A Hybrid Technique for Deterministic Algorithms with a Shortest Path Example

by

Miroslaw Malek

and

Ahmed Azfar Moin

Department of Electrical and Computer Engineering

The University of Texas at Austin

April 25, 1994

ABSTRACT

A general hybridization technique, potentially effective in improving algorithmic efficiency, is presented for deterministic algorithms. Extensions of existing hybrid techniques for nondeterministic algorithms to deterministic ones are also described. The new technique uses one or more input parameters to predict the behavior of a group of algorithms and then takes advantage of the fastest method for that instance. We have applied this technique to Dijkstra's and Floyd's algorithms for the shortest path problem to create a hybrid dependent on the size and density of the input graph. An analysis based on complexity of the algorithms confirms our empirical results. We have also presented a performance gain analysis that quantifies the significant improvement due the new algorithm. The hybrid algorithm was able to provide 16% and 26% improvement over each parent algorithm respectively, for large graphs and a uniform distribution of density.

When multiple algorithms exist for a certain problem, an over all improvement in algorithmic performance may be possible by creating a mixture of the existing methods. A combination of algorithms that exploits the relative strengths of each method and undermines its weaknesses should result in a more robust technique that provides faster or better solutions. We call such an amalgam a hybrid algorithm and the component algorithms its parents.

Until now, this hybrid technique has been successfully applied to combinatorial optimization problems [Malek et al., 1989a, Malek et al., 1989b, Kido, Kitano and Nakanishi, 1993, Hogg and Williams, 1993] that mostly fall into the intractable NP-complete or NP-hard class. The available heuristic methods for such problems generally promise a statistically feasible approximation to the optimal solution but can exhibit an unacceptable worst case behavior. Hybrid methods can filter out the decrepit aspects of individual methods by integrating heuristics that exhibit varying behavior, to achieve higher stability and finer quality for a solution in general. In this paper we present a similar idea to improve algorithmic performance of deterministic methods by harnessing two or more algorithms to the same task. We conjecture that a deterministic hybrid algorithm would be more efficient than its parents while always guaranteeing the optimal solution. As an example, we have created a hybrid for the shortest path problem, from existing algorithms by Dijkstra [Dijkstra, 1959] and Floyd [Floyd, 1962], that provides significant improvement over each parent algorithm for a uniform distribution of input.

The next section discusses current approaches towards generating a hybrid for both non-deterministic and deterministic types of problems and briefly mentions a few existing specimens. Then we describe our experiment and present results that prove the existence of a deterministic hybrid for the shortest path problem. The succeeding analysis, based on the complexities of the parent algorithms, confirms our empirical findings. Before concluding we provide an analysis quantifying the performance gains due to the improved algorithm.

Hybrid Algorithms

A hybrid algorithm technique is a method of combining two or more algorithms for improved performance and/or reliability. The integration of algorithms may be performed on a single processor or a parallel computer. In a parallel system multiple algorithms are executed simultaneously and stopped when one of them finds the solution first. This technique can be used for deterministic and nondeterministic problems but entails a hardware cost that may prove debilitating from an applications point of view, though it is a good way of determining the ability of a group of algorithms to form a hybrid.

We can use parallelism with greater efficacy with slightly more subtle techniques. Such techniques usually involve some sort of inter-processor communication. An example is the tabu search (TABU) and simulated annealing (SA) hybrid created for the traveling salesman problem (TSP) [Malek et al., 1989a, Malek et al. 1989b]. The two heuristics, TABU and SA, are run in parallel and stopped after a predetermined interval. Then the partial solutions of the two algorithms are exchanged and the algorithms are started again. This proves to be a useful technique in mixing the two heuristics and allowing them to take advantage of each others potential. Searching through the solution space, SA finds local optima quickly but tends to get stuck in one, while TABU is relatively better at escaping from a local optimum, thus a combination of the two holds a greater potential for finding the globally optimum solution. Another experiment involving two SA algorithms with different cooling schedules demonstrated up to eleven times speedup of a two processor implementation with respect to a single processor executing one of the two algorithms. An even more robust hybrid was claimed by combining genetic algorithms GA with TABU and SA [Kido, Kitano and Nakanishi, 1993]. In this hybrid GA was used as a global search to mix the results from TABU and SA, working as local searches. The two hybrids improved the stability and quality of the solution for the TSP.

