahead is largely unexplored. For some questions, like the k-server problem, allowing the algorithm to see some fixed number of requests in advance does not help at all: if a k-server algorithm is allowed lookahead l, then the adversary can repeat each request l times and obtain the same lower bound as against an algorithm with no lookahead. For other problems, however, it may be possible to obtain better performance. For example, how can algorithms for list update, or other data structures, benefit by being able to see part of the request sequence before deciding how to service it?

An alternative approach to relaxing on-line constraints is to allow the algorithm to alter some portion of the output. In coloring a graph on-line, changing the color of key nodes may help the algorithm use fewer colors. In scheduling processes on-line, occasionally

4.6 Open Questions

The graph constructed by the adversary in the proof that $\Omega(d \log n)$ colors are needed to color d-inductive graphs is not necessarily chordal. Since trees are chordal, there is a lower bound of $\Omega(\log n)$ for the performance ratio on chordal graphs, but it remains open if for any d and any on-line coloring algorithm A, there is a chordal graph with chromatic number d such that A uses $\Omega(d \log n)$ colors to color the graph.

Finally for the question of lookahead, is there a class of graphs for which less lookahead is required before the on-line performance improves? Or is there a general technique to extend lower bounds on the number of colors required without lookahead to the number of colors required with lookahead? Can one prove for more classes of graphs that if the lower bound for the number of colors required by an on-line algorithm to color a graph, G in the class is $c(n, \chi(G))$, then for any $l \leq \frac{n}{c(n,\chi(G))}$, the lower bound still holds to within a constant factor?

we have that

$$\begin{aligned} & \sum_{j=1}^{i} 2d's_j - \sum_{j \geq pd'} 2d's_j \\ \geq & \sum_{j=d'(p-1)+1}^{d'(p-1)+l} [\bar{F}F_j(p) + d'X_p] \\ \geq & \ 2d'lX_p \end{aligned}$$

Since $s_j = X_p$ for $i < j < pd'$,

$$\sum_{j=1}^{pd'-1} s_j - \sum_{j \geq pd'} s_j \geq (d'-1)X_p$$

- Case 2: $i \leq d'(p-1)$ Since $s_j - X_p = 0$ for $d'(p-1) < j < pd'$, and $s_j = 0$ for $j \geq pd'$,

$$(\sum_{j=1}^{pd'-1} s_j - \sum_{j \geq pd'} s_j - (d'-1)X_p) = \sum_{j=1}^{d'(p-1)} s_j \geq 0.$$

■

To prove that Property 1' holds, for $i \leq pd' - 1$, $d'(X_p - s_i)$ slots are used up and at most s_i nodes are added to color i. Thus,

$$\begin{aligned} A_i(p) & \leq A_i(p-1) - d'(X_p - s_i) + d's_i \\ & = A_i(p-1) - d'X_p + 2d's_i \end{aligned}$$

Thus using Lemma 29, Property 1' holds before the colors are reordered.

We can prove the following lemma which is sufficient to prove that Property 1' holds after the colors have been renumbered.

Lemma 31 *Suppose $j < i$ and $A_j(p) < A_i(p)$ before the colors are reordered, then*

(1) if $i \leq pd' - 1$, then for any $l \leq i - 1, A_i(p) \leq A_l(p-1) - d'X_p + 2d's_l$

(2) if $i > pd' - 1$, then for any $l \leq pd' - 1, A_i(p) \leq A_l(p-1) - d'X_p + 2d's_l$

The proof of Lemma 31 parallels the proof of Lemma 27.

Lemma 29 *If Property 1' and 2' hold after phase $p-1$ and the slack variables are chosen as indicated above, then for all i where $1 \leq i \leq pd'-1$, $s_i \geq 0$ and*

$$(2d'-1) + \sum_{j=1}^{i} \bar{FF}_j(p) \geq \sum_{j=1}^{i} [A_j(p-1) - d'X_p + 2d's_j]$$

The proof for Lemma 29 is similar to the proof for Lemma 26. The adversary chooses the edges for phase p so that if $i \leq pd'-1$, then $X_p - s_i$ cliques are adjacent to color i nodes. (If a clique is adjacent to color-i nodes, then all the nodes in the clique are adjacent to nodes that have been assigned the color i.) If $i > pd'$, then s_i cliques in phase p are adjacent to color-i nodes. The adversary uses $d'^2(p-1)X_p$ slots in phase p when playing against First Fit. He uses $d'(\sum_{j=1}^{pd'-1}(X_p - s_j) + \sum_{j \geq pd'} s_j)$ against algorithm A. The adversary then adds $d'(\sum_{j=1}^{pd'-1} s_j - \sum_{j \geq pd'} s_j - (d'-1)X_p)$ edges between any node in phase p and any node from a previous phase. Property 2' follows from Lemma 30:

Lemma 30 $\sum_{j=1}^{pd'-1} s_j - \sum_{j \geq pd'} s_j - (d'-1)X_p \geq 0$.

Proof: Let i be the largest integer such that $i \leq pd'-1$ and $s_i < d'X_p$.

- Case 1: $i > d'(p-1)$. Let $i = d'(p-1) + l$.

$$\sum_{j=1}^{i} s_j \geq \sum_{j=1}^{i} \frac{\bar{FF}_j(p) + d'X_p - A_j(p-1)}{2d'}$$

$$\sum_{j \geq pd'} s_j \leq \sum_{j \geq pd'} \frac{A_j(p-1)}{2d'}$$

and

$$\sum_{j=1}^{i} 2d's_j - \sum_{j \geq pd'} 2d's_j$$

$$\geq \sum_{j=1}^{i} [\bar{FF}_j(p) + d'X_p - A_j(p-1)] - \sum_{j \geq pd'} A_j(p-1)$$

$$\geq \sum_{j=1}^{i} [\bar{FF}_j(p) + d'X_p] - \sum_{j \geq 1} A_j(p-1)$$

Since $\bar{FF}_j(p) + d'X_p = \bar{FF}_j(p-1)$ for $j \leq d'(p-1)$, and

$$\sum_{j \geq 1} A_j(p-1) = \sum_{j=1}^{d'(p-1)} \hat{FF}_j(p-1) \leq \sum_{j=1}^{d'(p-1)} \bar{FF}_j(p-1)$$

$i \leq p$, $FF_{i-1}(p) \geq FF_i(p) + 2d$. Now $\hat{FF}_i(p)$ has the property that the colors can be grouped into blocks of d' consecutive colors. Colors within a block have the same number of slots, and a color from a lower block has at least $4d'^2$ more slots than a color from a higher block. In order to be able to use the same method as before, we define a new distribution $\bar{FF}$. $\bar{FF}$ has the property that $\bar{FF}_i(p) \geq 4d' + \bar{FF}_{i+1}(p)$. It is defined as follows:

$$\bar{FF}_i(p) = \hat{FF}_i(p) + 4d'(d' - i \pmod{d'})$$

Furthermore we have the property that

$$\hat{FF}_j(p) \leq \bar{FF}_j(p) \leq [4d'^2 + \hat{FF}_j(p)] \tag{4.10}$$

Because $\bar{FF}_i(p) \geq 4d'^2$, for all $i \leq pd'$,

$$\sum_{j=1}^{\lfloor pd'/2 \rfloor} \bar{FF}_j(p) \leq \sum_{j=1}^{\lfloor pd'/2 \rfloor} [\hat{FF}_j(p) + 4d'^2] \tag{4.11}$$

$$\leq \sum_{j=1}^{\lfloor pd'/2 \rfloor} \hat{FF}_j(p) + \sum_{j=\lfloor pd'/2 \rfloor + 1}^{pd'} 4d'^2 \tag{4.12}$$

$$\leq \sum_{j \geq 1} \hat{FF}_j(p) \tag{4.13}$$

Now when playing against an algorithm A, we would like to ensure the following two properties:

- **Property 1':** $\sum_{j=1}^{i} A_j(p) \leq (2d' - 1) + \sum_{j=1}^{i} \bar{FF}_j(p)$ for any $i \leq pd'$
- **Property 2':** $\sum_{j \geq 1} A_j(p) = \sum_{j \geq 1} \hat{FF}_j(p)$

From the two properties and the bound from line 4.13, we know that A has used at least at least $pd'/2$ colors after phase p. The slack variables for phase p are chosen after phase $p-1$ as follows: For $i = 1$ to $pd' - 1$,

$$s_i \leftarrow min_{pd'-1 \geq k \geq i} \left\{ X_p, \left\lceil \sum_{j=1}^{k} \frac{\bar{FF}_j(p) + d'X_p - A_j(p-1)}{2d'} \right\rceil - \sum_{j=1}^{i-1} s_j \right\}$$

If $s_{pd'-1} < X_p$ then for $i \geq pd'$,

$$s_i \leftarrow \min \left\{ X_p, \left\lfloor \frac{A_i(p-1)}{2d'} \right\rfloor \right\}$$

Otherwise, $s_i = 0$ for all $i \geq pd'$.

We use the following lemma.

We first prove the weaker bound of $\Omega(\ min\ \{d \log n, \frac{n}{l}\})$. Let $c = min\{\frac{n}{l}, d \log n\}$. The graph is chosen exactly according to the strategy for Theorem 17 except that the adversary inserts l independent nodes after each phase. This means that the algorithm with lookahead l cannot see any of the nodes in a subsequent phase. The adversary does not count the slots from the l independent nodes in determining where to add edges for the next phase. Thus, when phase $p-1$ is over, the adversary can already determine the edges for the phase p nodes. This implies that the adversary can force an algorithm with lookahead l to use c colors. The number of nodes in the graph is bounded by $2n$.

To prove the stronger bound we alter the adversary strategy in Theorem 17 so that there are $\Theta(\log n)$ phases. After the p^{th} phase, any on-line algorithm will have used $\Omega(pd)$ colors. Then when playing against an algorithm with lookahead, the adversary adds l nodes in between each phase, but there are now only $O(\log n)$ phases. Let $d' = \lfloor d/2 \rfloor$. In phase p, the adversary presents X_p d'-cliques. Now each node has at least d' slots. In the proof we attribute exactly d' slots to each node when it first arrives. If the size of the graph is n, then the adversary chooses the number of phases, c, to be $min\{\frac{n}{l}, \log(n/d^2)\}$. He chooses the X_i's as follows: $X_c = 4d'^2$ and $X_i = 2X_{i+1} + 4d'$. $O(n)$ nodes are used in the phases, and at most n nodes are used between phases. Since d' colors are used per phase, A will use $\Omega(\ min\ \{d \log n, \frac{dn}{l}\})$ colors.

The nodes within a clique are all adjacent to the same set of nodes. When the adversary plays against First Fit, each node in each clique presented in phase p is adjacent to nodes assigned colors $\{1, \ldots, (p-1)d'\}$. First Fit uses d' colors per phase. Let $\hat{FF}_i(p)$ be the number of slots of color i First Fit has after phase p under the new strategy.

Lemma 28 *For any phase $p \leq c$, and for every $i \leq p-1$ and for any $1 \leq j, k \leq d'$,*

(A) $\hat{FF}_{(i-1)d'+j}(p) \geq 4d'^2 + \hat{FF}_{id'+k}(p)$ *and* $\hat{FF}_{(p-1)d'+k}(p) = d'X_p$

(B) $\hat{FF}_{id'+k}(p) = \hat{FF}_{id'+j}(p)$

(C) *Each node in a clique presented in phase p is assigned a color in* $\{(p-1)d'+1, \ldots, pd'-1, pd'\}$.

Proof: Similar to lemma 25. ■

The adversary would like to use the same strategy as before in choosing the edges for the upcoming phase. However, the distribution $FF_i(p)$ had the property that for every

Since $s_{p-1} < X_p$, from the argument in Case 1b,

$$FF_{p-1}(p-1) - d \leq A_{p-1}(p-1) + (d+1)s_{p-1} \tag{4.9}$$

and

$$A_i(p) \leq A_{p-1}(p-1) - X_p + (d+1)s_{p-1}$$

■

4.5 On-line Coloring with Lookahead

Suppose the on-line model is slightly altered by allowing the algorithm to see the next l nodes before assigning a color to the present node. The adversary presents the nodes one at a time. The nodes are numbered in the order in which the adversary presents them. When each node is presented, its edges to previously presented nodes are also given. Vertex i must be assigned a color before node $i + l + 1$ is presented. In this case, we say that the graph is colored on-line with *lookahead* l.

Theorem 18 *If G is a d-inductive graph on n nodes , then G can be colored on-line with lookahead l using $O(\ min\{d \log n, \frac{dn}{l}\})$ colors.*

Proof: If $d \log n < (d+1)\frac{n}{l}$, then ignore the lookahead and use FF to color the graph. By Theorem 16, FF will use $O(d \log n)$ colors.

If $d \log n \geq (d+1)\frac{n}{l}$, then divide the nodes into $\frac{n}{l}$ groups of l consecutive nodes. (Consecutive in the order presented). The algorithm can see the subgraph induced by the nodes in an entire group before having to assign a color to the first node in the group. Since the subgraph induced by each group is d-inductive, it can be colored using $d+1$ colors. At most $d+1$ colors are used for every group, and at most $(d+1)\frac{n}{l}$ total colors are used. ■

The following theorem shows that this is asymptotically the best possible.

Theorem 19 *For every on-line graph coloring algorithm A with lookahead l, and for every d, there is a family of d-inductive graphs $\mathcal{G}$ such that for every $n \geq d^3$, there is a $G \in \mathcal{G}$ where G has n nodes and $A(G) = \Omega(\ min\ \{d \log n, \frac{dn}{l}\})$*

Using Lemma 25, we can upper bound $FF_i(p-1)+d$ by $FF_{i-1}(p-1)-d$. To upper bound $FF_{i-1}(p-1)$, we know from the choice of the s_i's,

$$0 \leq \left\lceil \sum_{j=1}^{i-2} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-2} s_j \tag{4.7}$$

Subtracting Line 4.7 from Line 4.4 and rearranging,

$$FF_{i-1}(p-1) - d \leq A_{i-1}(p-1) + (d+1)s_{i-1} \tag{4.8}$$

Putting Inequalities 4.6 and 4.8 together,

$$A_i(p-1) + (d+1)s_i \leq A_{i-1}(p-1) + (d+1)s_{i-1}$$

We can now bound the number of slots of color i after phase p.

$$\begin{aligned} A_i(p) &\leq A_i(p-1) - (X_p - s_i) + ds_i \\ &\leq A_{i-1}(p-1) - X_p + (d+1)s_{i-1} \end{aligned}$$

- Case 3a: $i \geq p$ and $s_{p-1} = X_p$. Since at most dX_p slots can be added to any color,

$$\begin{aligned} A_i(p) &\leq A_i(p-1) + dX_p \\ &\leq A_{p-1}(p-1) + dX_p \\ &= A_{p-1}(p-1) - X_p + (d+1)s_{p-1} \end{aligned}$$

- Case 3b: $i \geq p$ and $0 < s_{p-1} < X_p$.

$$s_i = \left\lfloor \frac{A_i(p-1)}{d+1} \right\rfloor$$

$$A_i(p-1) \leq (d+1)s_i + d$$

To bound the number of slots of color i after phase p, $X_p - s_i$ nodes can be added and s_i slots are used up. Thus,

$$\begin{aligned} A_i(p) &\leq A_i(p-1) + d(X_p - s_i) - s_i \\ &\leq dX_p + d \\ &\leq FF_p(p) + d \leq FF_{p-1}(p) - d \\ &= FF_{p-1}(p-1) - X_p - d \end{aligned}$$

If $s_i = 0$ then all X_p nodes in phase p are adjacent to a node of color i, and color i can not gain more slots relative to color j. Thus, it can not be the case that for $j < i$, $A_j(p) < A_i(p)$.

- Case 2a: $i \leq p-1$ and $s_{i-1} = X_p$.

$$\begin{aligned} A_i(p) &\leq A_i(p-1) - (X_p - s_i) + ds_i \\ &\leq A_i(p-1) - X_p + (d+1)s_i \\ &\leq A_{i-1}(p-1) - X_p + (d+1)X_p \\ &= A_{i-1}(p-1) - X_p + (d+1)s_{i-1} \end{aligned}$$

- Case 2b: $i \leq p-1$ $s_{i-1} < X_p$ and $s_i > 0$. s_{i-1} was chosen so that for some k where $i-1 \leq k \leq p-1$,

$$0 = \left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-1} s_j$$

If $k \geq i$, then $s_i = 0$ because s_i is chosen so that

$$s_i \leq \left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-1} s_j .$$

So we have that

$$0 = \left\lceil \sum_{j=1}^{i-1} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-1} s_j \tag{4.4}$$

From the choice of the s_i's,

$$s_i \leq \left\lceil \sum_{j=1}^{i} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-1} s_j \tag{4.5}$$

Subtracting Line 4.4 from Inequality 4.5,

$$s_i \leq \left\lceil \frac{FF_i(p-1) - A_i(p-1)}{d+1} \right\rceil .$$

Rearranging,

$$A_i(p-1) + (d+1)s_i \leq FF_i(p-1) + d \tag{4.6}$$

$$= \sum_{j=1}^{i} [A_j(p-1) - X_p + (d+1)s_j] \tag{4.3}$$

Line 4.3 and Lemma 26 imply that Property 1 holds before the colors have been renumbered according to their new number of slots. We use the following lemma to prove that Property 1 holds after the colors have been renumbered.

Lemma 27 *Suppose $j < i$ and $A_j(p) < A_i(p)$ before the colors are reordered, then*

(1) if $i \leq p-1$, then for every $l \leq i-1, A_i(p) \leq A_l(p-1) - X_p + (d+1)s_l$

(2) if $i > p-1$, then for every $l \leq p-1, A_i(p) \leq A_l(p-1) - X_p + (d+1)s_l$

Lemma 27 together with Line 4.3 implies that for $i \leq (p-1)$, $\sum_{j=1}^{i} A_j(p) \leq \sum_{j=1}^{i} A_j(p-1) - X_p + (d+1)s_j$ even after the colors are reordered according to their new number of slots. Thus using Lemma 26, Property 1 for $i \leq p-1$ holds after renumbering. Property 1 holds for $i = p$ because First Fit and A have the same total number of slots, and First Fit has only used colors $\{1, \ldots, p\}$,

$$\sum_{i=1}^{p} A_i(p) \leq \sum_{i \geq 1} A_i(p) = \sum_{i \geq 1} FF_i(p) = \sum_{i=1}^{p} FF_i(p)$$

Thus Property 1 still holds after the colors are reordered.

Proof of Lemma 27: We prove that if $j < i$ and if $A_j(p) < A_i(p)$ before the colors are reordered, then

$$(1) \text{ for } i \leq p-1, \; A_i(p) \leq A_{i-1}(p-1) - X_p + (d+1)s_{i-1}$$

$$(2) \text{ for } i > p-1, \; A_i(p) \leq A_{p-1}(p-1) - X_p + (d+1)s_{p-1}$$

In addition we prove that for every $i \leq p-1$,

$$A_i(p-1) - X_p + (d+1)s_i \leq A_{i-1}(p-1) - X_p + (d+1)s_{i-1}.$$

There are five cases.

