Proceedings of the 2003 Performance Metrics for Intelligent Systems (PerMIS) Workshop,
Gaithersburg, MD, August 16 - 18, 2003.

NOT(Faster Implementation ==> Better Algorithm), A Case Study

Stephen Balakirsky and Thomas Kramer

Intelligent Systems Division
National Institute of Standards and Technology
Gaithersburg, MD 20899-8230
Email: {stephen, thomas.kramer}@nist.gov

Abstract:
Given two algorithms that perform the same task, one may ask which is better. One simple answer is that the algorithm that delivers the "best" answer is the better algorithm. But what if both algorithms deliver results of similar quality? In this case, a common metric that is utilized to differentiate between the two algorithms is the time to find a solution. Measurements, however, must be performed using an implementation of an algorithm (not an abstract algorithm) and must be taken using specific test data. Because the effects of implementation quality and test data selection may be large, the measured time metric is an insufficient measure of algorithm performance and quality.

In this paper we present the specific case of several different implementations of the same Dijkstra graph search algorithm applied to graphs with various branching factors. Our experimental results show that quality rankings based on time may be heavily influenced by the choice of operational scenario and code quality. In addition, we explore possible alternative ranking schemes for the specific case of Dijkstra graph search algorithms.

Keywords: *algorithm performance, performance metric, planning systems, autonomous systems*

1 Introduction

Researchers constantly strive to develop new algorithms that can be shown to improve upon the performance of existing techniques. In many fields, optimal algorithms exist, so comparisons must be based on a metric other than result quality. When this is the case, many researchers have turned to algorithm run-time and memory usage as the primary evaluation metrics [3-6]. In fact, entire competitions have been held that have used execution time as one of the primary metrics [1].

In this paper, we will perform a case study evaluating the performance of three different implementations of the same algorithm. This study will demonstrate that computation time and memory usage are insufficient metrics for algorithm evaluation. In fact, variations in these metrics are possible even when ranking different implementations of the same algorithm. It will be shown that this variation is not due solely to code efficiency, but is due in part to basic implementation decisions that must be made by anyone who implements an algorithm on real computer hardware. In addition, computation time will be shown to be an inconsistent metric with one implementation receiving the shortest run-time in certain environments and a different implementation receiving the shortest run-time in others.

2 Case Study

As mentioned in the introduction, implementation performance depends on both algorithm quality and implementation quality. It was decided that the best way to determine the validity of run-time as an algorithm performance metric was to implement a single algorithm in multiple ways and to collect both run-time data and data supporting other metrics. If algorithm quality were the predominant factor, the run-times of the

different implementations should be roughly the same. If implementation quality is a major factor, the run-times should be significantly different. If the run-times differ, the other metrics should reveal the causes of the difference.

The algorithm that was chosen for this case study is Dijkstra's graph search algorithm [2]. This algorithm was chosen for the following reasons:

1) The algorithm is provably optimal. Since the algorithm provides optimal results, all implementations should return either the same answer or different but equally optimal answers.
2) The algorithm is relatively easy to understand and implement. This is only important in that it allows us to focus more attention on studying the metrics than on describing and implementing the algorithm.
3) The authors have access to three different implementations of the algorithm, two of which have actually been used in deployed systems. All of these were developed before this paper was envisioned, so there would be no need to be concerned with intentionally or unintentionally tailoring the implementations to prove a point.

Dijkstra's shortest path algorithm, also known as uniform cost search, is an uninformed graph search algorithm that first appeared in 1959 [2]. Graph search algorithms are used in many planning applications and strive to find the cheapest path through a graph that is composed of nodes (representing system states) connected by edges (representing system actions). The cost of a path through the graph is defined as the sum of the action costs (the edges) plus the costs of having occupied the traversed states (the nodes). The fact that no information about the number of steps in the final path, or the cost from the current state to the goal is utilized makes this an uninformed search. In fact, the uninformed search is only able to differentiate between a goal state and a non-goal state. For these reasons, uninformed searches are sometimes called "blind searches".

Dijkstra's algorithm [2] finds an optimal path as follows:

1) Create an empty set of *open* nodes. An open node is a node that the search has reached, but has not evaluated. Initialize the cost of reaching all nodes to be infinity.
2) Place the goal node into the *open* set and set its cost of being achieved to zero.
3) Find the least expensive member of the *open* set (denote this node by n_{cheap}) and remove it from the *open* set.
4) Compare the node n_{cheap} to the start node. The search proceeds from goal to start, so if the two are equal, then the search is finished!
5) Expand n_{cheap}. During this step, the cost of reaching each of n_{cheap}'s predecessors must be determined. The following steps occur for each predecessor:
    a) Determine the cost of the edge that connects n_{cheap} to the predecessor and the cost of occupying the predecessor.
    b) If the sum of these two costs plus the cost of n_{cheap} is less then the current cost of the predecessor, the edge is maintained as a forward pointing edge (set to bold in the figure), any previous forward pointing edge is removed, and the predecessor is added to the *open* set if it is not already there. Since the initial cost of each node is infinite, every node is added to the *open* set the first time it is reached.
6) Go to step (3).