Another example of a hybrid technique that uses inter-processor communication exists for the Graph Coloring Problem consisting of the Brelaz heuristic and heuristic repair [Hogg and Williams, 1993]. This hybrid method also uses the idea of different algorithms sharing information while running in parallel. Here a common 'black board' is used to allow the processors, executing different instances of the heuristics, to exchange hints. Each processor shares information by posting hints such as its present state or a partially found solution, then a processor reads a hint with some probability and tries to incorporate it into its current state. Experimentally, a group of cooperating algorithms were shown to generally outperform a group running independently. It is pertinent to note here that determining the nature of information to exchange is a non-trivial task, and even more difficult is relating, apriori, the impact of a particular hint to the performance of the hybrid.

The approach towards generating a hybrid from deterministic algorithms becomes slightly different than creating one from heuristics. This results from the predictability of most deterministic algorithms that limits our ability to coax these algorithms to improve their performance by changing initial conditions or location in the solution space. But this predictability can be brought to use in creating a hybrid taking a somewhat different approach. The key strategy for these algorithms is to determine their behavior as a function of the input. Depending on the type or application of an algorithm, its behavior is related to some property of the input, a common example is the size n, usually seen in the computational complexity of the algorithm as $O(n)$, $O(n^2)$, etc. The complexity function can depend on parameters other than the size but nevertheless it is a useful indicator of the ability of a group of algorithms to form a hybrid as we will show in our shortest path hybrid example. Using the computational complexity as a stepping stone we can empirically determine the exact behavior of the candidate parent algorithms. Then simply by scanning the input we can use our results to predict the winning algorithm and thus obtain a hybrid consisting of the better-performing parts of each parent.

Some deterministic algorithms may defy the approach based on predictability because of difficulty in determining an easily quantifiable input parameter. For example, in a sorting problem one algorithm might perform better if the array to be sorted is already ordered or some of its segments are ordered while another method's behavior may be independent of the ordering of the array. Since the level of ordering of an array is not an easily quantifiable parameter, it is difficult to predict algorithmic behavior based on it. In this case, using a parallel computer, we could accelerate a solution for some instances of the sorting problem in at least two ways. One approach would execute a different sorting algorithm on each processor for the same data set and deliver the first available sorted array. Another approach would essentially use a parallel version of divide-and-conquer: the problem is broken down into multiple subproblems that are solved in parallel. Several techniques exist for graphs that divide a graph problem into other ones with algorithms to be solved in parallel [Thurimella, 1989]. Another application of a parallel technique could be for a very efficient minimum spanning tree algorithm by Fredman and Tarjan using Fibonacci heaps [Fredman and Tarjan, 1987]. This algorithm creates distinct partial spanning trees starting from different nodes and then condenses each tree into a super node to form another graph. The process is repeated on the new graph until the complete spanning tree is found, i.e., a single super node remains. A good description of this algorithm is provided in [McHugh, 1990]. By growing a partial spanning tree on a single processor, the entire process can be conducted in parallel with the results mixed after each iteration. This would be the equivalent of combining several instances of the same algorithm to form a hybrid.

The Shortest Path Hybrid Algorithm

The shortest path problem is well established in graph theory with extensive applications in a wide range of fields. We believe that by creating an efficient hybrid for a set of commonly used algorithms we can uncover the potential applications of combining deterministic algorithms.

The version of the shortest path problem used in our analysis is stated as follows: given a simple directed graph (digraph), assuming non-negative edge weights, determine the shortest path from every node to all other nodes. This version of the shortest path problem is also known as all sources to all destination shortest path problem since we are to find the shortest path from each node (source) to all other nodes (destinations).