- Case 1: $s_i = 0$. Since $A_i(p-1) \leq A_{i-1}(p-1)$, then it is clear that

$$A_i(p-1) - X_p + (d+1)s_i \leq A_{i-1}(p-1) - X_p + (d+1)s_{i-1}.$$

we have that

$$\left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i+1} s_j \geq 0$$

■

The adversary chooses the edges for the next phase as follows: For $1 \leq i \leq p-1$, $X_p - s_i$ of the nodes in phase p are adjacent to nodes of color i. For $i \geq p$, s_i of the nodes in phase p are adjacent to nodes of color i. If $\sum_{i=1}^{p-1} s_i > \sum_{i \geq p} s_i$, the adversary then adds $\sum_{i=1}^{p-1} s_i - \sum_{i \geq p} s_i$ extra edges between any node in phase p and any node from phase $1, \ldots, p-1$.

When the adversary plays against First Fit, $(p-1)X_p$ edges are used in phase p, thus Property 2 holds if the adversary uses the same number of edges against algorithm A. Exactly $(p-1)X_p$ edges have been added in phase p against A, if $\sum_{i=1}^{p-1} s_i - \sum_{i \geq p} s_i \geq 0$.

If $s_{p-1} = X_p$, the claim clearly follows because $\sum_{i \geq p} s_i = 0$. Otherwise, if $s_i < X_p$, then from from the choice of the s_i's,

$$\begin{aligned}
\sum_{i \geq p} s_i &\leq \sum_{i \geq p} \frac{A_i(p-1)}{(d+1)} \\
\sum_{i=1}^{p-1} s_i &\geq \sum_{i=1}^{p-1} \frac{FF_i(p-1) - A_i(p-1)}{(d+1)} \\
\sum_{i=1}^{p-1} s_i - \sum_{i \geq p} s_i &\geq \sum_{i \geq 1} \frac{FF_i(p-1) - A_i(p-1)}{(d+1)} = 0
\end{aligned}$$

To prove that Property 1 holds after phase p, we first evaluate how the number of slots of each color has changed after algorithm A has colored all the nodes added in phase p. We verify that the claim holds before reordering the colors according to their new number of slots. Then we verify that the claim holds after the colors are renumbered. For each color $i \leq p-1$, $X_p - s_i$ slots are used up and at most ds_i new slots are added.

$$\sum_{j=1}^{i} A_j(p) \leq \sum_{j=1}^{i} [A_j(p-1) - (X_p - s_j) + ds_j] \tag{4.2}$$

For $i = 1$ to $p-1$,

$$s_i \leftarrow min_{p-1 \geq k \geq i} \left\{ X_p, \left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i-1} s_j \right\}$$

If $s_{p-1} < X_p$ then for $i \geq p$,

$$s_i \leftarrow \min \left\{ X_p, \left\lfloor \frac{A_i(p-1)}{d+1} \right\rfloor \right\}$$

Otherwise, $s_i = 0$ for all $i \geq p$.

We prove the following lemma

Lemma 26 *If Properties 1 and 2 hold after phase $p-1$ and the slack variables are chosen as indicated above, then for all i where $1 \leq i \leq p-1$, $s_i \geq 0$ and*

$$d + \sum_{j=1}^{i} FF_j(p) \geq \sum_{j=1}^{i} [A_j(p-1) - X_p + (d+1)s_j]$$

Proof: Since $FF_j(p-1) - X_p = FF_j(p)$ for all $i \leq p-1$, we just have to prove $s_i \geq 0$ and

$$d + \sum_{j=1}^{i} FF_j(p-1) \geq \sum_{j=1}^{i} [A_j(p-1) + (d+1)s_j].$$

Since all variables are integers, this follows if $s_i \geq 0$ and

$$\left\lceil \sum_{j=1}^{i} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i} s_j \geq 0.$$

The proof is by induction on i. We prove that for any k where $p-1 \geq k \geq i$,

$$\left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i} s_j \geq 0 \tag{4.1}$$

Property 1 ensures that the claim is true for $i = 0$. Now assume the claim is true for i. Since s_{i+1} is chosen to be the minimum over k of the value in Line 4.1, then s_{i+1} is just the minimum of a set of non-negative numbers and hence, $s_{i+1} \geq 0$. Furthermore, since for any k where $p-1 \geq k \geq i+1$,

$$\left\lceil \sum_{j=1}^{k} \frac{FF_j(p-1) - A_j(p-1)}{d+1} \right\rceil - \sum_{j=1}^{i} s_j \geq s_{i+1}$$

Now suppose the adversary plays against an arbitrary algorithm A. Let $A_i(p)$ denote the number of slots that algorithm A has of color i after phase p. We name each color according to the number of slots it has where color 1 has the most slots (i.e. $A_i(p) \geq A_{i+1}(p)$). If the relative number of slots of the colors changes, the colors are renamed accordingly. When the adversary plays against algorithm A, his goal is to keep A's distribution of slots among colors as spread out as First Fit's distribution. Specifically, after each phase p, he wants to maintain the following two properties:

- **Property 1:** $\sum_{j=1}^{i} A_j(p) \leq d + \sum_{j=1}^{i} FF_j(p)$ for any $i \leq p$
- **Property 2:** $\sum_{j\geq 1} A_j(p) = \sum_{j\geq 1} FF_j(p)$

If both Properties hold after phase p, A has used at least p colors because the total number of slots A has is more than the number of slots it has in the first $p-1$ colors after phase p. That is, since, $FF_p(p) \geq (d+1)$,

$$\begin{aligned} \sum_{i\geq 1} A_i(p) &= \sum_{i\geq 1} FF_i(p) \\ &> d + \sum_{i=1}^{p-1} FF_i(p) \geq \sum_{i=1}^{p-1} A_i(p) \end{aligned}$$

If the adversary has maintained both properties after phase $p-1$, he wants to choose the edges for the nodes in phase p such that no matter what valid coloring A uses on the nodes of phase p, both properties will be maintained. Notice that when the adversary plays against First Fit, he adds exactly $(p-1)X_p$ edges and X_p nodes to the graph in phase p. Thus the total number of slots increases by $dX_p - (p-1)X_p$. The adversary adds the same number of edges and nodes per phase when playing against any algorithm. This ensures Property 2 because the change in the total number of slots is the same for both A and First Fit. The adversary then must decide where to place these edges in order to maintain Property 1.

We define the following variables that determine the amount of "slack" in between A's distribution and First Fit's. s_i is the slack variable for color i. The slack variables are chosen in such a way that Algorithm A could add $(d+1)s_i$ more slots to color i for every $i \leq p-1$ and Property 1 would still hold after phase $p-1$.

$$= 3d\sum_{j=0}^{c-1}\left(\frac{d+1}{d}\right)^{j+1}$$

$$\leq 3d^2\left(\frac{d+1}{d}\right)^{c+1}.$$

Since $n \geq d^{2+\epsilon}$, $c = \Omega(d \log n)$.

First we describe the adversary's strategy against First Fit and then we describe how the adversary alters the strategy when playing against an arbitrary on-line graph coloring algorithm A. We then prove that A can do no better than First Fit.

If the adversary is playing against First Fit, then in the first phase, he presents X_1 independent nodes. First Fit assigns them all the color 1. In the second phase, the adversary presents X_2 nodes each of which is adjacent to a node that has been assigned color 1. First Fit will assign these nodes the color 2. In general, during phase p, every new node will be adjacent to $p-1$ nodes each of which is assigned a different color in $\{1,\ldots,p-1\}$. The adversary will only be able to do this if there are enough slots of colors $1,\ldots,p-1$. If he succeeds, then every node in phase p will be assigned the color p. Let $FF_i(p)$ denote the number of slots of color i remaining after phase p. We prove that the X_j's are chosen in such a way that that following lemma holds:

Lemma 25 *For any phase $p \leq c$, and for every $i \leq p$,*

(A) $FF_{i-1}(p) \geq 2d + FF_i(p)$ *and* $FF_p(p) = dX_p$

(B) *Every node in phase p is assigned the color p.*

Proof: By induction on p. Assuming the claim is true for every phase through phase p, we prove the claim for phase $p+1$. Since for every $i \leq p$, color i has at least dX_p slots and $dX_p \geq X_{p+1}$, there are enough slots of colors $1,\ldots,p$ so that every node in phase $p+1$ can be adjacent to nodes that have been assigned colors $1,\ldots,p$. First Fit will assign each new node the color $p+1$, which proves Part B of the lemma. In addition, the same number of slots of each color is used up, so we have for $i \leq p$, $FF_{i-1}(p+1) \geq 2d + FF_i(p+1)$. Each of the new nodes is assigned the color $p+1$ and each new node has d slots, so $FF_{p+1}(p+1) = dX_{p+1}$. The number of slots of color p after phase $p+1$ is $dX_p - X_{p+1}$, because X_{p+1} slots were used up in phase p. Since $dX_p \geq 2d + (d+1)X_{p+1}$, $FF_p(p+1) \geq 2d + FF_{p+1}(p+1)$. ■

Theorem 17 *For every on-line graph coloring algorithm A, and for every d, there is a family of d-inductive graphs $\mathcal{G}$ such that for every $n \geq d^{2+\epsilon}$, there is a $G \in \mathcal{G}$ where G has n nodes and $A(G) = \Omega(d \log n)$.*

The nodes are numbered from 1 to n in the order they are presented. The adversary presents a graph such that the number of edges from any node to higher-numbered nodes is at most d. To do this, he must ensure that after a node is presented, at most d more nodes will be adjacent to it. This ensures that G is d-inductive; an inductive order of G is the order in which the nodes are presented. We introduce the notion of *slots* which indicates for a particular node how many more edges can be attached to it without violating this condition. When a node is first presented, it has d slots. As each additional node adjacent to it appears, the number of slots decreases by one. When a node has no slots left, the adversary cannot add any more edges incident to that node. If an on-line algorithm is coloring the nodes of a graph, then the number of slots of color c is simply the total number of slots belonging to nodes that have been assigned color c.

The adversary works in phases starting with phase 1. He adds X_p nodes in the p^{th} phase and determines all the edges for these X_p nodes at the beginning of a phase. The total number of edges added during the p^{th} phase is $(p-1)X_p$. If the adversary plans to have c phases, then he chooses the X_p's as follows: $X_c = 2$.

$$X_{p-1} = \left\lceil \left(\frac{d+1}{d} \right) X_p + 2 \right\rceil$$

The adversary's goal is to force the algorithm to use p different colors by the end of the p^{th} phase. If he succeeds, then by the end of the last phase, the algorithm will have used at least c colors to color the entire graph. To bound the total number of nodes in the graph, we first bound the number of nodes in each phase. Using the fact that $X_c \leq 3$ and $X_{p-1} \leq ((d+1)/d)X_p + 3$,

$$X_p \leq 3 \sum_{i=0}^{c-p} \left(\frac{d+1}{d} \right)^i \leq 3d \left(\frac{d+1}{d} \right)^{c-p+1}.$$

The total number of nodes in the graph is the sum over all phases of the number of nodes added in each phase:

$$n \leq 3d \sum_{p=1}^{c} \left(\frac{d+1}{d} \right)^{c-p+1}$$

- Case 1: $d_i \neq 0$. Each node in S_i^{out} is colored with a different color in the range $1, \ldots c-1$. If a node j is in S_i^{out}, then $d_j \neq 0$. Thus, by the induction hypothesis, we can lower bound $\frac{P_i}{d_i}$ as follows:

$$\begin{aligned}
\frac{P_i}{d_i} &\geq \frac{1}{d_i}\left[1 + \sum_{l \in S_i^{out}} \frac{1}{d}\left(\frac{d+1}{d}\right)^{l-1}\right] \\
&\geq \frac{1}{d_i}\left[1 + \sum_{l=1}^{|S_i^{out}|} \frac{1}{d}\left(\frac{d+1}{d}\right)^{l-1}\right] \\
&= \frac{1}{d_i}\left(\frac{d+1}{d}\right)^{|S_i^{out}|} = \frac{1}{d_i}\left(\frac{d+1}{d}\right)^{c-1-|S_i^{in}|} = \frac{1}{d_i}\left(\frac{d+1}{d}\right)^{c-1-d+d_i}
\end{aligned}$$

  Since d_i is an integer between 1 and d, this expression is minimized for $d_i = d$.

$$\frac{P_i}{d_i} \geq \frac{1}{d}\left(\frac{d+1}{d}\right)^{c-1}$$

- Case 2: $d_i = 0$. Similarly to above, we get

$$P_i \geq \left(\frac{d+1}{d}\right)^{c-1-d+d_i} = \left(\frac{d+1}{d}\right)^{c-1-d}$$

■

The two claims yield the theorem because if FF colors a node with color c, then the size of the graph is at least $\frac{1}{d}\left(\frac{d+1}{d}\right)^{c-1-d}$.

$$c \leq log_{\frac{d+1}{d}} dn + d + 1 = O(d \log n)$$

4.4 Lower Bound

The following theorem shows that no on-line algorithm can perform asymptotically better than FF. Michael Saks has a simplified version of the proof of Theorem 17, but unfortunately, the simpler proof does not generalize for the case when lookahead is added to the model. We present a proof of the lower bound from which the lower bound with lookahead follows easily.

For $i = 1$: $\quad P_1^{(0)} = \ldots = P_n^{(0)} = 1$. So

$$\sum_{k=1}^{n} P_k^{(0)} = n$$

By the induction hypothesis, we know that

$$\sum_{k=i}^{n} P_k^{(i-1)} \leq n$$

Suppose $i \in S_k^{out}$, then

$$P_k^{(i)} = 1 + \sum_{l \in S_k^{out} \cap \{1,\ldots,i\}} \frac{P_l}{d_l} = \frac{P_i}{d_i} + P_k^{(i-1)}$$

If $i \notin S_k^{out}$, then $P_k^{(i)} = P_k^{(i-1)}$

For every k such that $i \in S_k^{out}$ there is an edge from k into i. There are at most d_i such edges, because there are at most d_i edges from higher-numbered nodes into i. So we have

$$\begin{aligned}
\sum_{k=i+1}^{n} P_k^{(i)} &\leq d_i \left(\frac{P_i}{d_i}\right) + \sum_{k=i+1}^{n} P_k^{(i-1)} \\
&= P_i^{(i-1)} + \sum_{k=i+1}^{n} P_k^{(i-1)} \\
&= \sum_{k=i}^{n} P_k^{(i-1)} \leq n
\end{aligned}$$

■

Lemma 24 *If i is colored with color c then*

$$\textit{(1) if } d_i \neq 0, \textit{ then } \frac{P_i}{d_i} \geq \frac{1}{d}\left(\frac{d+1}{d}\right)^{c-1}$$

$$\textit{(2) if } d_i = 0, \textit{ then } P_i \geq \left(\frac{d+1}{d}\right)^{c-1-d}$$

Proof:

By induction on c.

$|S_j^{out}| = c - 1 - (d - d_j)$. Note that there can be at most d_j edges from higher-numbered nodes directed into j. Associate with each node j a value P_j given by

$$P_j = 1 + \sum_{k \in S_j^{out}} \frac{P_k}{d_k}$$

Note that if $d_k = 0$, then there can not be any edges from higher-numbered nodes into node k. This implies that $k \notin S_j^{out}$ for every node j where $j > k$. Thus the above sum is well defined.

Intuitively, if node j gets color c, then P_j is the number of nodes used in the graph to force node j to get color c, or the "price" of node j. The constraint is that no node can have in-degree greater than d. d_j denotes the number of additional edges that can be directed into node j after it is first presented. We can think of these as "slots" for edges directed into j. So how many nodes were used to force j to get the color c? One for the node j itself. In addition, j must be adjacent to nodes assigned colors $\{1, \ldots, c-1\}$. This is the set S_j. A slot is used up from every node i in S_j^{out} (i.e. if there is an edge from j into i). Since P_i is the price of node i , and node j uses up one of its d_i slots, we add P_i/d_i to the price of node j. To formalize this, we need to prove that if the price of a node is P_i , then there are in fact at least P_i nodes in the graph. Also we need to prove that nodes that are assigned higher-numbered colors, have a higher price.

Lemma 23 *For any node j, $P_j \leq n$.*

Proof: We prove the claim by bounding partial sums defined as follows:

$$P_j^{(i)} = 1 + \sum_{k \in S_j^{out} \cap \{1, \ldots i\}} \frac{P_k}{d_k}$$

We show that for all $1 \leq i \leq n$:

$$\sum_{k=i}^{n} P_k^{(i-1)} \leq n$$

This implies the claim because $P_i^{(i-1)} = P_i$ and

$$P_i \leq P_i + \sum_{k=i+1}^{n} P_k^{(i-1)} = \sum_{k=i}^{n} P_k^{(i-1)} \leq n$$

By induction on i:

the nodes were presented in the reverse inductive order, then First Fit would use at most $d+1$ colors to color the graph. An inductive order of G defines an orientation of the edges called an *inductive orientation* obtained by orienting the edges from the higher-numbered nodes to the lower numbered nodes. Notice that in an inductive orientation, the in-degree of each node is bounded by d. In fact the upper bound of $O(d \log n)$ for First Fit holds for any graph whose edges can be oriented so that the in-degree of every node is no more than d. However, any such graph is $2d$-inductive (the average degree is no more than $2d$), so using the weaker assumption can only yields an improvement of a constant factor.

Theorem 16 *If G is a d-inductive graph on n nodes, then FF uses $O(d \log n)$ colors to color G.*

The analysis will make use of a fixed inductive orientation of the graph. Note that the order in which the adversary presents the nodes need not be an inductive order. The adversary would like to present a graph on which FF uses as many colors as possible and has as few nodes as possible. The nodes are numbered in the order they are presented. Examine the graph just after the adversary has presented node j. Let $N(j)$ denote the neighborhood of node j when it is first presented. Suppose node j gets color c. Then for every color in $\{1, \ldots, c-1\}$, there is a node in $N(j)$ that has been assigned that color. Fix a set of nodes S_j such that :

- For every $i \in S_j$, $i < j$.
- $S_j \subset N(j)$
- For every $k \in \{1, \ldots, c-1\}$ there is exactly one node in S_j that has been assigned the color k.
- $|S_j| = c - 1$

Notice that if we remove the edges between j and lower numbered nodes not in S_j, then FF will produce the same coloring. Let S_j^{in} be those nodes in S_j that are incident to edges directed into j, and S_j^{out} those nodes on S_j that are incident to edges directed out of j in the inductive orientation. We know that $|S_j^{in}| \leq d$. Let $d_j = d - |S_j^{in}|$. Then

4.2 Related Work

There has been considerable work on on-line graph coloring. In evaluating an on-line graph coloring algorithm, keep in mind that the worst possible performance ratio is n. Lovasz, Saks and Trotter have shown an algorithm that has a performance ratio of $o(n)$ [48]; their algorithm achieves a performance ratio of $O(\frac{n}{log^* n})$ on all graphs. This is close to the best any on-line algorithm can do for general graphs. Szegedy has shown that for any on-line algorithm A, and any integer k, there is a graph on at most $k(2^k - 1)$ vertices with chromatic number k on which A will use $2^k - 1$ colors [58]. This yields a lower bound of $\Omega(\frac{n}{(\log n)^2})$ for the performance ratio of any on-line algorithm on general graphs. Note that this lower bound does not say anything if we restrict our attention to the class of graphs where k is fixed and the number of vertices in the graph can be arbitrarily large. Vishwanathan has a randomized on-line algorithm for coloring graphs where the chromatic number is a constant but the size of the graph can grow [66]. His algorithm achieves a competitive ratio of $O(n/\sqrt{\log n})$ colors.