It should be noted that in some implementations of the Dijkstra algorithm, the search would proceed from n_s to n_g. The above algorithm description is valid for these approaches as well with the appropriate notation change (n_g replaced with n_s and n_s replace with n_g).

This search technique will produce a path that is both complete and optimal provided that the cost of traversing an edge and occupying the node at the end is never negative. In addition, with non-negative costs, an evaluated node can never be encountered a second time more cheaply than it was reached the first time, so evaluated nodes can be dismissed whenever they are found.

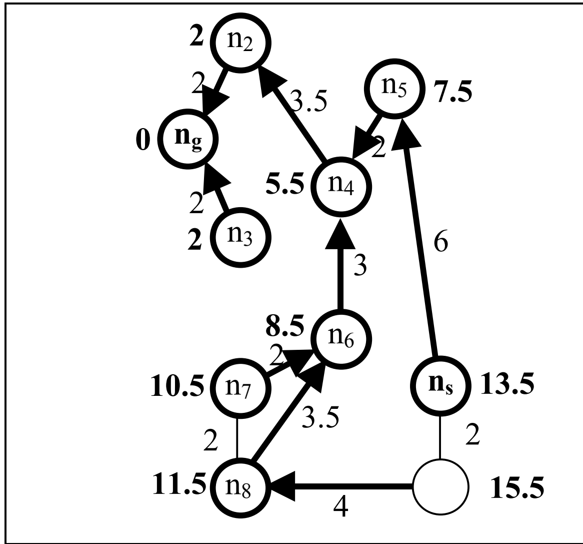

Figure 1: Example of uniform cost search graph expansion.

Figure 1 shows an example of Dijkstra's algorithm applied to a simple graph. In this figure, the circles represent nodes, with the bold circles representing nodes that have been expanded and evaluated. The bold numbers next to the nodes represent the optimal path cost from the goal to that node. The lines represent edges, and the numbers next to the lines represent the edge cost. The bold edges are edges that have been determined to be part of an optimal path, while the thin edges are not part of any optimal path. The optimal path from any expanded node to n_g lies along the decreasing cost path of bold edges in the direction of the arrows. For this example, the search proceeds from the node labeled n_g to the node labeled n_s. The search terminates at the optimal answer when the node n_s is examined for expansion. The optimal path found may be seen to be $n_s - n_5 - n_4 - n_2 - n_g$.

2.1 Implementation Variants

Timing tests and internal data collection were performed on three implementations of the Dijkstra search algorithm (DijkA, DijkB, and DijkC). While the algorithm as described above is quite simple, there are several implementation decisions that must be made when creating a running version of this algorithm. As will be shown, these implementation decisions can significantly affect the algorithm run-time performance.

One of the main requirements of the algorithm is the maintenance of the *open* set. Nodes must be added and removed from this set, and the cheapest member of this set must be available for step (3) of the algorithm. As may be seen from the above algorithm description, the functions of this set are described, but how to implement these functions is left to the discretion of the implementer. Our three implementations implement this set as a list. However, this list is implemented by employing three different techniques:

- DijkA uses a pseudo-list. This is an array made to behave like a singly linked list.
- DijkB uses a doubly linked list overlaid on an array. Each list cell contains the index in the array of its predecessor and its successor on the list.
- DijkC uses a singly linked list of nodes. Each node has a pointer to its predecessor on the list.

The other main difference in the *open* set implementation is that DijkB maintains the set as an ordered set (ordered by increasing cost), whereas DijkA and DijkC do not. This has the following effects:

- It is trivial for DijkA and DijkC to add a node to the list (just place it in the first place available), while DijkB must compare a node that is being added to the nodes already on the list in order to make the insertion at the correct point.
- It is trivial for DijkB to find the cheapest node (it is the first one on the list), while

DijkA and DijkC have to examine every node on the list to find the cheapest one.

- When the branching factor of a graph is small, DijkB will require fewer cost comparisons than DijkA and DijkC to keep the *open* list in order and find the cheapest member of the list, while when the branching factor is large, DijkB will require more cost comparisons in performing these functions.