Two well-known shortest path algorithms are by Dijkstra [Dijkstra, 1959] and Floyd [Floyd, 1962]. Both algorithms have a complexity of $O(n^3)$ where n is the size or the number of nodes of the input graph. We can enhance Dijkstra's algorithm by using a priority queue. If a Fibonacci heap data structure [Fredman and Tarjan, 1987] is used to implement the priority queue, we gain an asymptotic improvement over a naive implementation of Dijkstra's algorithm [Horowitz, Sahni and Freed, 1993]. The complexity of the modified algorithm is $O(n^2 logn + ne)$ where e is the number of edges in a graph. It is important to note that the complexity of the two methods is comparable and this fact makes them good candidates for trial parent algorithms. Deterministic algorithms with significantly different complexities present a much smaller probability of generating a successful hybrid.

We define *density* of a simple graph of size n as the ratio of the number of edges e to the maximum number of possible edges in that graph, e_{max}. The maximum number of edges a simple graph can contain is $n(n-1)/2$ and a simple digraph can have twice as many edges. Hence the *density* of a simple digraph is defined mathematically:

$$density = e / n(n-1) \qquad (i)$$

This definition forces *density* to be a number between zero and one that is directly proportional to the number of edges e for a fixed n.

The Experiment

Since the complexities of the two algorithms under analysis were a function of n and e (or *density*, since it is directly proportional to e for a given n), our aim was to develop a model for the algorithms' behavior as a function of these two parameters. We achieved this by measuring the execution times of the two algorithms for a wide range of input graphs. We used a graph generator that created a random graph of a specified n with approximately the given number of edges, i.e., the number of edges specified were generated randomly and then repeated edges were deleted, thus the specified e was an upper limit on the actual e. We also added non-negative random weights to each edge. By changing the parameters n and *density* of the input graph we could observe the behavior of the candidate parent algorithms. The experiment was performed on IBM RISC6000/350 machines.

To observe the behavior of the two algorithms as a function of the two input parameters n and *density*, we proceeded as follows: For a single execution of the experiment we held one of the variables to a constant value and varied the other. Then by multiple executions of the experiment, with a different value of the variable being held constant each time, we were able to obtain the behavior of the two algorithms as a function of the two parameters.

First we chose *density* to be constant for a single run of the experiment and *n* was varied in the range of 100 to 900 nodes. A sample run of this experiment is shown in Figure 1, where *density* was kept constant at a value of 0.28. We depict a limited input range in the figure to emphasize the important value of *n*, 160 in this case, that forms the cross-over point where the two algorithms switch places and one becomes faster than the other. We call this cross-over point the hybrid point because of its relevance in creating the hybrid algorithm. For an execution of the experiment with a different *density* we obtain curves similar to the ones shown in Figure 1, except with a different hybrid point. Thus from multiple execution of the experiment, with a different *density* value in the range of zero to one for each case, we were able to obtain the behavior of the hybrid point as a function of *density*.