Researchers have considered on-line coloring of more restricted classes of graphs. One can show that bipartite graphs can be colored on-line using $O(\log n)$ colors. This bound is matched by a lower bound of $\Omega(\log n)$ for any on-line algorithm on trees. Kierstead and Trotter have shown an on-line algorithm that achieves an optimal performance ratio of 3 on interval graphs [41]. Kierstead has shown that FF has a constant performance ratio on the class of interval graphs [40]. Gyarfas and Lehel have shown that FF achieves a constant performance ratio on split graphs, complements of bipartite graphs, and complements of chordal graphs [24]. Although FF does well on some restricted classes of graphs, one can show that it does quite poorly in general. In fact, there is a bipartite graph on $2k$ vertices on which FF will use k colors. The graph is simply the complement of a perfect matching on $2k$ vertices.

4.3 Upper Bound

A graph G is d-inductive if the nodes of G can be ordered in such a way that each node has at most d edges to higher-numbered nodes. Such an ordering on the nodes is called an *inductive order*. An inductive order is not necessarily unique for a graph. Note that if

of lookahead was not simply a tradeoff in l , but rather produced a threshold effect. In addition, the point at which lookahead becomes an advantage is quite high.

4.1 An Application

Lovasz, Naor, Newman, and Wigderson use the upper bound in this paper in examining the relative power of determinism, randomization and non-determinism for search problems in the Boolean decision tree model [47]. In their model, the input is limited to "yes" instances of a decision problem. They examine the number of bits of the input that must be probed in order to solve the corresponding search problem. For example, given a k-chromatic graph and a coloring on its nodes that uses fewer than k colors, find two neighbors which are colored the same color. The input is an assignment of colors to nodes of a graph. The decision problem is whether there are two neighbors assigned the same color. Since the input graph is assumed to be k-chromatic and the coloring uses fewer than k colors, the input is an accepting input. The question is, how many bits of the input must be examined in order to find the two neighbors which have been assigned the same color. Lovasz *et al.*show simultaneous exponential gaps between the non-deterministic, randomized and deterministic complexity for this problem.

Let G be a m node graph (G is not part of the input). G has the property that for $r = (\log m)^2$, G is not r-colorable. Furthermore, any vertex induced subgraph with $O(m^{1/16})$ nodes has a vertex of degree at most $\log m$. (They prove that such a graph exists). The input is an r-coloring of the graph. The problem is to find two neighbors that have the same color. Lovasz *et al.*show that the non-deterministic complexity is $O(\log r)$, the randomized complexity is $O(r \log r)$ and $\Omega(\sqrt{r})$, and the deterministic complexity is $\Omega(m^{1/16})$.

To prove the lower bound on the deterministic complexity, as the algorithm probes the nodes, the adversary colors the graph using First Fit. Since any subgraph of size $O(m^{1/16})$ is $(\log m)$-inductive, after $O(m^{1/16})$ probes, the adversary will have only used $O(\log m)^2$ colors (less than r) and the algorithm will not have found and edge whose end-points have the same color.

processor allocation the cost might be delaying the scheduling of a task. In the case of register allocation, the price might be keeping extra registers to store values temporarily before they are placed in their assigned register. In either case, it is useful to know, given lookahead l, how much better an on-line algorithm can do. The only previous work on on-line algorithms with lookahead was done by Graham, Chung and Saks in the area of dynamic location problems (1-server problems with excursions) [14]. They characterize all graphs on which a dynamic location problem can be solved optimally with a fixed lookahead.

We examine on-line graph coloring for the class of inductive graphs. A d-inductive graph has the property that the vertices can be assigned distinct numbers in such a way that each vertex is adjacent to at most d higher numbered vertices. First Fit (FF) is the most natural on-line algorithm a practitioner would think of: it assigns to each vertex the lowest numbered color possible (i.e. the lowest numbered color such that the vertex is not already adjacent to any vertices of that color). We show that FF will use $O(d \log n)$ colors on any d-inductive graph with n vertices. Karloff has a proof of the upper bound for First Fit that was discovered independently [36]. We show that this bound is tight: for any on-line graph coloring algorithm A, there is a d-inductive graph on which A uses $\Omega(d \log n)$ colors. Since $d+1$ is an upper bound on the chromatic number of any d-inductive graph, this yields a bound for any d of $\Omega(\log n)$ on the performance ratio for any on-line algorithm for d-inductive graphs. The upper bound on the number of colors used yields an upper bound on the performance ratio for graphs where d and the chromatic number are closely related. For example, since planar graphs are 5-inductive and chordal graphs are $\chi(G)$-inductive, the number of colors used by FF is $O(\log n)$ for planar graphs and $O(\chi(G) \log n)$ for chordal graphs, which yields a performance ratio of $O(\log n)$ for both classes. The bound for planar graphs is tight to within a constant factor because one can show that for any on-line algorithm A, there is a tree T such that $A(T) = \Omega(\log n)$. Since trees are also chordal, the result also yields a tight lower bound on the performance ratio for any on-line algorithm for chordal graphs.

When lookahead is added to the model, our results indicate that a substantial amount of lookahead is required to give an on-line algorithm any advantage. We show that even with lookahead $\frac{n}{\log n}$, an on-line algorithm still requires $\Omega(d \log n)$ colors to color a d-inductive graph. For lookahead $l > \frac{n}{\log n}$ we can do better, because we can color a d-inductive graph on-line in $\Theta(\ min\{d \log n, \frac{dn}{l}\})$ colors. It is surprising that the advantage

Chapter 4

Coloring Inductive Graphs On-Line

In this chapter we consider a classical problem in Computer Science, Graph Coloring, put into an on-line setting. In particular, we examine the problem of coloring *inductive* graphs on-line. The work in this chapter, except for section 4.1, can be found in [29].

In an instance of on-line graph coloring, the graph is presented one vertex at a time, and when a vertex is presented, the edges from that vertex to all previously presented vertices are also given. Each vertex must be assigned a color, different from the colors of its neighbors, before the next vertex is presented. The object is to minimize the number of colors used. On-line graph coloring can be applied to processor assignment and register allocation problems [9], [53]. Another application of this result is discussed in section 4.1.

We measure an on-line algorithm in comparison to the optimal off-line algorithm. Let $A(G)$ be the number of colors used by algorithm A on the graph G. Let $\chi(G)$ be the chromatic number of G, the minimum number of colors required to color G off-line. We are interested in finding an on-line graph coloring algorithm A, that for a class of graphs $\mathcal{C}$, minimizes

$$max_{G \in \mathcal{C}} \frac{A(G)}{\chi(G)}.$$

This is the *performance ratio* of A for the class $\mathcal{C}$.

A fundamental issue of great importance is lookahead: what is it worth to know the future? It may be possible to delay a decision at some extra cost. In the case of

Open Question 1 *Show that for all G and k, $c_{LRU,k}(G) \leq c_{FIFO,k}(G)$.*

We believe that 2FAR performs far better than our results suggest:

Open Question 2 *For directed graphs, how close to $c_k(G)$ is $c_{2FAR,k}(G)$? Is there a near-optimal algorithm for all directed access graphs?*

The solution of the following questions appear to need a randomized variant of the two-person game used in proving the vine-decomposition lower bound (Theorem 3):

Open Question 3 *Is there a "universal" randomized algorithm that is close to optimal on every G? Given G, what is a lower bound on the competitiveness of a randomized algorithm against an oblivious adversary?*

Let $M = \max_{t \in T_{k+1}(G)} |\ell(t)|$. The proof of the following theorem follows from [19].

Theorem 14 *For any randomized algorithm R, $c_{R,k}(G) \geq H_{M-1}$.*

Consider the following randomized marking algorithm for tree access graphs, which is a variant of the algorithm in [19]. The algorithm proceeds in phases as with all marking algorithms. Let T be the tree formed by the set of nodes marked in the previous phase. Call a leaf L of T *dead* when the path from L to the set of nodes marked in this phase has no servers on it. On a fault, pick a live leaf L at random and evict the first server on the path from L to the marked nodes.

Theorem 15 *There is a constant b such that the above randomized algorithm achieves a competitiveness of $b \log M$ on every tree access graph G, where $M = \max_{t \in T_{k+1}(G)} |\ell(t)|$.*

Proof: Let g be the number of new nodes requested in a phase. We count faults in T during the phase (there are only g other faults). When a leaf dies, it remains dead for the rest of the phase. Thus the number of live leaves is non-increasing with time. Label each evicted node with the name of the random (live) leaf that led to its eviction. Every hole in T is labelled with the name of some leaf. After a leaf dies, no more holes are labelled with the name of that leaf.

If ℓ is the number of live leaves at the time a hole is labelled, then the hole is labelled with the name of a particular leaf with probability $1/\ell$. Suppose that just before a leaf L dies, the number of live leaves is ℓ. Then the expected number of holes labelled L is bounded above by g/ℓ: There are at most g holes and each one is labelled L with probability at most $1/\ell$.

Let L_i be the ith leaf in T that dies. The number of faults the algorithm incurs on nodes in T is at most the number of nodes in T that get labelled in the phase $\leq \sum_{i=1} Mg/(M-i+1) \leq bg \log M$. ■

3.5.3 Open problems

By Theorem 7, LRU is never more than twice as bad as FIFO, and is often better. It would be worth removing the factor of two:

algorithm is at least $\ell - 1$, and 2FAR achieves this bound. ■

Note that LRU cannot achieve this performance (because of the $k+1$-node cycle). It is possible to extend the above proof to show that on structured program graphs in which no strongly connected component has more than $k+g$ nodes, $c_{2FAR,k}(G)$ is at most $2gc_k(G)$, but we suspect that the actual performance of 2FAR is better. Irani *et al.*[31] have recently shown that a natural extension of 2FAR achieves a competitiveness within a constant multiple of $c_k(G)$ for all structured program graphs.

3.5.2 Randomized paging algorithms

We return to undirected access graphs. It has been shown [19, 50] that in the Sleator-Tarjan model, the competitiveness of the optimal randomized on-line paging algorithm is H_k (the kth harmonic number) against an oblivious adversary [4]. Can one always hope for such a dramatic improvement over deterministic algorithms? We show that when G is the $k+1$ node circle, no randomized algorithm can achieve a competitiveness better than $(\lceil \log k \rceil)/2$. Thus randomization can help by at best a constant factor in this case. We then show that a variant of the marking algorithm [19] is within a constant factor of the optimal randomized algorithm for the important case when G is a tree.

Proposition 22 *No randomized paging algorithm achieves a competitiveness less than* $\lceil \log(k+1) \rceil/2$ *on the* $k+1$*-node circle against an oblivious adversary.*

Proof: By the von Neumann minimax principle [68], it suffices to provide a probability distribution on input sequences such on which the expected number of faults for any deterministic on-line algorithm is at least $\lceil \log(k+1) \rceil/2$ during a period when an off-line algorithm makes only one fault. The sequence is generated in phases, during each of which ends when every node has been hit at least once in the phase (so that the off-line algorithm incurs one fault in each phase). At the beginning of a phase, the sequence first walks from its present position to the point diametrically across the circle, choosing one of the two ways of getting there equiprobably. ¿From there, it proceeds to the mid-point of the nodes yet to be hit in the phase, choosing equiprobably one of the two ways of getting there, and so on. Thus on each of these $\lceil \log(k+1) \rceil/2$ steps it causes the on-line algorithm to fault with probability $1/2$. ■

follows. If there is a node u not reachable from v that has a server on it, it uses the server at u. If not, let H be the maximal strongly connected component containing v, composed of cycles $C_1, \ldots C_m$ used in its inductive construction by rule (ii) above. Form an undirected graph D each of whose nodes represents a cycle C_i, with an edge between two nodes if the corresponding cycles share a node. Consider the sub-graph F of D induced by those C_i that currently contain any blue nodes (borrowing the terminology of Section 3.4). Rather than choose the server at the greatest distance from the set of currently marked nodes (as FAR would), 2FAR uses a two-stage process. Call a node in F a *peripheral node* if it is not an articulation node in F. It first selects (arbitrarily) a peripheral node C_i in F; this is the cycle from which it will bring the server. Next, let u be the node of C_i linking it to the rest of F; the server at the node that is the predecessor of u in C_i (which is in fact furthest from the marked nodes) is used.

Theorem 13 *For any structured program graph G in which every strongly connected component has size $\leq k+1$, $c_{2FAR,k}(G) = c_k(G)$.*

Proof: Since 2FAR always tries to use a server from a portion of the access graph that is unreachable from the present request, we confine our attention to the strongly connected components of G. Let us focus on such a component C containing $k+1$ nodes, k of which have servers on them (call the unoccupied node the *hole*). Note that when the hole is on one of the loops in C, every node in that loop is requested before 2FAR incurs a fault on that hole. This leads us to view each loop in C as a "super-node" that is requested all at once in the request sequence. To this end we represent C as a tree T in the following manner. There are two types of nodes in T, *loop nodes* and *cut nodes*. There is one loop node representing each loop of C. There is one cut node representing each node of C to which a loop has been attached by rule (ii) above in the synthesis of structured program graphs. An edge in T exists between a loop node and a cut node if the corresponding loop contains the corresponding node. Thus T is a bipartite tree, each of whose leaves is a loop node. The hole resides in loop of C, and we think of it as residing at the corresponding loop node of T.

2FAR always keeps the hole at a loop node that is a leaf of T, moving it to a new leaf of T on incurring a fault. Let ℓ be the number of leaves of T. It is easy to see now (along the lines of Proposition 3) that the competitiveness of any deterministic on-line

vine decomposition of G, there exists a vine decomposition $(T, \mathcal{P})$ that gives a lower bound of at least r_1. Taking another vine decomposition whose set of paths $\mathcal{P}$ is the set of blue paths at the end of the first sub-phase, there exists a vine decomposition $(T, \mathcal{P})$ that gives a lower bound of at least r_2. Since the lower bound is given by the maximum over all vine decompositions $c_k(G)$ is $\Omega(r_1 + r_2)$.

Theorem 12 *For any G and $k = n - 1$, $c_{TS,k}(G)$ is $O(c_k(G))$.*

Again, since TS is a marking algorithm, we also have $c_{TS,k}(G) \leq k$.

3.5 Extensions and further work

3.5.1 Directed graphs and structured program graphs

We now consider a class of directed graphs that captures the control flow of a program (Example 3). A *structured program graph* is meant to model the access pattern generated by an execution of a program written in a structured programming language. It is a directed acyclic graph (dag) built from the following rules:
(i) There is a unique source node s and a unique sink t, and a directed path from s to t.
(ii) A structured program graph can be derived from another by attaching to a node v in it a directed cycle $\mathcal{C}$: we identify v and one node in $\mathcal{C}$. This is to model loops in a program (DO/WHILE loops).
(iii) Series/Parallel Composition: Two program graphs can be composed in parallel by identifying their sources and sinks respectively to form a new program graph. This is to model branching (IF/CASE statements). Two program graphs can be composed in series: the sink of one is identified with the source of the other.

Notice that an arbitrary GOTO statement cannot be captured by such a graph. We now present a variant of FAR which we call 2FAR, and prove that $c_{2FAR,k}(G)$ is within a constant factor of $c_k(G)$ for any structured program graph G provided every strongly connected component of G has size at most $k + O(1)$.

2FAR is also a marking algorithm; we describe it here for the case when every strongly connected component of G has size at most $k+1$, showing that it is optimal in this case. On a fault at node v, 2FAR decides which server at an unmarked node to choose as

spanning tree of the graph. ■

We note that the bound cannot be improved by much in terms of M_G and D_G: it is tight to within a constant factor on both the complete graph and the $(k+1)$-node cycle. The recent work of Irani *et al.*[31] shows that FAR achieves essentially the best possible competitiveness on every access graph G. Their proof makes use of the vine decomposition lower bound introduced in Section 3.2, thus establishing additionally that the vine decomposition provides the best possible lower bound (to within a constant) on $c_k(G)$ for every access graph G.

For the special case $n = k+1$ we present another algorithm, which we call Two-Second (TS), that does achieve a constant factor from optimal.

We describe TS using the same terminology used to describe FAR. TS is a marking algorithm and proceeds in phases. Each phase of TS consists of two sub-phases. In the first sub-phase TS serves each request by vacating a blue node whose degree is not two, if such a node exists. (The specific node is selected arbitrarily.) The sub-phase ends when all such nodes are red. At the end of this sub-phase all remaining blue nodes are of degree two and hence can be partitioned (in a unique way) into node disjoint paths connecting nodes of degree other than two. In the second sub-phase TS serves each request by vacating the middle node of one of the current blue paths (chosen arbitrarily). The second sub-phase ends when k nodes are red. For $i = 1, 2$, let r_i be the number of faults in the ith sub-phase.

There exists a tree T such that $r_1 = O(|\ell(T)|)$. This follows from the following graph-theoretic proposition [43].

Proposition 20 *Let $n_2(G)$ be the number of nodes of degree $\neq 2$ in a graph G. Then, there exists a tree $T \subseteq G$ such that $|\ell(T)| = \Omega(n_2(G))$.*

Proposition 21 *Let $\mathcal{P}$ be the set of paths consisting of the unmarked blue nodes at the end of the first sub-phase. Then $r_2 = O(\sum_{P \in \mathcal{P}} 1 + \lfloor \log |P| \rfloor)$.*

Proof: Each request is served by the server at the mid-point of some path $P \in \mathcal{P}$. The next fault is when this mid-point is requested. However, to request this node at least half of the nodes on P have to be marked. The bound follows. ■

To prove that $r_1 + r_2$ is within a constant factor of optimal we use the lower bound given by the vine decompositions of G. Taking the tree T of Proposition 20 as a tree in a

sub-phases, showing that at most $2gM_G$ blue nodes are vacated during each sub-phase. This will yield a bound of at most $(g + 2gM_g)(1 + \lceil \log D_G \rceil)$ faults in the phase, which is at most $6c_k(G)\lceil \log 2k \rceil$ times the number of faults of the adversary in the phase.