2.2 Timing Tests Performed

Timing tests were conducted using the "time" command on a Unix system. The "time" command prints out the elapsed time from when a process starts until it finishes and prints out the percent of CPU time used by the process. Higher CPU usage gives faster times. Timing tests were repeated until at least three tests used at least 98% of the CPU time. Times were constant within 2 percent across the tests that passed that threshold. Only the test with the highest CPU usage is shown in Figure 2.

An abstract graph generator automatically generated the graphs that were searched for this project. The user of the generator specifies the number of nodes in the graph, the number of successors (also known as children) of each node, and the maximum cost of an edge. The number of successors is an environmentally dependent parameter. For example, a simple mobility planning system may use a four connected graph with four successors per node (the vehicle can move North, South, East, or West), while a more complex system may use a more highly connected graph with many successors per node. The generator writes the graph, assigning the children randomly and assigning integer-valued edge costs randomly from zero to the maximum specified by the user. For the graphs used in these tests, the maximum cost to traverse an edge and occupy the node at the end was set to 50.

As expected, the paths found during all trials by DijkA, DijkB, and DijkC had the same cost. Surprisingly, the paths found were also identical. This is surprising since it seemed likely that there would be several paths of equal minimal cost. Since the algorithm is optimal, finding *one* of these paths was expected. However, the three executables consider nodes in a different order from each other, so finding the *same one* of these paths was not expected as different paths of the same cost may have been found.

2.3 Results

Figure 2 depicts the timing results for the three variations run in two different environments. As shown in the figure, DijkB and DijkC consistently outperform DijkA. However, the relative speed of DijkB and DijkC vary with respect to the chosen environment. When the graph being searched has two successors per node, DijkB is about twice as fast as DijkC. When the graph has sixteen successors per node, DijkC is about twice as fast.

How should these results be interpreted? Is the *algorithm* instantiated in DijkB and DijkC better then the *algorithm* instantiated in DijkA? Which implementation, DijkB or DijkC is the better implementation? As sharing code becomes a more and more common practice, this timing performance trap may become more and more common. It is very easy to download someone else's source code for a particular algorithm and then run it without modification against an algorithm that is under development. However, without fully understanding the particular implementation, there is no way to determine how efficient the code is. There may also be no way of knowing if the code has been optimized for a particular environment, perhaps at the cost of lower efficiency in other environments. We therefore contend that pure timing statistics are rather meaningless when comparing different algorithms.

Unfortunately, it would seem that more complex performance metrics must be developed to truly differentiate algorithm performance from implementation performance. In order to explore this further, the three implementations were rebuilt with added internal instrumentation to capture other performance metrics.