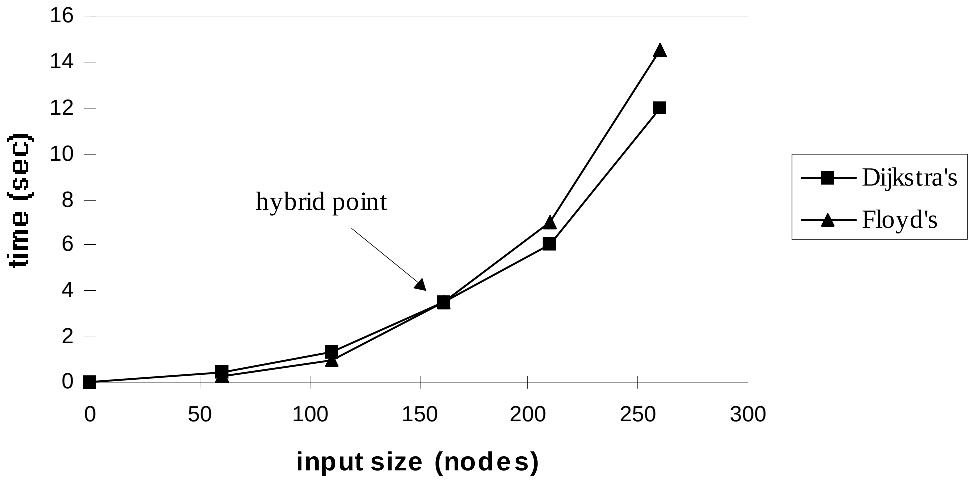

Figure 1: Behavior of Dijkstra's and Floyd's algorithms with increasing *n* for a fixed *density* value of 0.28

In another version of our experiment we timed the two algorithms for a single execution of the experiment by fixing the *n* to a precise value and varying *density* of the graph over a wide range (zero to one) and then obtained hybrid points from multiple executions of the experiment, each with a different *n*, in the range of 100 to 900 nodes. The *density* curves resulting from this experiment were in the form of straight lines, shown in Figure 2 for a sample execution with *n* fixed at 700. The straight lines are explained by the fact that Floyd's algorithm is independent of *density* while the modified Dijkstra's algorithm is linearly dependent on *e*, hence on *density*, for a given *n*. These results clearly indicated a hybrid point at a certain *density* for a given *n*, 0.56 in the case of Figure 2 for a given *n* of 700.

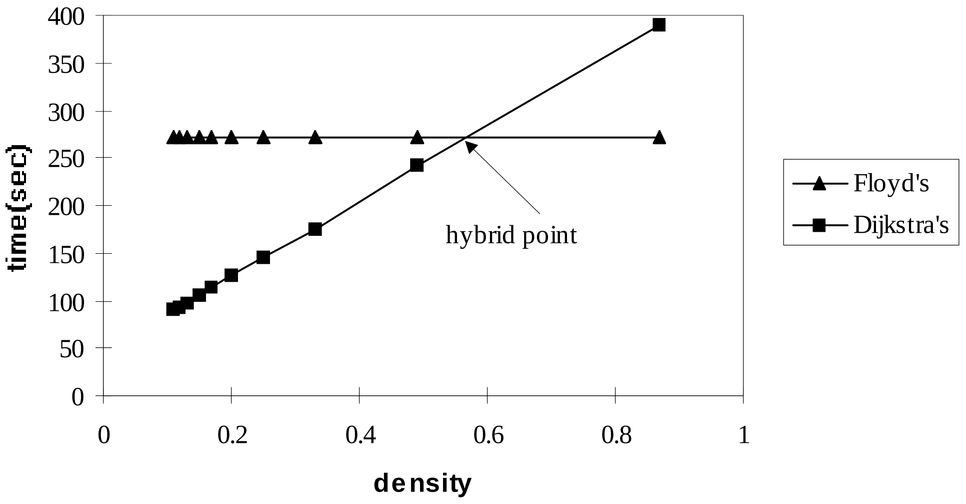

Figure 2: The behavior of the parent algorithms with *density* for *n* fixed at 700

To obtain the hybrid point as the function of *n* we executed multiple runs of the above experiment with a different *n*, ranging from 100 to 900, for each execution and plotted the resulting hybrid points in Figure 3.