Let $v_1, v_2, \ldots$ be the sequence of nodes vacated in the phase, and let d_i be the distance from v_i to the nearest red node at the time it is vacated. The sequence $d_1, d_2, \ldots$ is non-increasing, and $d_1 \leq D_G$. Let i_2 be the smallest index such that $d_{i_2} \leq d_1/2$; the first sub-phase ends with the vacation of v_{i_2-1} and the second sub-phase begins with v_{i_2}. In general, the jth sub-phase, for $j \geq 2$, begins at the smallest index i_j such that $d_{i_j} \leq d_{i_{j-1}}/2$. Let S_j be the set of red nodes at the beginning of sub-phase j.

Divide the sequence of nodes vacated in the jth sub-phase into *blocks* of g successive requests. Because the number of vacant blue nodes at any time is at most g, at least one node of each block is marked by the time *any* node of any subsequent block is vacated. Pick such a node for each block, and call it a *rep*. We bound the number of reps by $2M_G$, and this will imply a bound of $2gM_G$ on the number of nodes vacated in the sub-phase. Let r_j be the number of reps in sub-phase j. The number of nodes marked during the subphase is at least $r_j d_{i_j}/2$. This follows from the fact that when a rep is marked, the next rep to be marked is at a distance of at least $d_{i_j}/2$ from the set of marked nodes. Also, at the beginning of the sub-phase the first rep to be marked is at a distance of at least $d_{i_j}/2$ from the set of marked nodes. Thus, at least $d_{i_j}/2$ new nodes are marked between successive reps being marked.

We get that for any two reps u, v, the distance from u and v to any member of S_j is greater than $d_{i_j}/2$. Also, the distance between u and v is greater than $d_{i_j}/2$. (The last follows from the fact that the second of $\{u, v\}$ to be vacated must be at distance greater than $d_{i_j}/2$ from the first, which is in the set of marked nodes by then.) The shortest path then from any rep to S_j is at most d_{i_j} and does not contain any other reps.

Imagine contracting S_j to a node; we now argue that a tree can be grown from S_j whose leaves include at least half of the reps. Furthermore the size of the tree is no larger than $|S_j - S_{j+1}|$. Thus the size of the tree after S_j is expanded is no larger than S_{j+1} which is no more than than k. To build the tree, pick half of the reps. Consider the union of the shortest paths of each rep to S_j. The set of nodes contained in the union has no more than $r_j d_{i_j}/2$ nodes. Because none of the paths contain any reps, each of the reps is a leaf of a

3.4 Nearly optimal algorithms for paging

Is there an on-line paging algorithm whose competitiveness is close to $c_k(G)$ on every graph G? We seek a "universal" algorithm — informally, an algorithm whose description is independent of G (and hopefully succinct). LRU and FIFO are universal algorithms, but by Example 2 neither is close to optimal on all G. We now describe a simple and natural algorithm FAR and show that its competitiveness is close to $c_k(G)$ on every G.

FAR is a marking algorithm. Consider a phase of FAR. Call the nodes requested (and marked) in the current phase *red* and the nodes requested (and marked) in the previous phase *blue*. Notice that all blue nodes have servers on them at the beginning of the current phase.

FAR chooses the unmarked server to use on a fault by vacating a blue node whose distance to the nearest red node (in G) is maximum. The intuition behind FAR is as follows: it is known [2, 50] that the optimal (off-line) paging algorithm on any sequence is to vacate the node whose next request occurs furthest in the future. FAR attempts to approximate this by vacating a node that is far from currently red nodes in G, and thus likely to be requested far in the future.

Theorem 11 *For any G and k, $c_{FAR,k}(G) \leq 6c_k(G)\lceil \log 2k \rceil$.*

In words,$c_{FAR,k}(G)$ is within $4\lceil \log 2k \rceil$ of $c_k(G)$ on every G, a performance LRU and FIFO cannot match. By Proposition 4, we know that $c_{FAR,k}(G) \leq k$, implying that FAR is optimal in the Sleator-Tarjan model (G is the complete graph), including the constant. Recently, Irani *et al.*[31] have shown that on every undirected access graph G, $c_{FAR,k}(G)$ is within a constant multiple of $c_k(G)$, thus improving on Theorem 11 above and showing that the performance of FAR cannot be improved by more than a constant factor.

Proof of Theorem 11: Let a *new* node be a node requested in the current phase that is not blue. Let g be the number of new nodes in the current phase. By Proposition 4, the adversary incurs at least $g/2$ faults for this phase, amortized. We bound the number of faults in the phase for FAR by the number of nodes vacated by FAR in the phase.

Let $M_G = \max_{T \in \mathcal{T}_k(G)} |\ell(T)|$, and let D_G be the maximum diameter of any connected k-node induced subgraph of G. We divide the current phase into $1 + \lceil \log D_G \rceil$

Proof of Claim 2: If a lonely LRU server moves, the number of lonely LRU servers decreases by one or OPT incurs a fault. If a server becomes lonely then OPT incurs a fault. All the LRU servers that are currently lonely will service a request within the next three LRU faults. If OPT does not incur a cost then after three LRU faults the servers will be realigned. ■

Proof of Proposition 19: Observe that for $n > 5$ $c_{LRU,4}(n\text{-cycle}) \leq (n-1)/(n-4)$. We now show that $c_4(n\text{-cycle}) \geq (n-2)/(n-4)$. The Proposition follows.

Assume the on-line and off-line servers are aligned on nodes $\{1,2,3,4\}$ and that $\{5,6,\ldots,n\}$ are vacant. Suppose that the adversary requests node 5. There are two cases. CASE 1: If the on-line algorithm does not vacate node 1 (say leaving an on-line hole at $i \in \{2,3,4\}$), then the adversary requests $4,\ldots,i$ and then forces a ratio ≥ 2 by the adversary having served 5 with server at 1 and then forcing realignment at $\{2,3,4,5\}$. CASE 2: The on-line has responded to request for 5 by vacating node 1 and now has servers on $\{2,3,4,5\}$. The adversary now requests node 6. There are two subcases.

Case 2.1: The on-line algorithm responds to the request by vacating node $i \in \{2,3\}$. The adversary then requests $7,\ldots,n,1,2,3$ and forces realignment on nodes $\{n,1,2,3\}$. The total cost to the on-line algorithm is at least two (for the requests to $\{5,6\}$) $+n-4$ (for the requests to $\{7,\ldots,n,1,\ldots,i\}$) $= n-2$. The adversary pays $n-4$ by serving all faults with the server originally on node 4.

Case 2.2: The on-line algorithm responds to the request by vacating node $i \in \{4,5\}$. The adversary then requests $5,4,5,6,\ldots,n,1$ and forces realignment on $\{n-2,n-1,n,1\}$. The total cost to the on-line algorithm is at least three (for the requests to $\{5,6,5,4,5,6\} + n - 5$ (for the requests to $\{7,\ldots,n,1\} = n-2$. The adversary pays $n-4$ by serving the faults at $\{5,6\}$ with the servers originally on nodes $\{2,3\}$ and then alternating these same two servers in an LRU fashion so as to serve the faults at $\{7,\ldots,n\}$ and finish the request sequence with servers on $\{n-2,n-1,n,1\}$.

■

LRU faults. Suppose OPT does not fault before time t_3 and the servers are not aligned at time t_3. Then the lonely LRU server has not serviced a request in the interval from t_1 to t_3. This means the server on v_2 is the lonely server and the servers on v_4 and v_3 service the two LRU faults.

After the request at time t_1, the request sequence must go from v_1 to v_o; LRU services the request with the server on v_4. Then the request sequence goes to v_4 and LRU uses the server on v_3 to service the request. Furthermore, each of these nodes is hit without hitting any other nodes, and v_o is reached without hitting v_4. Thus, v_1 is adjacent to v_o and v_4 is adjacent to either v_o or v_1.

With the above adjacency information, and the fact that (v_1, v_2, v_3, v_4) and (v_2, v_3, v_4, v_o) are LRU-connected, it can be determined that either the subgraph induced by $\{v_1, v_2, v_3, v_4, v_o\}$ is biconnected or the subgraph induced by $\{v_1, v_2, v_4, v_o\}$ is biconnected. Since there are no five node biconnected subgraphs in G, we can assume the latter. The edges (v_o, v_1),(v_1, v_2),(v_2, v_3) are in the graph. Also, v_4 is adjacent to v_o or v_1 and to v_2 or v_3. v_o is adjacent to one of $\{v_2, v_3, v_4\}$.

Any node can be connected to at most one node in $\{v_1, v_2, v_4, v_o\}$. Thus, v_3 is adjacent only to v_2 (since (v_2, v_3, v_4, v_o) are LRU-connected).

After time t_3, v_2 is still occupied by the lonely LRU server which is now the least recently used. v_3 has the lonely OPT server and the current request is one of $\{v_o, v_4, v_1\}$. Without loss of generality, suppose it is v_1. In order to make LRU fault, a new node must be requested (v_3 is only reachable through v_2). Call the new node v_n. If the servers are unaligned at his point, then v_3 still contains the lonely OPT server. In order to make LRU fault, v_3 must be requested.

After time t_3, the sequence goes from v_1 to v_n to v_3. LRU faults twice and OPT faults once. We prove that if there is one more LRU fault and OPT does not fault, then the servers become aligned. The lonely OPT server occupies one of $\{v_o, v_4\}$. In order to hit the lonely OPT server, the sequence must go back through v_n and v_1. (v_n and v_3 are connecteded to only one node in $\{v_1, v_2, v_o, v_4\}$. v_n is adjacent to v_1 and v_3 is adjacent to v_2). Thus $\{v_3, v_n, v_1\}$ are occupied both by LRU and OPT servers and the lonely LRU server is the least recently used. If on the next LRU fault, OPT does not fault, then the servers become realigned. ■

vacant node v. Suppose that the on-line algorithm vacates node u to service the request. Because the graph is biconnected, there is a path from v to u that passes through at most one other vertex, w. The adversary requests w then u. On-line vacates node x to service the request to u. Because the graph is biconnected, there is a path from u to x that does not use nodes other than w and v. The adversary takes this path to node x. The on-line algorithm has incurred 3 faults, while there is still a node z that has not been requested. The off-line algorithm services the initial request to node v with the server on node z, and thus only incurs one fault. The adversary requests nodes v,w,u, and x until the servers are aligned again. ■

Proof of Proposition 18: Recall that An LRU server is *lonely* if the node it occupies is not occupied by an OPT server. Similarly, an OPT server is *lonely* if the node it occupies is not occupied by an LRU server. We prove two claims. ■

Claim 1: Start with the servers aligned. If within three LRU faults OPT has incurred only one fault and the servers are not aligned, then within three more LRU faults, either OPT incurs two faults or OPT incurs one fault and the servers become aligned.

Claim 2: From any configuration, within three LRU faults, the servers are aligned or OPT has incurred a fault.

If the servers are aligned, we start a phase when the servers become unaligned and end the phase when the servers are realigned. The two facts imply that the number of LRU faults in a phase is no more than three times the number of OPT faults in a phase.

Proof of Claim 1: A sequence of nodes (v_1, v_2, v_3, v_4) are said to be *LRU-connected* if it is possible for v_i to be occupied by the ith most recently used server for all $1 \leq i \leq 4$. In such a case v_1 is adjacent to v_2, v_3 is adjacent to either v_1 or v_2, and v_4 is adjacent to one of $\{v_1, v_2, v_3\}$.

We start with the servers aligned. Let t_1 be the time of the first LRU fault. OPT faults at time t_1 as well. After t_1, there is one lonely LRU server. Let v_o be the node occupied by the lonely OPT server. Let v_i be the node occupied by the ith most recently used server. Thus, (v_1, v_2, v_3, v_4) are LRU-connected. By examining the configuration just before t_1, (v_2, v_3, v_4, v_o) are LRU-connected.

If a lonely LRU server services a request, then either OPT incurs a fault or the number of lonely LRU servers decreases by one. Let t_2 and t_3 be the times of the next two

Proof of Proposition 16: Let A be any on-line algorithm. We consider a phase to start and end when the adversary and A have their servers aligned. Let $v_0, v_1, \cdots, v_{l-1}, v_0$ be a l-cycle and without loss of generality suppose that the phase begins with servers on nodes v_0 and v_{l-1}. The adversary requests nodes $v_1, v_2, \cdots$. CASE 1: A performs like LRU on $v_1, v_2, \cdots, v_{l-2}$ and does not separate its servers. Then the adversary requests v_{l-1} and serves $v_1, \cdots, v_{l-1}$ at the optimal cost of $l-2$ (having left a server on v_{l-1}). The adversary then continues to request v_{l-2}, v_{l-1} until A aligns its servers on v_{l-1} and v_{l-2}. Clearly A's cost is at least $l-1$. CASE 2: A separates its servers while serving the request for node v_i for some i satisfying $1 \leq i \leq l-2$. At this point, A will have servers on v_i and $v_{(i-2) \bmod l}$ and its cost on the requests $v_1, \cdots, v_i$ will be i. On the other hand, the adversary will serve the requests $v_1, \cdots, v_i$ as would LRU at the same cost i, but with servers now on v_i and v_{i-1}. The adversary then keeps requesting v_{i-1}, v_i until A aligns its servers with the adversary so that A's cost at the end of the phase is at least $i+1$. Clearly $1 + 1/i \geq 1 + 1/(l-2)$. ■

Theorem 10 *For all G, $c_{LRU,4}(G) \leq (3/2)c_4(G)$.*

Proof: To prove the theorem we use three propositions.

Proposition 17 *If G has a biconnected component on five nodes, then $c_4(G) \geq 3$.*

Proposition 18 *If G has no biconnected subgraph on five nodes, then $c_{LRU,4}(G) \leq 3$.*

Proposition 19 *If G is an n-cycle, for $n > 5$, then $c_{LRU,4}(G) \leq (5/4)c_4(G)$.*

Before proving these propositions we show how they imply Theorem 10. We consider four possible cases: CASE 1: The graph G has at least one vertex of degree at least four. Then, $c_4(G) \geq 3$ by Proposition 3. Further, for every graph G, $c_{LRU,4}(G) \leq 4$. CASE 2: The graph G has a biconnected component of size five. Then, by Proposition 17, $c_4(G) \geq 3$. CASE 3: Not cases (1) and (2), and the graph G has at least one vertex of degree three. Then, $c_4(G) \geq 2$ by Proposition 3. By Proposition 18, $c_{LRU,4}(G) \leq 3$. CASE 4: Not cases (1), (2) and (3). In this case the graph G is either a path or an n-cycle for some $n > 5$. If G is a path, then $c_{LRU,4}(G) = 1$. Otherwise, by Proposition 19, $c_{LRU,4}(G) \leq (4/3)c_4(G)$. ■

Proof of Proposition 17: The request sequence is restricted to a five node biconnected subgraph of G. We start with the servers aligned. The adversary requests the

3.3.4 Caching

LRU is used both as a paging strategy and a caching strategy. In caching, the number of blocks in the cache is quite small (i.e. $k = 2, 4$), so it is of special interest to examine LRU's behavior for these values of k. In these cases, LRU compares very favorably to the performance of any on-line algorithm.

Theorem 9 *For all G, $c_{LRU,2}(G) = c_2(G)$.*

Proof: If G is a tree, then the optimality of LRU is assured by Theorem 8. So we can now assume G has a cycle of length $l \geq 3$. The following two propositions are clearly sufficient to show the optimality of LRU when $k = 2$.

Proposition 15 *If the smallest cycle in G has length $l \geq 3$, then $c_{LRU,2}(G) \leq 1 + 1/(l-2)$.*

Proposition 16 *If G has a cycle of size $l \geq 3$, then $c_2(G) \geq 1 + 1/(l-2)$.*

■

Proof of Proposition 15: We will account for the relative costs by considering phases of requests $v_1, v_2, \cdots, v_t$ (for $v_j \neq v_{j+1}$) where LRU and the adversary are aligned before the request to v_1 (say on nodes v_{-1} and v_0) and v_t is the first request on which LRU pays but the adversary does not. Note that after serving v_t, LRU and the adversary must be aligned on nodes v_t and v_{t-1} so that a new phase begins. We need to show that in any phase, the ratio of LRU's cost to the adversary's cost is no more than $1 + 1/(l-2)$. In order to eventually avoid paying for v_t while LRU pays, the adversary must clearly separate its servers. Say the last time this separation takes place in the phase is on the request to node v_i with $1 \leq i < t$ so that the servers are not aligned again until the request to v_t. If the request sequence ever reverses itself after the request to v_i (i.e., $v_{j+1} = v_{j-1}$ for some $j \geq i$), then the adversary must pay to cover v_{j+1} while LRU does not pay so that when the phase ends, LRU and the adversary will have had the same cost in this phase. So we may assume the sequence $v_i, v_{i+1}, \cdots v_t$ does not reverse itself. Clearly the adversary has left a server on v_{i-2} while serving $v_i, v_{i+1}, \cdots, v_{t-1}$ and $v_t = v_{i-2}$. That is, $v_{i-2}, v_{i-1}, \cdots v_t$ is a (perhaps not simple) cycle of length $t - i + 2 \geq l$ (for some integer c), the adversary pays $t - i$ on $v_i, v_{i+1}, \cdots, v_{t-1}$, while LRU pays $t - i + 1$. Clearly $(t - i + 1)/(t - i) \leq 1 + 1/(l - 2)$. ■

is greater or equal to the number of tokens placed on nodes in $T(v)$ in the interval (t_0, t_1).

Consider $LRUtree(t)$ for some point in time in the interval (t_0, t_1). $LRUtree(t)$ has a leaf in $T(v)$. Whenever OPT incurs a fault in the interval (t_0, t_1), the first node without a token reached on the path from this leaf to the requested node gets a token. This node is in $T(v)$ because node v is on the path and never gets a token. Thus, the number of tokens that are placed on nodes in $T(v)$ in the interval (t_0, t_1) is greater or equal the number of OPT faults in the interval (t_0, t_1).

We now count the number of OPT faults in the interval (t_0, t_1). Examine the set of nodes that are occupied by lonely LRU servers at time t_1. These nodes have been requested since time t_1 because each lonely LRU server has serviced a request since time t_1. For each one of these nodes, an OPT server has left this node in the interval (t_0, t_1). Furthermore, none of these nodes are in $T(v)$ because no node in $T(v)$ is requested in the interval (t_0, t_1). This yields that the number of OPT faults in the interval (t_0, t_1) is greater or equal to the number of OPT servers that leave a node in $T(v)$ in the interval (t_0, t_1) plus the number of lonely LRU servers at time t_1.