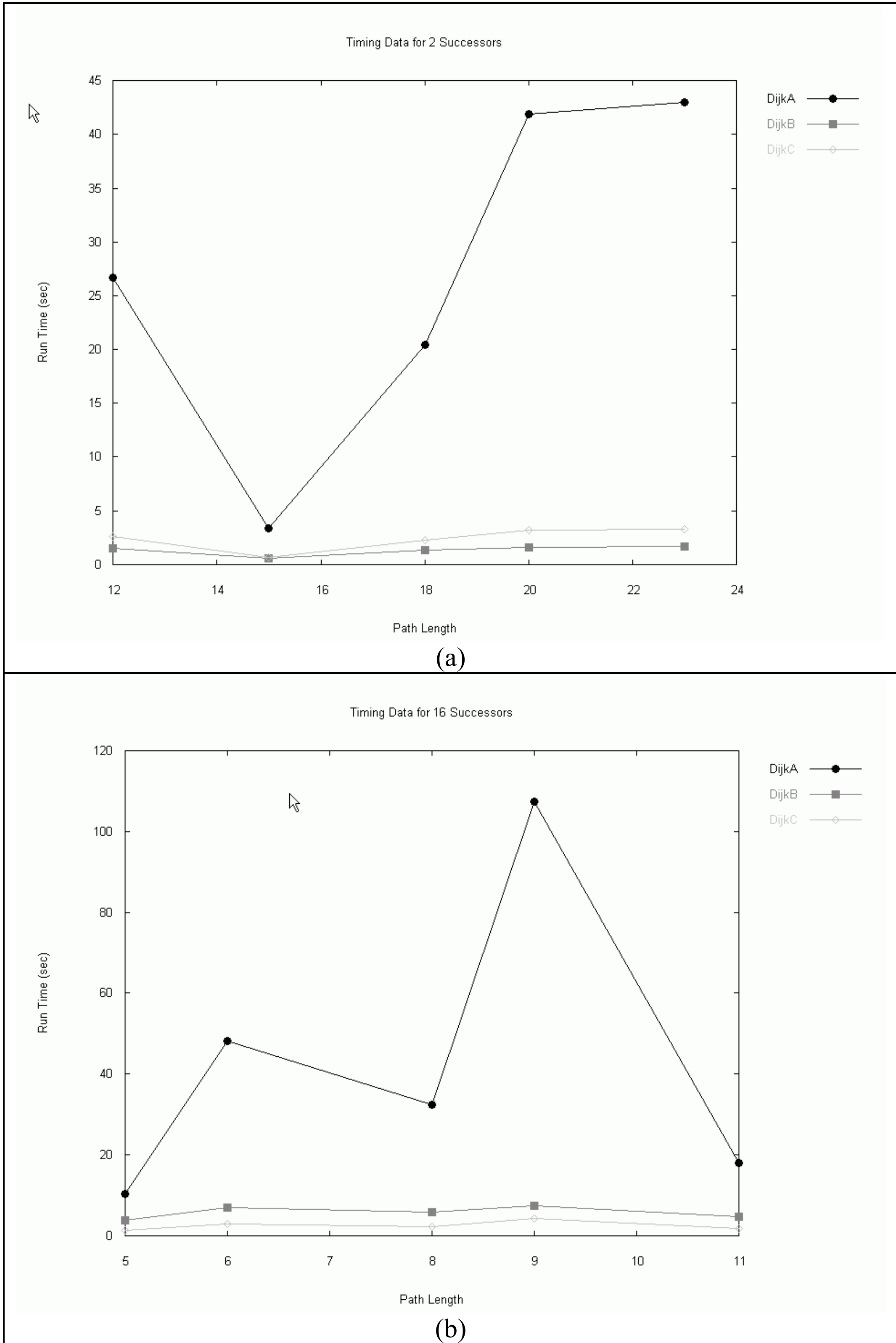

Figure 2: Time comparison of various implementations for two sample environments.

The internal instrumentation that was added captured six types of data, most importantly: (1) the number of nodes that the search opened, (2) the number of cost comparisons required either to keep the open set in order (for DijkB) or to find the cheapest of the open nodes (for DijkA and DijkC) and (3) the number of cycles (as described above) made before the answer was determined. The rationale for measuring the number of nodes opened is that at the core of any search algorithm, the search is examining various nodes to determine the "cheapest" path from the start to the goal. A perfect algorithm would only consider the nodes and edges that actually make up the correct path. Therefore, it would seem that a viable measure of algorithm performance would be the number of extra nodes and edges examined. Since each graph used in our tests had a fixed number of edges leading away from each node, the number of edges examined was proportional to the number of nodes examined.

As expected, DijkA and DijkC opened the same number of nodes; in fact exactly the same nodes, but not always in the same order. DijkB opened either the same number of nodes as DijkA and DijkC or slightly more (a fraction of a percent more in all but one case, where it opened ten percent more). This shows that the *algorithm* performance is nearly identical for all three variants. Also as expected, DijkB displayed the lowest set overhead for small branching factors, while DijkA and DijkC tied for lowest set overhead for large branching factors.

The large time difference experienced by DijkA is explained by a factor that was not measured in these experiments but is known from examining the code to be inefficient list handling. This could be measured readily.

3 Summary and future work

The case study presented in this paper shows that algorithm run-time is not an adequate "catch-all" performance metric. Indeed, it is possible for different implementations of the same algorithm to display vastly different environment dependent run-time performance and vastly different implementation quality dependent performance. This shows the need to fully understand the algorithms being evaluated and to meaningfully instrument the algorithms in order to gain true performance information.

In the future, additional search algorithms will be instrumented in a similar manner and added to this evaluation. It is hoped that this will display the number of nodes and edges evaluated as a truly relevant performance metric for graph search algorithms.

References

1. Bacchus, F., "The AIPS '00 Planning Competition," *AI Magazine*, Vol. 22, No. 3, 2001, pp. 47-56.

2. Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numerische Mathematik*, Vol. 1, 1959, pp. 269-271.

3. Manzini, G., "BIDA*: An Improved Perimeter Search Algorithm," *Artificial Intelligence*, Vol. 75, No. 2, 1995, pp. 347-360.

4. Sen, A. K., "Searching graphs with A*: applications to job sequencing," *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, Vol. 26, No. 1, 1996, pp. 168-173.

5. Shekhar, S., "Path computation algorithms for advanced traveller information system (ATIS)," 1993, pp. 31-39.

6. Stentz, A., "Optimal and Efficient Path Planning For Unknown and Dynamic Environments," *International Journal of Robotics and Automation*, Vol. 10, No. 3, 1995, pp. 89-100.