Putting all the inequalities together: The number of OPT servers in $T(v)$ at time t_0 is greater or equal to the number of OPT servers that leave $T(v)$ in the interval (t_0, t_1) plus the number of lonely LRU servers at time t_1. Thus the number of OPT servers remaining in $T(v)$ at time t_1 is at least the number of lonely LRU servers at time t_1. Since the number of lonely OPT servers is the same as the number of lonely LRU servers, all the lonely OPT servers are in $T(v)$ at time t_1. ■

Corollary 13 *When G is a path, $c_{LRU,k} = 1$.*

We conclude this subsection by considering a mesh. Suppose that G is an $n \times n$ mesh, and that $n^2 = k + 1$. It is easy to see that in this case, since a mesh is biconnected $c_{LRU,k} = n^2 - 1$. On the other hand for this graph $c_k(G) \geq a(G) \geq \lfloor 2/3n^2 \rfloor + O(n)$. This is because the mesh contains a tree with $\lfloor 2/3n^2 \rfloor + cn$ leaves, for some constant c. The next proposition follows.

Proposition 14 *When G is a mesh, $c_{LRU,k}(G) \leq 3/2c_k(G) + o(1)$.*

are done. Otherwise, let r be the requested node. Assume that up until this point in time, all lonely LRU servers have tokens. OPT is about to incur a fault because the OPT server is vacating v. Since v is a leaf of $OPTtree(t)$ and a node in $LRUtree(t)$, there is some leaf l of $LRUtree(t)$ such that the path from l to r passes through v. Furthermore, v is the first non-lonely node on this path, and hence v is the first node along the path from l to r without a token. Thus v gets a token just before the request. When an LRU server services a request, it loses its token, but then it is no longer lonely. ■

We now turn to prove the lemmata.

Proof of Lemma 8: If the requested node is a leaf of $LRUtree(t)$, then it is clear that the number of tokens placed after the next fault is no more than $a(G)$. To prove the lemma, we have to prove that when the requested node is an interior node in $LRUtree(t)$, there is one leaf in the tree for which no token is placed. Since the requested node is a leaf of $OPTtree(t)$, there is a path from the requested node to a leaf of $LRUtree(t)$ that only contains nodes with lonely LRU servers. No token will be placed for this leaf because all the servers on this path have tokens. ■

Proof of Lemma 9: Suppose some LRU server with no token becomes the least recently used server at time t_1. The node where the server is located, v, is a leaf of $LRUtree(t_1)$. Let $T(v)$ be the subtree of G rooted at v that does not contain any other LRU server. For the proof we need the following proposition.

Proposition 12 *At time t_1, all of the lonely OPT servers are in $T(v)$.*

Proposition 12 implies Lemma 9 because at the next LRU fault, either OPT will incur a fault (and the server on v receives a token before it moves) or v will be requested before the fault and the server on v will not be the least recently used. ■

Proof of Proposition 12: Let $t_0 (< t_1)$ be the last time node v was requested. No nodes in $T(v)$ are requested in the interval from t_0 to t_1, denoted (t_0, t_1). On the other hand all the nodes in $LRUtree(t_1)$ are requested in this interval.

Because all the lonely LRU servers have tokens, at time t_0, the number of OPT servers in $T(v)$ is greater or equal to the number of LRU servers without tokens in $T(v)$.

No nodes in $T(v)$ are requested in the interval (t_0, t_1), so no node in $T(v)$ can be given a token twice. Thus, the number of LRU servers without a token in $T(v)$ at time t_0

this scheme, at each point of time, some of the LRU servers will have *tokens*. Whenever, an LRU server services a fault, it has to "pay" a token. Below, we describe how the tokens are distributed, and prove that (i) at most $a(G)$ tokens are placed for any OPT fault, and (ii) all the LRU faults are serviced by servers with tokens.

Suppose that OPT faults at time t, Consider the tree formed by $LRUtree(t)$. For every leaf in this tree that is not the requested node, consider the path from the leaf to the requested node. Place a token on the first node on the path without a token (if there is such a node). If an LRU server services a fault, then throw away its token. Theorem 8 follows from the following two lemmata.

Lemma 8 *At most $a(G)$ tokens are placed for any OPT fault.*

Lemma 9 *No LRU server without a token will service an LRU fault.*

■

Before proving the lemmata, we need a few facts.

Proposition 10 *If G is a tree, then OPT maintains its servers in a connected subgraph of G.*

Proof: By induction on the number of requests. Suppose that up to request t, OPT's servers are in a connected subgraph of G. Let $OPTtree(t)$ denote the tree formed by the nodes occupied by OPT servers just before time t together with the node of the tth request. If OPT faults at time t, then it evicts the server that is on the node whose next request occurs farthest in the future. This is always a leaf of $OPTtree(t)$. Thus OPT's servers will be connected after time t. ■

An LRU server is *lonely* if the node it occupies is not occupied by an OPT server. Similarly, an OPT server is *lonely* if the node it occupies is not occupied by an LRU server.

Proposition 11 *Every lonely LRU server has a token on it.*

Proof: Consider a node v with both an LRU and an OPT server such that after a request at time t, the LRU server will be lonely. If the LRU server on v already has a token, we

constant factor. Now, we need to lower bound FIFO's competitiveness:

Theorem 7 *For any G with $n \geq k+1$, $c_{FIFO,k} \geq (k+1)/2$.*

Proof: The adversary picks a subgraph H on $k+1$ nodes and only requests nodes from that subgraph. Let v and w be two adjacent nodes in H such that w is not an articulation node. At the beginning of each phase FIFO has node v vacant and the server on node w is the server that has moved the most recently. On a fault, FIFO moves the server that has moved the least recently. The adversary requests every node where FIFO has its hole until every node (except v) has been vacated once. The paths from hole to hole in the access graph are covered by FIFO's servers. Requesting these nodes does not affect FIFO's behavior. Furthermore, since w is not an articulation node, the adversary can move from hole to hole without requesting w. At this point, every one of FIFO's servers has moved once. Node w is vacant, and the server that occupies node v has moved the least recently. The adversary requests node w and FIFO moves the server on node v to node w. The phase ends: the hole is on node v, and the server on node w is the server that FIFO moved most recently. During the phase, FIFO incurs a fault on every node in the graph ($k+1$ faults). The adversary services the sequence by moving the server on node w to node v and then back to node w for the last request in the phase (2 faults). ■ Theorem 7 and the fact that LRU is k-competitive on any G [60] imply that $c_{LRU,k}(G) \leq 2c_{FIFO,k}(G)$ for all G. On many graphs, however, $c_{LRU,k}(G)$ is much smaller than $(k+1)/2$. This provides an explanation of why LRU should be preferred to FIFO in practice.

3.3.3 LRU on trees and meshes

Let G be a tree. Then, $a(G)$, the lower bound for $c_{LRU,k}(G)$, becomes $\max_{T \in \mathcal{T}_{k+1}(G)} |\ell(T)| - 1$. By proposition 3 this is also the lower bound for any on-line algorithm. For this important class of access graphs, we show that LRU achieves this lower bound, implying that LRU is optimal.

Theorem 8 *When G is a tree, $c_{LRU,k}(G) = a(G)$.*

Proof: Let $LRUtree(t)$ be the tree formed by the nodes occupied by LRU's servers and the next node requested at time t. To prove the theorem we use a charging scheme. In

In the first sub-phase, the adversary starts by requesting nodes that have servers on them in order to fix the order in which the servers were last used. Specifically, the adversary requests nodes $g+1, g+2, \ldots, k+g-\alpha(H)$. Going from node i to node $i+1$ the adversary requests only nodes whose number is greater than i. As a result of this, node $g+i$ is the ith least recently used node. Now, the adversary requests node 1, which is connected to node $k+g+\alpha(H)$, where the previous request was, and then, nodes $2, \ldots, k+g'-1$, where $k+g'$ is the non-articulation node with the highest st-number. This time, going from node i to node $i+1$ the adversary requests only nodes whose number is less than i.

We analyze the number of faults incurred by the adversary and LRU in this sub-phase. The adversary incurs g faults, by serving nodes 1 to g using the server currently on node $k+g'$. Note that since the degree of nodes 1 to $g-1$ is two, the adversary can avoid nodes 1 to $g-1$ while hitting nodes $g+1, \ldots, k+g'-1$.

Because of the ordering of the nodes, LRU would incur one fault for each node in H that corresponds to a node numbered in the range $[1, k+g'-1]$ in F that is not an articulation node, totalling to $k+g-\alpha(H)-1$.

Before starting the second sub-phase the adversary requests nodes $g, \ldots, k+g'-1$ repeatedly, so that both LRU and the adversary have their holes on nodes $\{k+g'\} \cup \{1, \ldots, g-1\}$. The second sub-phase is symmetric to the first one. The adversary requests nodes $k+g'-1$ down to g. Going from node $i+1$ to node i the adversary requests only nodes whose number is less than i (but at least g). As a result of this, node $k+g'-i$ is the ith least recently used node. Now, the adversary requests nodes $g-1, \ldots, 1, k+g+\alpha(H), \ldots, g+1$. This time, going from node $i+1$ to node i the adversary requests only nodes whose number is greater than i.

Again, the adversary incurs g faults, by serving nodes $g-1$ down to 1 and $k+g'$ using the server currently on g. Because of the ordering of the nodes, LRU would incur one fault for each node in the ranges $[1, g-1]$, and $[g+1, k+g+\alpha(H)]$ that is not an articulation node, totalling to $k+g-\alpha(H)-1$. ■

3.3.2 Comparison of LRU vs. FIFO

We use access graphs to differentiate between the the competitiveness LRU and FIFO can achieve. We have determined LRU's competitiveness on any graph up to a

properties: (i) node s is numbered 1 and node t is numbered n; and (ii) every node other than s and t are adjacent to a lower numbered node and a higher numbered node. In the sequel we refer to nodes by their st-number. The following propositions follow from the properties of st-numbering.

Proposition 6 *For each $i > 1$, there is a path between node $i-1$ and node i that only contains nodes numbered less than i.*

Proposition 7 *For each $i < n$, there is a path between node $i+1$ and node i that only contains nodes numbered greater than i.*

We return to the proof. Given H, we compute an open ear decomposition of one of its biconnected components that contains only one articulation node. (Such a biconnected component must exist.) Consider the non-trivial ear with the highest index. Without loss of generality we may assume that this ear contains at least g internal nodes. Otherwise, we may concentrate on the subgraph $H' \subset H$ given by removing these internal nodes, and get a better bound. Call this ear the "start" path of H. Note that none of the nodes on this path are articulation nodes.

Define a new graph F as follows. The set of nodes of F consists of one node for each non-articulation node of H and two nodes for each articulation node of H. Two nodes u and v in F are connected by an edge if the nodes in H corresponding to u and v are connected by an edge. Now, remove from F all edges that correspond to trivial ears with endpoints in the "start" path. Notice that F is biconnected and contains $k + g + \alpha(H)$ nodes. Moreover, since we removed all the trivial ears, the degree of all nodes in F that correspond to nodes in the "start" path (except the last) is two. Later, when no confusion may arise, we identify the nodes in H and in F.

Since F is biconnected, it has an st-numbering [45], where s is (the node corresponding to) the first internal node in the "start" path and t is its adjacent endpoint.

The adversary works in phases. At the beginning of a phase both LRU and the adversary have their holes on nodes $\{1, \ldots, g\}$, and their servers on the rest of the nodes of H. (Notice that the nodes numbered 1 to g in F each corresponds to a distinct node of H.) A phase consists of two sub-phases. In each sub-phase the adversary incurs g faults while LRU incurs at least $k + g - \alpha(H) - 1$ faults.

$c - \beta(H) + (k + g' - c)g'/g$.

We get

$$2 + \frac{k + g' - \alpha(H') - 1}{g'} \geq 2 + \frac{c - \beta(H) - 1}{g'} + \frac{k + g' - c}{g} \geq$$
$$2 + \frac{c - \beta(H) - 1 + k + g' - c}{g} \geq \frac{k + g - \beta(H)}{g}$$

■

Proof of Theorem 5: Break the sequence into phases. A phase ends when requests to k different nodes have been seen. Let g be the number of nodes requested in the current phase that were not requested in the previous phase. By Proposition 4, the cost of the optimal algorithm for the current phase is at least $g/2$, amortized. Consider the set of $k + g$ nodes that are requested in either the current or the previous phase. Let H be the subgraph induced by these $k + g$ nodes. The number of faults that LRU incurs in the phase is bounded by k. Furthermore, the number of faults that LRU incurs in the phase is also bounded by $k + g - \beta(H)$, because LRU does not vacate an articulation node whose removal results in components of size less than k. ■

Proof of Theorem 6: Let H be any subgraph of G on $k + g$ nodes. We prove that $c_{LRU,k}(G) \geq (k + g - \alpha(H) - 1)/g$. Without loss of generality, we assume that no articulation node in H separates it into components one of which consists of less than g nodes. Otherwise, let H' be a subgraph of H with $k + g'$ nodes given by removing one such component. Clearly, $(k + g' - \alpha(H') - 1)/g' \geq (k + g - \alpha(H) - 1)/g$, and thus proving for H' suffices.

In the proof we use two graph theoretic notions: *ear decomposition* and *st-numbering*. First, we review their definitions.

An ear decomposition [67] of a graph G, starting at a specified edge e is a decomposition of the edges of G into simple paths (ears) $P_0, P_1, \ldots, P_m$, where P_0 is e, and for each path P_i, $i > 0$, each of its endpoints is contained in some P_j, $j < i$, but none of its internal nodes is contained in such an ear. An ear decomposition is *open* if none of the ears is a cycle. An ear P_i, $i > 0$, is *trivial* if it consists of a single edge. A graph is biconnected if and only if it has an open ear decomposition starting from any edge.

An st-numbering of a graph G with n nodes and a specified edge (s, t) is a numbering of the nodes of G with distinct numbers in the range $[1, \ldots, n]$ with the following

The subgraph H' is defined recursively. Initially, H' is set to H. As long as H' contains an articulation node $v \notin \mathcal{B}(H)$ that disconnects H' into components one of which is of size at least k, set H' to be the subgraph induced by the nodes of this component together with v.

Suppose that H' has $k+g'$ nodes. We give a lower bound on the number of nodes that are not articulation nodes in H'. Let $\Gamma(H)$ be the set of articulation nodes in H that are (i) not in $\mathcal{B}(H)$ and (ii) share a biconnected component with a node from $\mathcal{B}(H)$. In the construction of H', the biconnected components removed from H are ones that does not contain any node in $\mathcal{B}(H)$. This implies that all nodes in $\Gamma(H)$ are in H'. Consider a node $v \in \Gamma(H)$. Since $v \notin \mathcal{B}(H)$, the removal of v from H separates H into components, one of which is of size at least k. Call the rest of the components the "small" components of v. The total size of the "small" components is at most $g-1$. For any two nodes u and v in $\Gamma(H)$, the "small" components of u and v are disjoint.

Consider a node $v \in \Gamma(H)$ in H'. There are two possibilities: either v is not an articulation node in H', or v is an articulation node.

Claim: If v is an articulation node then the subgraph of H' induced by v and the nodes from its small components (that are in H') has a biconnected component of size at least $g'+1$ that contains only one articulation node.

Proof: The subgraph of H' induced by v and the nodes from its small components must have a biconnected component with only one articulation node. If this biconnected component has less than $g'+1$ nodes then H' contains an articulation node $v \notin \mathcal{B}(H)$ that disconnects H' into components one of which is of size at least k, contradicting the definition of H'. ■

Now, we give a lower bound on the number of non-articulation nodes in H'. Let c be the total number of nodes in the biconnected components of H' that contain a node from $\mathcal{B}(H)$. Since these biconnected components are the same as the biconnected components of H, c is also the total number of nodes in biconnected components of H that contain a node from $\mathcal{B}(H)$. Let $\Gamma'(H)$ be the set of nodes in $\Gamma(H)$ that are articulation nodes in H'. The total number of nodes in the "small" components of nodes in $\Gamma'(H)$ is $k+g-c-(g-g')$. The size of the "small" components of a single node in $\Gamma'(H)$ is at most $g-1$. Thus, $|\Gamma'(H)| \geq (k+g'-c)/g$. For each node in $\Gamma'(H)$ we have at least g' nodes that are not articulation nodes. We conclude that the number of non-articulation nodes in H' is at least

Moreover, we prove that for any graph G, the two parameters differ by a constant factor.

LRU has the property that it maintains all of its servers in one connected subgraph of the access graph at all times. Given a subgraph $H \subseteq G$ with $k+g$ nodes, let v be an articulation node in H; that is, the removal of v from H separates H. Consider the components obtained by removing v. If no such component contains at least k nodes and if LRU's servers are completely contained in H, then v cannot be a hole, or otherwise LRU's servers would no longer be connected.

Let $\alpha(H)$ denote the number of articulation nodes in H. Let $\beta(H)$ denote the number of articulation nodes in H, whose removal separates H into components none of which contains k nodes.

Define

$$a(G) = \max_{k \geq g \geq 1}\{\max_{H \in \mathcal{H}_{k+g}(G)}(k+g-\alpha(H)-1)/g\}$$

$$b(G) = \max_{k \geq g \geq 1}\{\max_{H \in \mathcal{H}_{k+g}(G)}\min\{k, k+g-\beta(H)\}/g\}.$$

We prove the following three theorems.

Theorem 4 $b(G) \leq a(G) + 2$.

Theorem 5 $c_{LRU,k}(G) \leq 2b(G)$.

Theorem 6 $c_{LRU,k}(G) \geq a(G)$.

Proof of Theorem 4: Let H be a subgraph of G with $k+g$ nodes. We show that there exists a subgraph $H' \subseteq H$ with $k+g'$ nodes, for which $2 + (k+g'-\alpha(H')-1)/g' \geq \min\{k, k+g-\beta(H)\}/g$. The theorem follows.

There are two cases. Case 1: The graph H has a biconnected component H' with $k+g' > k$ nodes. In this case $\alpha(H') = 0$, and $(k+g'-1)/g' \geq k/g \geq \min\{k, k+g-\beta(H)\}/g$. Case 2: The graph H has no biconnected components with more than k nodes. If $\alpha(H) = \beta(H)$, then $(k+g-\alpha(H)-1)/g + 2 \geq (k+g-\beta(H))/g \geq \min\{k, k+g-\beta(H)\}/g$.

Suppose that $\alpha(H) > \beta(H)$. Let $\mathcal{B}(H)$ be the set of articulation nodes in H whose removal separates H into components none of which contains k nodes (i.e., $|\mathcal{B}(H)| = \beta(H)$).

in Theorem 3 to within a constant factor.

3.2.3 Useful properties of paging algorithms

Many of the algorithms studied in this paper are *marking* algorithms [19, 35]. A marking algorithm proceeds in phases. At the beginning of a phase all the nodes are unmarked. Whenever a node is requested, it is marked. On a fault, the marking algorithm vacates a server from an unmarked node (chosen by a rule specified by the algorithm), and brings it to the request. A phase ends at the first fault after every server is on a marked node (equivalently, a phase ends when k different nodes have been requested during the phase). At this point all the nodes become unmarked and a new phase begins. We use the following proposition to analyze the competitive ratio of marking algorithms.

Proposition 4 *(a) If g_i is the number of nodes requested in the ith phase that were not requested in the $i-1$th phase, then the cost of the adversary during the first i phases is at least $(\sum_{j=1}^{i} g_j)/2$. (b) If A is a marking algorithm then for any graph G, $c_{A,k}(G) \leq k$.*

The proof of part (a) is due to Fiat *et al.* [19]. The proof of part (b) is due to Karlin *et al.* [35].

The following result of Belady [2] (see also [50] and [62]) will also prove useful. Consider the off-line paging algorithm that, given an entire request sequence σ, uses the following policy: on a fault, it evicts the item in fast memory whose next access occurs furthest in the future in σ. This algorithm is called OPT, and the reason is evident from the following proposition:

Proposition 5 *For every request sequence σ, the cost of algorithm OPT is the same as the optimal cost.*

3.3 Analysis of LRU

3.3.1 LRU on general access graphs

We analyze LRU's competitiveness by defining two parameters of the graph G. One is used to lower bound $c_{k,LRU}(G)$ and the other is used to upper bound $c_{k,LRU}(G)$.

with A's and the adversary's servers aligned and all of the nodes unmarked. A node becomes marked when it is requested.

The rules according to which the adversary generates the next request are as follows. If A has a hole on a marked node, the adversary requests this hole. In case the previous request is not adjacent to this hole, the adversary requests all the nodes in a path from the previous request to the hole that consists only of marked nodes. (Such a path always exists since the marked nodes form a connected component.) Suppose that there is no hole on a marked node. If A has a hole on a leaf x of the tree, the adversary requests all the nodes in a path from the previous request to x that consists only of marked nodes and interior tree nodes. If A has a hole on a path P_i, for $i > 1$, the adversary requests all the nodes in a path to the hole that consists only of marked nodes, interior tree nodes, and at most one half of the unmarked nodes in P_i.

Suppose that A has all of its holes on unmarked nodes in P_1. As long as the number of unmarked nodes in P_1 is at least $2g$, the adversary can hit at least $g/2$ holes by requesting half of the unmarked portion of P_1 that has the most holes. In the process the adversary only requests marked nodes, interior tree nodes, and half of the unmarked nodes in P_1. If the number of unmarked nodes in P_1 is less than $2g$ (but more than g), the adversary can hit one more hole requesting all but g unmarked nodes.

The adversary continues to hit A's holes in this manner until the end of the phase, when all but g nodes have been marked. The adversary services the sequence by initially moving all its g holes to the g nodes that are left unmarked at the end of the phase. Thus, the adversary incurs no more than g faults per phase. The adversary requests all marked nodes until the servers are aligned. The phase ends, and all the nodes are declared unmarked.

Except for the g nodes left unmarked at the end of the phase, A incurs a fault for every leaf node that is not on any path, because such a leaf node is marked only when it is requested while A has a hole there. Each path P_i, for $i > 1$, is hit $v(P_i)$ times, and A incurs a fault each time. In at least $v(P_1) - \lceil \log g \rceil$ of the hits on P_1, A incurs at least $g/2$ faults. Thus, A incurs at least $(v(P_1) - \lceil \log g \rceil)g/2 + n_T + \sum_{i>1} v(P_i) - g$ faults in a phase. ■

Irani *et al.* [31] have shown that for any graph G there always exists a subgraph H which either has $k+1$ nodes or is a single cycle that achieves the maximum of the expression

T can be reached from any other leaf in T by a path that contains only interior nodes in T. Since Sleator and Tarjan prove a lower bound of $l(T) - 1$ for the paging problem with $l(T) - 1$ slots of fast memory and $l(T)$ virtual memory pages [60], we have:

Proposition 3 *For any G, k, and any on-line algorithm A, $c_{A,k}(G) \geq \max_{T \in \mathcal{T}_{k+1}(G)} |\ell(T)| - 1$.*

The adversary can obtain any lower bound The adversary only requests nodes in T. Note that this lower bound is weak on some graphs, e.g., when G is a cycle on $k+1$ nodes. (Here Proposition 3 only provides a constant lower bound, whereas by Example 2 we know that $c_k(G)$ is $\Theta(\log(k))$. We now remedy this with a more sophisticated graph-theoretic construction.

A *vine decomposition* $\mathcal{V} = (T, \mathcal{P})$ of any graph H is a tree T in H together with a set of paths $\mathcal{P} = P_1, P_2, \ldots$ such that (i) the end-points of each of the paths are nodes in T, the internal nodes are not in T; (ii) the internal nodes of the paths are disjoint; (iii) the union of the nodes in T and on the paths gives the node set of G. Let $\mathcal{V}(H)$ be the set of all vine decompositions of H.

Let P_1 be the longest path and let n_T be the number of leaves of T not on any path. The *value* of a path P, denoted $v(P)$, is defined to be $1 + \lfloor \log |P| \rfloor$. (The length of a path $|P|$ is the number of its edges.) Let $\mathcal{H}_x(G)$ denote the set of connected node-induced subgraphs of G containing x nodes.

Theorem 3

$$c_k(G) \geq \max_{1 \leq g \leq k} \max_{H \in \mathcal{H}_{k+g}(G)} \{ \max_{\mathcal{V} \in \mathcal{V}(H)} \left(\frac{v(P_1) - \lceil \log g \rceil}{2} + \frac{\sum_{i>1} v(P_i) + n_T}{g} \right) \} - 1$$

Proof: Without loss of generality we assume that G is not a tree (or else $\mathcal{P}$ is empty and the bound degenerates to that of Proposition 3. Suppose that we are given an on-line algorithm A. We show an adversarial sequence of requests that forces A to achieve a competitiveness greater or equal to the desired lower bound. The adversary concentrates on a subgraph $H \subseteq G$ achieving the maximum of the lower bound expression. We claim that in H, $|P_1| > g$. Otherwise, the subgraph $H' \subseteq H$ with $k + g'$ nodes, given by repeatedly removing nodes on paths other than P_1 until $|P_1| > g'$ would give a better bound.

Since H contains $k + g$ nodes, there are g "holes" (nodes where A does not have a server) in H at any time. We partition the request sequence into phases. A phase starts

satisfying the condition that $A(u, < v_1, ..., v_k >, s) = (< v'_1, ...v'_k >, s')$ implies $u \in v'_1, ..., v'_k$. That is, given a page request u, the next *configuration* of fast memory pages (= servers) must include u. A request sequence $\sigma = u_1, u_2, ...$ is admissible if for all i, $< u_i, u_{i+1} > \in E$ in which case we can extend the mapping to request sequences in the obvious way. Before developing the tools for analyzing specific paging algorithms, it would be helpful to know that the synthesis of optimal on-line paging algorithms is, at least theoretically, achievable. The following results are similar in spirit to decidability results of McGeoch *et al.* [51] for server problems.

Proposition 1 *In general, it is undecidable if a given paging algorithm A achieves a given competitive ratio.*

Proof: For every i, we could construct an algorithm A_i which follows a known competitive algorithm on the jth request if the ith Turing machine on input i halts in at most j steps, else it follows a known algorithm with no finite competitive ratio. ■

Proposition 2 *For any k, any finite access graph $G = (V, E)$ and any finite state algorithm (i.e., $|S|$ is bounded by a computable function of $n = |V|$), we can compute $c_{A,k}(G)$ in space polynomial in n and $\log |S|$. (All the paging algorithms we consider are indeed finite state.)*

Proof: Because S is finite, there must be a reachable configuration/state pair $< v_1, ..., v_k >$ and s, and a finite length sequence σ with $A(\sigma, < v_1, ..., v_k >, s) = (< v_1, ..., v_k >, s)$ such that the adversary can extract the ratio $c_{A,k}(G)$ by forcing A into $(< v_1, ..., v_k >, s)$ and then repeating the sequence σ. ■

3.2.2 Lower bounds on the competitiveness

Given an access graph G, observe that $c_{A,k}(G)$ and $c_k(G)$ are monotone in the addition of edges to G; thus $c_{A,k}(G)$ is bounded above by the competitiveness of A in the Sleator-Tarjan model (G is the complete graph). Also, a lower bound on $c_{A,k}(G)$ or $c_k(G)$ can be obtained by deleting edges in G. Let $\mathcal{T}_i(G)$ denote the set of trees on i nodes in G. For a tree T, let $\ell(T)$ denote the set of leaves of T. The adversary can restrict its request sequence to paths in T and obtain any lower bound for A on T that an adversary can obtain for $l(T) - 1$ servers on an $l(T)$-clique. This follows from the observation that any leaf in

bounds for LRU is as a characterization of "bad" access graphs for LRU. A refinement of our analysis shows that LRU is optimal among on-line algorithms for the important special case when G is a tree (Example 1). We also give results showing that for small k, LRU is close to optimal on every G; these results are of great interest in *caching*, where k is typically two or four.

Section 3.4 addresses the question of a universal algorithm whose competitiveness is close to $c_k(G)$ for every k and G. We show that an extremely simple and natural algorithm (FAR) achieves a competitiveness within $O(\log k)$ of $c_k(G)$ for all k and G, a performance that LRU or FIFO cannot match. We then show that a different algorithm (TS) comes within a constant factor of $c_k(G)$ when $n = k+1$. This special case ($n = k+1$) is important because it is a well-studied case [49, 54, 60] that often provides important insights into the performance of an algorithm when $n > k + 1$; indeed all known lower bounds for on-line paging and server systems are proved with $n = k + 1$. Recently, Irani *et al.*[31] have shown that our algorithm FAR comes within a constant factor of $c_k(G)$ for all k and G.

Section 3.5.1 deals with an important class of directed access graphs which we call *structured program graphs*: such graphs represent access patterns likely to arise in the execution of programs written in a structured language with loops and branches, but no GOTOs. This access model is especially interesting for studying instruction caches. We analyze a simple and natural algorithm; one conclusion we draw is that our algorithm is optimal when $n = k + 1$.

Section 3.5.2 presents some results on randomized algorithms: we show that a variant of an algorithm of Fiat *et al.* [19] is within a constant factor of optimal whenever G is a tree, and also show that there are access graphs for which randomization cannot improve the competitiveness by more than a constant factor. We conclude with a number of open problems and directions for further work.

3.2 General results for paging with access graphs

3.2.1 The complexity of determining the competitiveness

Let $G = (V, E)$ be an access graph (directed or undirected). Any on-line paging algorithm A with a set S of states managing k pages is a mapping $V \times V^k \times S \rightarrow V^k \times S$

The literature of computer performance modeling and analysis contains much related work, both theoretical and empirical. Denning [16] (and references therein) develops the *working set* model of program behavior for capturing locality of reference. Spirn [62] gives a comprehensive survey of models for program behavior, although competitive results had not been studied at that time. Shedler and Tung [59] were the first to propose a Markovian model of locality of reference: a Markov chain whose states are the pages, with transition probabilities capturing locality. Other researchers have extended this approach, using it for at least two purposes: (1) to compare the performances of different paging strategies and to tune paging parameters [25, 46], and (2) to improve program behavior by restructuring programs [18, 26], so that program blocks and data are packed into virtual memory pages so as to ensure good locality of reference. While our focus here will be on the first of these goals, our model and some of our results are likely to prove useful in studies of the latter problem.

Typical questions we study using our model are: (1) How do LRU and FIFO compare on different access graphs? (2) Given the access graph model of a program, what is the performance of LRU on that program (i.e., what is $c_{LRU,k}(G)$)? (3) Given an access graph G, what is $c_k(G)$? can a good paging algorithm be tailor-made given an access graph G? (4) Is there a "universal" algorithm whose competitiveness is close to $c_k(G)$ on every access graph? By Example 2, LRU and FIFO are not candidates. (5) What is the power of randomization in the access graph model?

3.1.3 Outline of results

In Section 3.2 we give several general results for paging with access graphs. We first show that for any finite state algorithm A with state set S, we can compute $c_{A,k}(G)$ in space polynomial in n and $\log|S|$. We then give lower bounds on $c_k(G)$, first using a spanning tree characterization and then using a more sophisticated graph decomposition we call a *vine decomposition*.

Section 3.3 is an in-depth study of the LRU (least-recently used) algorithm. The main results (Theorems 6 and 5) determine $c_{LRU,k}(G)$ for every G and k to within a factor of two. Our technique uses combinatorial properties of small subgraphs of G involving the number of articulation nodes in each subgraph. Another useful way of viewing our tight

might arise in practice, and then describe related prior work. Let $|V| = n$.

Example 1 *The access pattern of a program is often governed by the data structures it uses: for instance, the access graph for a program performing operations on a tree data structure is likely to resemble a tree, whereas for a program doing picture processing or matrix computations it is likely to resemble a mesh. Tree access patterns arise in many important applications: in data structures for key storage and retrieval, in game-tree evaluation, and in branch-and-bound algorithms. In Section 3.3 we show that LRU is optimal on every tree and that FIFO performs badly even on trees.*

Example 2 *Let G be a cycle on $n = k + 1$ nodes. It is easy to see that $c_{LRU,k}(G)$ and $c_{FIFO,k}(G)$ are both k in this case — the sequence that goes clockwise continuously causes both algorithms to fault on every request, whereas the optimal algorithm need fault only once in k requests. On the other hand, $c_k(G) = \lceil \log(k+1) \rceil$. Consider an algorithm that operates in* phases*: whenever a node is requested during a phase, the node is marked. Thus the set of unmarked nodes at any time in a phase forms a path; on a fault, the algorithm serves with the server at the mid-point of this path. If on a fault there is no server on an unmarked node, the phase is declared over and all nodes are unmarked. Clearly the adversary incurs at least one fault in each phase, and our on-line algorithm at most $\lceil \log(k+1) \rceil$. The lower bound uses the same ideas. Thus both LRU and FIFO perform poorly on this access graph (a fact observed in practice — LRU and FIFO perform poorly on loops just larger than k, leading the designers of the ATLAS computer [42] to develop a paging algorithm with a "loop detector"). Thus both $c_{LRU,k}(G)$ and $c_{FIFO,k}(G)$ are far larger than $c_k(G)$. In Section 3.4 we give on-line paging algorithms that are close to optimal on every undirected access graph (including ones with loops).*

Example 3 *The control flow of a program affects its access graph: the access graph for a program might look like a series-parallel graph with loops (possibly nested) hanging off. The graph itself is static, and is available at compile time, before the program is run. The path taken by an execution and the number of times around each loop are data-dependent; we model these by the adversary's choices. In such a case, when the control flow of the program determines the access graph ("structural locality" [25]), it is important to consider digraphs. This is the subject of Section 3.5.1.*

3.1.2 The access graph model and related previous work

The access graph for a program $G = (V, E)$ is a graph in which each node corresponds to one of the pages that a program can reference. An adversary selects the sequence of page references, subject to locality constraints imposed by the edges of G: following a request to a page (node) u, the next request must be either to u or to a node v such that (u, v) is in E. All the algorithms we study are *lazy* [49, 54]: a page is evicted only on some fault. Furthermore, we assume without loss of generality that the algorithms are unaffected by repeated hits to a page. Thus, we may assume that following a request to u, the next request is to an adjacent node. Sections 3.2–3.4 deal with the case when G is undirected, and Section 3.5.1 deals with directed access graphs.

Let $A(\sigma)$ be the number of faults made by a paging algorithm A on request sequence σ, and let $OPT(\sigma)$ be the number of faults made by an optimal algorithm (that has the entire sequence σ available to it in advance) [2]. We say that A is a *c-competitive algorithm* (on a graph G) if for all σ (where σ is a walk on G), $A(\sigma) - c \cdot OPT(\sigma)$ remains bounded by a constant. The *competitiveness* of A on G, denoted c_A, is the infimum of c such that A is c-competitive. We denote the competitiveness of on-line algorithm A with k pages of fast memory on access graph G by $c_{A,k}(G)$. We denote $\min_A c_{A,k}(G)$ (the min taken over on-line algorithms A) by $c_k(G)$, and call it the competitiveness for access graph G with k page slots. $c_k(G)$ is the best competitive ratio an on-line algorithm can hope to achieve on G.

For brevity, we refer to the k page slots of fast memory as *servers* [49]. Thus when we speak of moving a server to a page, we mean that the page is brought into that slot in fast memory. However, the access graph is not to be confused with the graphs determining the distance metrics the k-server problem [49] or in metrical task systems [7]; the "distance" traveled by a server between any two nodes of G is unity. Rather, the access graph restricts the request sequences. When G is the complete graph, our model specializes to the Sleator-Tarjan model. Our premise is that access graphs for real programs are not complete graphs, and that some of the discrepancies between the Sleator-Tarjan theory and empirical observations can be explained by this. Also, for convenience we refer to the optimal off-line algorithm and the source of requests together as *the adversary*, who generates the sequence and serves the requests off-line. We now give some examples of access graphs that

3.1.1 Motivation

Early work on evaluating paging algorithms focuses on probabilistic analysis [21], assuming that the request sequence is drawn from a probability distribution. Indeed, Franaszek and Wagner [21] compare the page fault rate of LRU and FIFO to that of the optimal algorithm, in the spirit of competitiveness, for the case where the input sequence is drawn from an arbitrary probability distribution.

Sleator and Tarjan then show that no deterministic on-line paging algorithm can achieve a competitiveness less than k, and that a number of algorithms used in practice, including *Least-Recently-Used* (LRU) and *First-In First-Out* (FIFO), are k-competitive and thus optimal by this measure. Practitioners voice at least two reservations about these results: (1) the analysis does not make a distinction between LRU and FIFO, whereas in practice LRU is almost always superior to FIFO; (2) the analysis suggests that the number of faults by LRU on a program could be k times the optimal number of faults, whereas in practice [69] the ratio is often much smaller. Further, the approach does not allow us to treat an important practical question: given some knowledge of the memory access patterns of a program, can one use it to tune the paging algorithm for better performance?

We show here that these concerns can be addressed by augmenting the Sleator-Tarjan approach with an important concept used by paging algorithms: the sequence of pages referenced by real programs exhibits *locality of reference*. It has long been observed [2, 16, 42, 59] that each time a page is referenced by a program, the next page to be referenced is very likely to come from the same small set of pages. The fact that real programs exhibit locality of reference is the feature that makes having a two-level store of memory useful in the first place.

In this paper we develop the *access graph*, a model of a program's reference patterns. Each access graph determines a class of inputs. We use the model to prove results addressing the above practical concerns, together with a number of other results. We thus provide a theoretical framework for studying the important idea of locality of reference in programs, while retaining the best features of the Sleator-Tarjan measure.

Chapter 3

Competitive Paging with Locality of Reference

3.1 Overview

The strength of the Sleator-Tarjan style of analysis is that it is more robust than probabilistic analysis, while more practical than simply analyzing the maximum cost of an algorithm over all inputs. However, competitive analysis is still a worst-case analysis, in that the adversary can choose the input which maximizes the ratio of an on-line algorithm's cost to the optimal off-line algorithm's cost. This leads to unduly pessimistic results because often, in practice, the input to a problem is not arbitrary.

We address this issue by considering the paging problem on restricted classes of inputs. The restrictions we choose reflect inputs that occur in practice. In addition, we assume that an on-line algorithm knows in advance if the input it will receive falls in a particular class, so that the algorithm can tailor its behavior for each class of inputs. In this sense, the problem is less "on-line" because some information about the input sequence is available in advance. The work in this chapter appears in "Competitive paging with locality of reference" and was done in collaboration with Allan Borodin, Prabhakar Raghavan, and Baruch Schieber [6]. Some of the results in this paper have been improved upon in a very recent paper by Irani, Karlin and Phillips [31]. This thesis only contains results from the first paper.

vertex or right-vertex will be successfully matched: if the right-vertex can not be matched by the greedy algorithm, then his corresponding left-vertex in the perfect matching must already have been matched.

Karp *et al.*consider randomized bipartite matching against an oblivious adversary [38]. They show that a very simple randomized strategy achieves an asymptotically tight bound of $n(e-1)/e + o(n)$. The algorithm orders the left-vertices according to a random permutation and then applies the greedy algorithm with the left-vertices in the random order. Khuller, Mitchell and Vazirani and independently Kalayanasundaram and Pruhs give an algorithm which achieves an optimal competitiveness of $2n-1$ for weighted bipartite matching, with the constraint that the weights obey the triangle inequality [39] [34]. Without the triangle inequality there is no on-line strategy whose competitiveness is bounded by a function of n.

class of graphs, d-inductive graphs.

2.3.2 On-line Matching

Finding a large matching in a graph is another example of a classic problem that extends naturally to an on-line setting. Consider a situation where there are a set of n jobs and n people who enter an employment center one at a time. Each person is qualified to perform some subset of the jobs. The center must assign each person a job before he/she leaves without knowing what the qualifications of future people will be. Once an assignment is made, it cannot be revoked.

The problem described above is on-line bipartite matching. In a bipartite graph, there are n left-vertices and n right-vertices. The left-vertices (jobs) are given in advance and the right-vertices (people) arrive one at a time. As each right-vertex arrives, its edges to the left-vertices are also given. The algorithm must choose one of these edges to include in the matching or must decide not to include any of them.

We restrict our attention to those graphs that have a perfect matching. It is an easy exercise to show that any deterministic on-line algorithm can be forced to match only $n/2$ edges. Karp, Vazirani and Vazirani show a lower bound of $n/2$ even for randomized algorithms against an adaptive on-line adversary using the following example [38]. For the first $n/2$ right-vertices, the adversary adds edges between the new vertex and any left-vertex that has not been matched by either the adversary or the algorithm. The adversary adds a random one of these edges to the matching. If I_n denotes the number of edges in the intersection of the adversary's and the algorithm's matchings after the first $n/2$ right-vertices have arrived, then $E[I_n] = O(\log n)$. For the next $n/2$ right-vertices, the adversary adds an edge between the new vertex and any left-vertex that the adversary has not matched. The adversary matches the new vertex arbitrarily. I_n upper bounds the number of edges the algorithm can match in the second phase. The adversary matches every right-vertex, and the algorithm matches at most $n/2 + I_n$ right-vertices.

The bound of $n/2$ is asymptotically tight. Consider the greedy algorithm that assigns each new vertex to the lowest numbered unmatched left-vertex, if one exists. The greedy algorithm will be able to match at least $n/2$ right-vertices. To prove the bound, consider the edges in a perfect matching. For every such edge, either the incident left-

2.3 Graph Growing Problems

2.3.1 On-line Graph Coloring

Many problems in Computer Science have natural on-line formulations as well as a natural notion of competitiveness. Take graph coloring as an example. A valid *coloring* of a graph is an assignment of colors to vertices such that if two nodes are adjacent, they do not receive the same color. $\chi(G)$ is the minimum number of colors used in any valid coloring of G. A graph coloring algorithm receives an input graph G and determines a valid assignment of colors to nodes. It is well known that the problem of finding a valid coloring for graph which uses the minimum number of colors is NP-hard [37].

The input to an on-line graph coloring problem is a graph with an order imposed on the vertices. The vertices are presented in order and when each vertex is presented, all the edges from the vertex to previously presented vertices are also given. The on-line graph coloring algorithm must assign a color to each vertex before the next vertex is presented. Let A be an on-line graph coloring algorithm and $A(G)$ be the number of colors that A uses to color G. The *performance ratio* of an on-line graph coloring algorithm A is the maximum over all n node graphs (or the maximum over some class of n node graphs) of $A(G)/\chi(G)$. On-line graph coloring can be applied to processor assignment and register allocation problems [9], [53].

Notice that there is nothing in the definition that limits the computational power of the on-line algorithm. Although most of the on-line algorithms that have been devised are efficient, an on-line algorithm is not restricted from computing, for example, all possible colorings of the partial graph seen thus far.

Besides graph coloring, all of the problems that have been discussed so far have had the property that the input is a (possibly infinite) sequence of 'requests'. For these problems it is important to have algorithms whose competitiveness is independent of the length of the sequence. However, many problems, like graph coloring, have bounded input size. In such cases, the best performance ratio achievable by any on-line algorithm may be a function of the size of the input.

Chapter 4 contains an in-depth study of on-line graph coloring, a discussion of previous work and applications as well as an analysis of on-line graph coloring of a particular

oryless algorithms which, as the name suggests, do not maintain any state information. Memoryless algorithms can only use information about the metric space and the current placement of the servers to determine which server to send to satisfy a particular request. Raghavan and Snir suggest a very natural, memoryless algorithm, called *harmonic* which is defined as follows. Let $d_1, \ldots, d_k$ be the distance of each server from the current request. Harmonic sends server i with probability $(1/d_i)/(\sum_{j=1}^{k} 1/d_j)$. Raghavan and Snir show that harmonic is 2-competitive and $(n-1)^2$-competitive against a non-adaptive adversary when $k = 2$ and $k = (n-1)$, respectively. They also show that harmonic is $\binom{k}{2}$-competitive against a *lazy* adversary in any metric space. A lazy adversary is restricted to requesting a point that is occupied by an off-line server but not by an on-line server if such a point exists.

Fiat, Rabani, and Ravid show an algorithm, *expand-contract*, whose competitiveness in any metric space is bounded by a function which exponential in $O(k \log k)$. This is the first algorithm whose competitiveness in any metric space was shown to be bounded by a function of k. The algorithm is defined recursively in terms of l-server algorithms for $l < k$. The base case is simply the greedy 1-server algorithm.

In proving the upper bound on the competitiveness of expand-contract, Fiat *et al.*use an interesting technique, first introduced in [19] and later developed in [20] which is essentially a *MIN* operator over on-line server algorithms. If there is a set of m on-line algorithms such that for every input at least one of them incurs a cost that is bounded by α times the optimal cost, then one can construct an on-line algorithm that is $\alpha(2em + 1)$-competitive on every input. (e is the base of the natural logarithm).

The next breakthrough for the k-server problem then came when Grove proved that the harmonic algorithm is $(\frac{5}{4}k * 2^k - 2k)$-competitive in any metric space. The result not only improves the best bound of any algorithm for general metric spaces, but does so using *harmonic* which is a very simple algorithm.

algorithms when the number of servers is $n-1$ or 2, respectively. However, implementing their algorithms requires space linear in, and time quadratic in, the minimum of the number of requests seen so far and the number of points in the metric space. It is more desirable to have an algorithm whose time and space complexity per request is a function of the number of servers. Irani and Rubinfeld and later Chrobak and Larmore show algorithms that achieve a constant competitiveness for the 2-server problem. Both algorithms have the property that they only maintain one variable and perform only a constant number of operations to decide which server to service a particular request [33] [12].

It was open for some time to find a general algorithm such that there is a function of k which bounds the algorithm's competitiveness in any metric space. As a result, researchers turned to special cases, mostly restricted metric spaces. Chrobak, Karloff, Payne and Vishwanathan show an optimal k-competitive algorithm when the metric space is the real line [10]. The algorithm is very simple: upon a request to a point x, if x is to the left or to the right of all the servers, just move the closest server. Otherwise move the server directly to the left and the server directly to the right of x towards x at the same speed. When one of the servers reaches x, then both servers stop. Chrobak and Larmore prove that the natural extension of this algorithm for trees is k-competitive as well [11].

Coppersmith, Doyle, Raghavan and Snir apply the theory of random walks and its relationship to electrical networks to the k-server problem and task systems [13]. They show randomized k-competitive algorithms against an adaptive on-line adversary for so-called *resistive spaces*. Resistive spaces include every metric space for which a k-competitive algorithm had been proven at the time the paper was written and many more metric spaces as well. Coppersmith *et al.*show how to compute a value for each pair of points in a resistive space such that on a request to a node v, if there is no server on node v then the server sitting on node v_i services the request with probability proportional to the value for (v_i, v). The algorithm is simple and memoryless. They also show how to devise algorithms for some non-resistive spaces by approximating the original metric space by a resistive metric space. The approximation technique yields a $2k$-competitive algorithm for the circle which is the first algorithm for this metric space which was shown to have a bounded competitive ratio.

It is also interesting to consider what happens when the computational resources of an on-line algorithm are restricted. Raghavan and Snir examine the tradeoff between memory and the number of random bits the algorithm can use. They also consider *mem-*

2.2 The K-server Problem

Much of the research effort that on-line algorithms have received in recent years has been directed towards the celebrated *K-server problem.* Part of the reason for the interest is that the k-server problem models problems that arise in paging, weighted caching, and planning the movement of diskheads. The popularity, however, is largely due to the appeal of the problem which is both simple to define and practical in flavor. The k-server problem was formulated by Mannasse, McGeogh, and Sleator [49] as follows:

There are k servers which are free to move around from point to point in a metric space. The input is a sequence of requests, where each request is the name of a point in the space. An algorithm must send one of the servers to each point in the sequence in order. The cost of an algorithm is the total distance traveled by all the servers. An algorithm is *on-line* if it determines which server to send to satisfy a particular request without any knowledge of the future requests. An algorithm is *off-line* if it bases its choices on the entire sequence of requests. The optimal off-line strategy can be determined using dynamic programming. The k-server conjecture states that there is a deterministic k-competitive algorithm in any metric space.

If the metric space is the unit metric space (i.e. the distance between any two points is one), then the k-server problem is simply the paging problem. Each point in the space represents a page of memory. A page resides in fast memory if the corresponding point is occupied by a server. The cost of replacing a page in memory by another (moving a server from one point to another) is one.

Manasse *et al.*show that in any metric space with at least $k+1$ points, the competitiveness of a deterministic $k-server$ algorithm is at least k. The proof uses a nice averaging technique: for every on-line algorithm A, the adversary constructs a sequence such that there are k different algorithms, the sum of whose costs on the sequence is equal to A's cost. Thus for one of the algorithms, A's cost is at least k times greater. Since the lower bound holds for any metric space with at least $k+1$ points, the proof is an alternate proof that the competitiveness of any deterministic paging algorithm is at least k. The proof can be easily extended for randomized algorithms against an adaptive on-line adversary.

If the number of servers is 2 or $n-1$, where n is the number of points in the metric space, the bound is tight. Manasse *et al.*demonstrate optimal $n-1$ and 2-competitive

input is a sequence of tasks, $\mathbf{T} = T_1, T_2, \ldots, T_m$. A schedule for a sequence of tasks is a function $\phi : \{1, \ldots m\} \rightarrow$ the set of states, where $\phi(i)$ is the state in which the i^{th} task is performed. The cost of a schedule is the sum of the transition costs plus the sum of the costs of processing the tasks:

$$\sum_{i=0}^{n-1} d(\phi(i), \phi(i+1)) + \sum_{i=0}^{n-1} T_{i+1}(\phi(i+1)))$$

An algorithm receives a sequence of tasks and must determine a schedule for the sequence. An on-line algorithm must determine in which state to perform a given task without any knowledge of what the future tasks will be (i.e. $\phi(i)$ only depends on $T_1, \ldots T_{i-1}$). Let $A(\mathbf{T})$ denote the cost of the schedule that algorithm A uses on input $\mathbf{T}$. $OPT(\mathbf{T})$ is the cost of the optimal schedule for $\mathbf{T}$. A is a *c-competitive algorithm* [35] if for all $\mathbf{T}$, $A(\mathbf{T}) - c \cdot OPT(\mathbf{T})$ remains bounded by a constant. Borodin *et al.*show an algorithm that achieves an optimal competitiveness of $2n - 1$ for any *metrical* task system. (A metrical task system has the property that $\forall i, j,\ d(i, j) = d(j, i)$). For the special case where the task system is *uniform* ($\forall\ i \neq j,\ d(i, j) = 1$), they show a randomized $2H_n$-competitive algorithm and a lower bound of H_n on the competitiveness any randomized on-line algorithm.

Since the number of states in a system is often very large, the upper bound for task systems, when applied to particular special cases, does not yield very strong results. Consider paging, for example. If there are k slots in the fast memory and m virtual memory pages, the number of states is $\binom{m}{k}$. However, there is a deterministic paging algorithm that is k-competitive. The strength of the lower bounds for task systems comes from the fact that the set of allowable tasks is not restricted in any way: any non-negative vector of length n is a valid task. Realistically, in most systems, the set of possible tasks is a subset of all non-negative n-vectors. In the paging example, if a page is requested, the corresponding task vector is 0 for the states where the requested page is in the fast memory and infinite otherwise. In the next section we consider *Server Systems*, a special case of task systems for which it is possible to achieve a competitiveness that is independent of the number of states in the system.

Chapter 2

On-line Models and Applications

In this chapter we discuss some of the areas in which on-line algorithms have been studied. The following discussion by no means covers all of the areas where problems surface that are inherently on-line. A whole spectrum of topics have yielded interesting problems for the study of on-line algorithms. Indeed, because the problem of making decisions with partial information is so fundamental, there will continue to be many areas where the theory of on-line algorithms applies.

2.1 Task Systems

In computer systems, there are often a number of ways to perform any given task. How a task is performed not only affects the cost of performing the current task, but also affects the resulting state of the system, which in turn affects the the cost of performing future tasks. Borodin, Linial and Saks formulate a general model for a system on which a series of tasks must be performed [7]. Their model includes many applications as special cases (operations on data structures, paging, processor scheduling, server systems).

A *task system* consists of a set of n states and an $n \times n$ distance matrix. The distance between two states i and j, denoted $d(i,j)$, is the cost of moving from state i to state j. We assume that the distance matrix is non-negative, has zero entries along the diagonal and obeys the triangle inequality. A task T $(= \langle T(1), T(2), \ldots, T(n) \rangle)$ is a vector of length n, where $T(i)$ is the cost of performing the task in the i^{th} state. The

analysis model to address some of its shortcomings. Chapter 4 looks at a classic problem in Computer Science in an on-line setting: On-line Graph Coloring. We analyze how well the class of d-inductive graphs can be colored on-line. The class of d-inductive graphs includes planar graphs and chordal graphs as important subclasses. We also explore the issue of lookahead: how valuable is it to know the future? We determine tight bounds on how the solution quality can be improved as a function of how much input an algorithm can see in advance.

with a sequence of requests to virtual memory pages. If the page requested is in fast memory (a *hit*), it incurs no cost; but if not (a *fault*), the algorithm must bring the requested page into fast memory at unit cost, and decide which of the k pages currently in fast memory to evict to make room for it. The goal is to minimize the number of faults. A paging algorithm is *on-line* if it makes the decision as to which page to evict without knowledge of future requests. Sleator and Tarjan give a competitive analysis of deterministic paging algorithms in their paper on List Update. They show that a commonly used paging algorithm *Least-Recently-Used* is k-competitive. Least-Recently-Used, as the name suggests, on a fault, evicts the page that was requested least recently. Sleator and Tarjan also show that k is a lower bound for the competitiveness of any deterministic on-line algorithm. Their argument is easily extended to show a lower bound for any randomized algorithm against an adaptive on-line adversary.

If the adversary can not see any of the algorithm's random choices in choosing the input sequence, a randomized on-line algorithm can do much better. It has been shown that the competitiveness of the optimal randomized on-line paging algorithm is H_k (the k^{th} harmonic number) against an oblivious adversary [19, 50]. Thus randomization yields dramatically stronger results if the adversary must choose the input before the algorithm makes any of its random choices. There is an exponential difference between the competitiveness achievable against an oblivious adversary as against an adaptive on-line adversary.

Similar work has been done on List Update, although the results are less dramatic. Since for List Update there is a deterministic 2-competitive algorithm, randomization will only help in improving this constant factor. Since the lower bound of 2 on the competitiveness for any on-line algorithm also holds for randomized algorithms playing against an adaptive on-line adversary, improvements can only be obtained against an oblivious adversary. Irani showed the first algorithm whose competitiveness is a constant less that two. Reingold, Westbrook and Sleator show a very simple 1.75-competitive randomized algorithm against an oblivious adversary. ([32], [55], [30]).

The development of competitive analysis along with amortized analysis and the results that distinguish the different types of randomized adversaries comprise the present theoretical framework for on-line algorithms. In Chapter 2 we introduce some of the areas to which this theory has been applied as well as more general models which include many applications. Chapter 3 is an in-depth study of paging in which we augment the competitive

answer σ optimally because they can determine the entire sequence before deciding how to answer it. The difference in their strengths lies in the fact that an adaptive adversary is able to devise a more costly sequence for an algorithm.

Ben-David *et al.*prove two very powerful theorems about the relative strengths of these adversaries:

Theorem 1 *If there is a randomized algorithm that is c-competitive against any adaptive off-line adversary, then there also exists a c-competitive deterministic algorithm.*

Theorem 2 *If A is c-competitive against any adaptive on-line adversary and there is a d-competitive algorithm against any oblivious adversary, then A is $(c \cdot d)$-competitive against any adaptive off-line adversary.*

In particular, the two theorems together imply that if there is a c-competitive algorithm against any adaptive on-line adversary, then there is a deterministic algorithm that suffers only a quadratic increase in the competitiveness. Unfortunately, Theorem 1 is non-constructive. For a wide class of request answer games (*Task Systems* [7]), however, Ben-David *et al.*show that if there is an c-competitive algorithm against an adaptive off-line adversary, then one can construct a $((1+\epsilon)c)$-competitive deterministic algorithm for any $\epsilon \geq 0$. They also show how to explicitly construct a deterministic $(c \cdot d)$-competitive algorithm if the conditions for Theorem 2 are met and the proof of c-competitiveness uses a computable potential function. All currently known proofs have this property. Deng and Mahajan show that Theorem 1, in general, can not be constructive [15]. They show an infinite request-answer game such that there is a 1-competitive randomized computable strategy but there is no computable c-competitive strategy for any $c > 1$. Irani and Karp show that Theorem 2 is tight for request-answer games. The example appears in [4].

1.3.1 Oblivious vs. Adaptive On-line Adversaries

The difference between an oblivious adversary and an adaptive adversary is much greater than the difference between adaptive on-line and off-line adversaries. This is best illustrated in the Paging Problem. A paging algorithm manages a two-level store consisting of a fast memory that can hold k pages, and a large slow memory. The algorithm is presented

performance of on-line algorithms. Our model, however, is not yet well defined because it must be determined whether the adversary is allowed to see the algorithm's random choices in choosing the next request in the sequence and in deciding how he will service the sequence of requests himself. The amount of information available to the adversary will determine the strength of the adversary. We measure the strength of a randomized on-line algorithm in terms of the strength of the adversary it plays against, as well as the competitiveness it achieves.

Ben-David *et al.*define three types of adversaries and explore the relationship between them [4]. In their paper, they also introduce the most general framework for on-line algorithms, *request-answer games*. Almost every on-line problem in the literature can be formulated as a request-answer game. An adversary plays against an on-line algorithm by presenting it a sequence of requests which the algorithm services. The algorithm must answer each request before the next request is presented. The adversary must also provide a set of answers to the sequence presented to the algorithm. Any set of requests and answers has a cost associated with it. We have the following three types of adversaries in increasing order of power:

- Oblivious: An oblivious adversary determines the sequence with no knowledge of the algorithm's answers.
- Adaptive On-line: An adaptive on-line adversary bases the choice of the next request on the algorithm's answers up to that point. However, he must decide how to answer the present request without knowing how the algorithm answers the present or future requests.
- Adaptive Off-line: An adaptive on-line adversary bases the choice of the next request on the algorithm's answers up to that point. He can wait to see the entire sequence before deciding how to answer any request.

Suppose an adversary plays against a randomized algorithm A and presents a sequence σ to the algorithm. Let $ADV(\sigma)$ denote the cost of the adversary's answers to the sequence σ. Let $E[A(\sigma)]$ denote the expected cost of algorithm A on σ. A is c-competitive against a particular type of adversary if for any adversary of that type, $E[A(\sigma)] - c \cdot ADV(\sigma)$ is bounded by a constant. Both the oblivious and the adaptive off-line adversary are able to

inverted if it appears in a different order in MTF's list than in OPT's list. Notice that the potential function is alway non-negative and is initially 0 if both MTF and OPT start with the same list. Sleator and Tarjan show that on each operation, the amortized cost of MTF with this potential function is no greater than twice the cost of the optimal off-line algorithm. Suppose an item i is accessed. Let o be the location of item i in OPT's list after the access. The cost of the optimal off-line algorithm is at least o. Let m be the location of item i in MTF's list before the access (i.e. MTF's cost in accessing item i). Let x be the number of items that precede i both in OPT's list after the access and MTF's list before the access. We know that $x < o$. When MTF moves item i to the front of the list, x inversions are created and $m - x - 1$ inversions are removed. MTF's amortized cost is $m-(m-x-1)+x < 2x \leq 2o$. Thus, MTF's amortized cost is less than twice the optimal's cost for the access. The analysis for paid exchanges, insertions and deletions is similar.

Sleator and Tarjan introduce *splay trees*, a very simple method of maintaining a set of linearly ordered items in a tree. They prove that the amortized cost to access an item in their scheme is $O(\log n)$ [61]. In a manner similar to Move-to-Front, an accessed item is "rotated" by a series of local operations up the path from the item to the root, so that the item is at the root of the tree. Thus, frequently accessed items are kept towards the top of the tree. Sleator and Tarjan show that splay trees are competitive against any static optimum search tree. The famous and still unresolved *Splay Tree Conjecture* postulates that the strategy is also competitive against a dynamic optimal off-line strategy. Such a result would be analogous to Sleator and Tarjan's result for List Update. A more complete definition of the Splay Tree Conjecture may be found in [61]. Tarjan has written a broader survey of applications of amortized analysis [65].

1.3 Randomization in the On-line Model

In playing against an on-line algorithm, an adversary has two tasks. First he must devise an input sequence for the algorithm to service (answer). Secondly, he must service the sequence himself. The cost the algorithm incurs on the adversary's sequence is then compared to the cost the adversary incurs in servicing the sequence. If the algorithm uses randomization, the adversary is handicapped because he does not know exactly what state the algorithm is in at any given point. Thus, we would expect randomization to improve the

of tokens for each operation. On moves which are less costly, tokens are saved which can then be used on subsequent operations. The name *potential function* stems from a physical interpretation of a the system. Just as the potential energy in a physical system refers to the energy stored in the system, the potential function measures the amount of work stored in a combinatorial system. The potential function is chosen so that if a series of cheap operations are performed, the resulting state of the system has high potential. Thus, the work that is saved is stored in the system. The work can be later released if a more costly operation must be performed.

Amortized analysis was first used to bound the cost of performing sequences of operations on data structures. There are two types of bounds on the amortized cost per operation. The first uses amortized analysis simply to bound the the average cost of an operation in a sequence. Examples of the first type include amortized analysis of balanced search trees, the union-find data structure, and self-adjusting data structures, including skew heaps and splay trees. ([52], [27], [28], [22], [1], [63]).

A stronger type of result for data structures uses amortized analysis together with competitive analysis to bound the amortized cost of an operation with respect to an optimal off-line algorithm. Typically, a potential function is defined that represents the "distance" of A's configuration to the optimal off-line algorithm's configuration. If the potential function is well chosen, then on moves where A pays much more than the optimal algorithm, its configuration moves closer to OPT's configuration after the operation is performed. The amortized cost of such a move is low because the extra cost of the operation is offset by the drop in potential.

We use List Update to illustrate the use of amortized analysis in conjunction with competitive analysis. In their competitive analysis of the MTF algorithm for List Update, Sleator and Tarjan show that for each operation, the amortized cost incurred by Move-to-Front is no more than twice the cost of the optimal off-line algorithm. Previously, Bentley and McGeogh [8] had shown that Move-to-Front is competitive against any static ordering of the list (i.e. the optimal off-line algorithm that can choose an initial ordering for the list but then can not alter the ordering).

In the Sleator-Tarjan proof of the 2-competitiveness of Move-to-Front, the potential function is chosen to be the number of *inverted pairs of items.* A pair of items is

1.2 Amortized Analysis

Amortized analysis is a useful tool that was developed about the same time as competitive analysis. The two techniques are often used in conjunction, although the use of one does not necessarily imply the use of the other. Amortized analysis is used in analyzing the running time of an algorithm that performs a sequence of operations. Data structures are a classic application: in fact, the idea was initially developed for use in analyzing data structures, although it has been useful in many other contexts. (Task systems [7], Server Systems [49]) The framework consists of a system and a set of operations. Generally, a sequence of operations is to be performed on the system, so we are mainly concerned with the amount of time required to perform the whole sequence instead of the time required to perform a single operation. In the worst case, however, a particular operation in a long sequence may be quite costly, even though the average cost per operation can be low. Using the worst-case cost per operation yields overly pessimistic results on the time to perform the whole sequence of operations. The goal of amortized analysis is to analyze the worst case, over all sequences, of the average cost per operation.

A *Potential Function*, Φ, is a map of configurations of the system to the real numbers. The potential function is chosen to represent how costly it will be to perform an operation on a system: if the potential is high, the cost of performing an operation may be high. Suppose C and C' are the configurations of the system before and after an operation is performed. Let t be the cost of performing the operation. The *amortized* cost of the operation is $t + \Phi(C) - \Phi(C')$. If the potential of the final configuration is at least as large as the potential of the initial configuration, and if we can bound the the worst-case amortized cost of each operation, then the same bound holds for the average cost per operation over any sequence of operations. Let t_i and a_i be the cost and amortized cost, respectively, of some algorithm A on the i^{th} operation. Let Φ_i denote the value of the potential function after A has performed the i^{th} operation. Then

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_n + \sum_{i-1}^{n} a_i.$$

Most often the potential function is chosen to be non-negative and initially 0, so the amortized cost of the sequence upper bounds the actual cost of the sequence. The term *amortization* borrows from banking terminology in that the algorithm is allocated some number

the maximum amount on every access. Similar adversary strategies can foil algorithms in most on-line scenarios.

There has been extensive work on List Update where the input consists of a sequence of accesses. For each access in the sequence, the item to be accessed is independently chosen according to a fixed distribution over the items [8], [5], [44], [56], [60]. There is, however, no reason to assume that there will not be some correlation between successive operations. Competitive analysis circumvents the need to estimate the distribution that generates the input because it gives a performance guarantee on every input sequence. The idea, which was first presented by Sleator and Tarjan in analyzing algorithms for List Update [60], is to evaluate an on-line algorithm in comparison to the optimal *off-line* algorithm. An algorithm is off-line if it can see the entire input in advance. The comparison is done on an input-by-input basis. Let A be an on-line algorithm, and σ an input sequence. We denote the cost of A on σ by $A(\sigma)$. In the case of List Update, $A(\sigma)$ is the cost that algorithm A incurs in performing the sequence σ of operations. Let $OPT(\sigma)$ denote the cost incurred by the optimal off-line algorithm on input σ. We say that A is a *c-competitive algorithm* [35] if for all σ, $A(\sigma) - c \cdot OPT(\sigma)$ remains bounded by a constant. For a randomized algorithm A, we replace $A(\sigma)$ by its expectation over A's random choices in the above definition. The *competitiveness* of A, denoted c_A, is the infimum of c such that A is c-competitive. Since we are comparing on-line algorithms to the optimal off-line algorithm, we are focusing on what is lost in processing the information on-line. Some sequences are inherently difficult: for example, for List Update, a difficult sequence would be one that accesses many different items in turn. An on-line algorithm is not expected to perform well on such sequences because on these, even the optimal off-line algorithm incurs a high cost.

Sleator and Tarjan introduce the *Move-to-Front* (MTF) algorithm, for List Update and prove that it is 2-competitive. MTF, as the name suggests, moves each accessed item to the front of the list after it is accessed. Intuitively, MTF mimics the behavior of the optimal off-line algorithm: both MTF and the optimal algorithm keep frequently accessed items towards the front of the list. In the analysis of MTF, the distance between MTF's and the optimal's configuration is represented by a *potential function*. The use of a potential function is a technique called *amortized analysis* and is used in the analysis of many on-line algorithms. In the next section we introduce amortized analysis and use it analyize the competitive ratio of MTF.

the choice of algorithm, it is necessary to estimate the distribution produced in practice in order to develop algorithms that perform well in practice. Unfortunately, information about the distribution may not be available. It may also be the case that the input is not generated by a distribution at all. Moreover, it is desirable to develop algorithms that are more robust, in that they will perform well under a variety of settings.

1.1 Competitive Analysis

Competitive analysis provides a way to do a meaningful worst-case analysis of on-line algorithms [60], [35]. Thus, no assumptions are made about the distribution of the input. To illustrate competitive analysis, we introduce the *List Update Problem* , which will be used as a running example throughout this chapter. The problem is to perform a sequence of operations on an unordered list of items. The input is a sequence of operations, where each operation accesses, inserts, or deletes an item. The cost of performing an access is one plus the number of items that precede the accessed item in the list. After an item is accessed, it can be moved anywhere closer to the front of the list at no extra cost. The goal is to model a linked list storing unordered data, where to access an item, one must begin at the front and search linearly through the list until the item is found. In searching for the item, it is possible to maintain a pointer to the position where the item is to be placed once found. The item, then, can be moved to the new position in constant time. A new item may be inserted anywhere in the list, and any item may be deleted. The cost of an insertion(deletion) is one plus the number of items that precede the new(old) item in the list after it is inserted (before it is deleted). An item can also be moved anywhere in the list at any time using a *paid exchange*. The cost of moving the item via a paid exchange is the distance the item is moved.

For many problems, The worst-case cost of an on-line algorithm does not provide meaningful information about the quality of the algorithm. We use List Update to illustrate the shortcomings of a worst-case analysis. A common lower bound technique is to pretend that an algorithm plays against an adversary. The adversary observes the behavior of the algorithm and accordingly constructs a bad input sequence for the algorithm. An adversary who plays against a deterministic algorithm for List Update can always choose to access the last item in the algorithm's list. Thus, any deterministic algorithm can be forced to pay

Chapter 1

Introduction

A computational problem is said to be *on-line* if it requires that irrevocable decisions be made about the output without complete knowledge of the entire input. The problem of how to make good decisions based on partial information is widespread and fundamental in Computer Science. Applications of on-line problems range from resource allocation to robot motion planning to maintaining dynamic data structures to network routing. The question is what kind of solution quality can be obtained if the output must be produced *on-line*? Although computationally efficient algorithms are preferred, the principal issues we study are information-theoretic.

Because many problems in Computer Science are inherently on-line, on-line algorithms have been designed and evaluated since the 1960's. In order to analyze on-line algorithms, some formal theoretic framework is necessary. Traditional worst-case complexity usually fails here, since, for many on-line problems, any algorithm will have an input that forces arbitrarily poor performance. In other words, an adversary who chooses an input sequence to foil an on-line algorithm can examine the present state of an algorithm and choose a difficult input based on the state.

Early work on on-line algorithms focuses on analyzing the performance of algorithms where the input is generated according to some fixed distribution [59, 21, 56, 5]. Most of this work is concerned with analyzing data structures and paging algorithms. Thus, the "quality" of an algorithm is measured by its running time for a fixed distribution, and the running time depends on the chosen distribution. Since the choice of distribution affects

Contents

generosity and support.

Mike Luby has been both friend and colleague. His thoughts and ideas have helped me a great deal. I am very indebted to Mike for his patience in reading manuscripts and listening to practice talks but even more so for his warmth and friendship.

I feel very fortunate to have had the opportunity to have worked with so many great co-authors. The work in Chapter 3 was done in collaboration with Allan Borodin, Prabhakar Raghavan, and Baruch Schieber. I especially enjoyed the month I spent at Watson Labs, working with Prabhakar and Baruch which is where much of the work on Chapter 3 began. Since then I have had many more intersting discussions on that topic with Baruch, Prabhakar, Allan Borodin, Anna Karlin and Steven Philips.

It has been very exciting to be part of the Computer Science community in Berkeley. Not only is the environment intellectually rich, but the group has provided me with some of the best friends I have ever had. I have thoroughly enjoyed the many hours spent with Diane Hernek. My only consolation in leaving her is that I am sure we will be friends for life. I am grateful to Nina "Tiny-But-Glamorous" Amenta for begin so much fun. I have also enjoyed many great times with Will Evans, Dana Randall, David Zuckerman, Michael Kearns, Dan Jurafsky, Ramon Caceres, Moni Naor, Dalit Naor, David Wolfe and Mark Gross.

I would also like to thank my family, Keki, Alice and David Irani and Carolyn Helmke. They are my foundation on which everything else rests.

Dedicated to my family
Keki, Alice, and David Irani

Both as a teacher and an advisor Dick Karp's clarity of thought and insight have been an inspiration to me. It has been a great pleasure to work with someone with such skill and who clearly enjoys problem-solving so much. Although his praise did not come cheaply, the respect that he showed me was crucial to building my confidence. I have grown a great deal under his guidance, and I feel very fortunate to have had him as an advisor.

An important turning point for me in graduate school was when I started working with Prabhakar Raghavan. Over the past couple of years I have learned so much from him, both because he is so open with his thoughts and ideas and simply from having him as a role model. Doing research with Prabhakar has been great fun, as well as instructive. He has been both a first class mentor and a great friend.

The influence of Manuel Blum has been an integral part of my education in Berkeley. His unique insight and ideas have helped shape the way I think and work. I want to thank Umesh Vazirani for his wonderful classes. He has given some of the most enjoyable lectures I have ever attended. I have also enjoyed many interesting discussions, technical and otherwise, with Raimund Seidel.

Ronitt Rubinfeld played a essential role in my survival in graduate school. She was my first real professional colleague, as well as my confidant and ally for the last five years. Even though I have known her so well for so long, I am still amazed at her unfaltering

presented, its edges to previously presented vertices are also given. Each vertex must be assigned a color, different from the colors of it neighbors, before the next vertex is given. In the sprit of competitiveness, we evaluate on-line graph coloring algorithms by their *performance ratio* which measures the number of colors the algorithm uses in comparison to the chromatic number of the graph. We consider the class of d-inductive graphs. A graph G is d-inductive if the vertices of G can be numbered so that each vertex has at most d edges to higher numbered vertices. We analyze the very simple algorithm First Fit which assigns each vertex the lowest numbered color possible, and show that if G is d-inductive then FF uses $O(d \log n)$ colors on G. We show that this bound is tight. Since planar graphs are 5-inductive, and chordal graphs are $\chi(G)$-inductive, out results yield bounds on the performance ratio of First Fit on these important classes of graphs. We also examine on-line graph coloring with lookahead. An algorithm is on-line with lookahead l, if it must color vertex i after examining only the first $l + i$ vertices. We show that for $l < \frac{n}{\log n}$ no on-line algorithm with lookahead l can perform better than First Fit on d-inductive graphs.

This research was supported by the International Computer Science Institute and by an IBM Graduate Fellowship.

Committee Chairman: Richard Karp

Competitive Algorithms for On-line Paging and Graph Coloring

Sandra Irani

Abstract

A problem is said to be *on-line* if it requires that irrevocable decisions be made about the output before having complete knowledge of the entire input. The problem of how to make good decisions based on partial information is widespread and fundamental in Computer Science. We measure the performance of an on-line algorithm with respect to the performance of the optimal off-line algorithm. The measure we use, *Competitive Analysis*, was first developed by Sleator and Tarjan [60] and gives us the ability to make strong theoretical statements about the performance of on-line algorithms without making probabilistic assumptions on the input. We analyze the competitiveness of on-line algorithms for two problems: paging and on-line graph coloring.

In the first problem, we develop a refinement of competitive analysis for paging algorithms which addresses some of the areas where traditional competitive analysis fails to represent what is observed in practice. For example, traditional competitive analysis is unable to discern between LRU and FIFO, although in practice LRU performs much better than FIFO. In addition, the theoretical competitiveness of LRU is much more pessimistic than what is observed in practice. We also address the following important question: given some knowledge of a program's reference pattern, can we use it to improve paging performance on that program?

We address these concerns by introducing an important practical element that underlies the philosophy behind paging: locality of reference. We devise a graph-theoretical model, the access graph, for studying locality of reference. We use it to prove results that address the practical concerns mentioned above. In addition, we use our model to answer the following questions: How well is LRU likely to perform on a given program? Is there a universal paging algorithm that achieves (nearly) the best possible paging performance on every program? We answer these questions without compromising the benefits of the Sleator-Tarjan model, while bringing it closer to practice.

The second problem that we consider is on-line graph coloring. In an instance of on-line graph coloring, the vertices are presented one at a time. As each vertex is