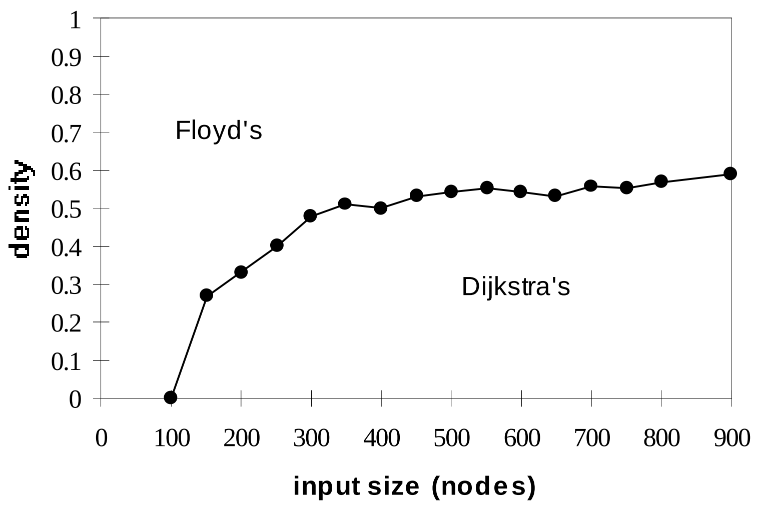

Figure 3: The solution space map for the hybrid shortest path algorithm, containing the hybrid curve that divides it into two regions for Floyd's and Dijkstra's algorithms.

Figure 3 represents a two dimensional solution space map for the shortest path problem encompassing all possible *density* values and *n* ranging from 0 to 900 nodes. The curve joining the hybrid points divides the solution space map into two regions, one where Floyd's algorithm is dominant and the other where the modified Dijkstra's algorithm reigns. Theoretically Dijkstra's algorithm with Fibonacci heaps is expected to perform efficiently for relatively sparse graphs, i.e., when *e* is much smaller than $O(n^2)$, otherwise it is slower than Floyd's method. This results from the fact that if *e* is $O(n^2)$ then the complexity of Dijkstra's algorithm transforms from $O(n^2 logn + ne)$ to $O(n^2 logn + n^3)$ or simply $O(n^3)$ with a much larger constant than Floyd's algorithm which is also $O(n^3)$. The map in Figure 3 is consistent with theoretical expectations, i.e., Dijkstra's algorithm occupies the sparser region of the map and Floyd's algorithm inhabits the region with higher *density* values.

A graph, represented by its *n* and *density*, will fall either in Dijkstra or Floyd region of the map, and automatically select the faster algorithm. Thus by interpreting a solution space map as a function of some parameter of the input, we can choose the faster algorithm for a particular instance of the problem. An approximate map can be created by analyzing the computational complexity of individual algorithms and then refined empirically for specific environments.

Analysis

From analysis of the actual implementation of the algorithms it was clear that Floyd's algorithms behavior could be modeled very effectively by its complexity with multiplication by a constant. Hence the actual behavior of Floyd's algorithm is mathematically:

$$Floyd(n,e) \cong \alpha n^3 \qquad (ii)$$

where α is a machine dependent constant

The complexity of Dijkstra's algorithm with Fibonacci heaps is based on the concept of cost amortization and thus represents a fairly good approximation to the actual behavior of the algorithm. We found it sufficient to multiply each term in the complexity function by a constant to model the actual behavior of the algorithm. Mathematically:

$$Dijkstra(n,e) \cong \beta n^2 logn + \gamma ne \qquad (iii)$$

where β and γ are machine dependent constants

From our definition a hybrid point exists where the two algorithms perform similarly, hence we could obtain a hybrid point by equating *(ii)* and *(iii)*:

$$\beta n^2 \log n + \gamma n e = \alpha n^3$$

since we need the hybrid point as a density value we use (i) to obtain

$$\beta n^2 \log n + \gamma n 2(n-1) density = \alpha n^3$$

$$\Rightarrow \quad density = (\alpha n - \beta \log n) / \gamma (n-1)$$

$$\Rightarrow \quad hybrid(n) = (\alpha n - \beta \log n) / \gamma (n-1) \qquad (iv)$$

taking the limit as n tends to infinity we obtain

$$hybrid(n) = \alpha / \gamma \qquad (v)$$

(v) represents the asymptotic value of the hybrid point as a function of *n*. It is entirely a function of the constants that themselves are dependent on the implementation and architecture used. To obtain the constants α, β and γ is a matter of reading them from the graphs of the *density* curves as the one shown in Figure 3. α, β and γ can be read from the y-intercept of the Floyd curve, y-intercept and the slope of the Dijkstra curve respectively. We obtained the values of the constants for all executions of the experiment and found minor differences in the recorded values. The mean of the constants with 95 % confidence intervals are:

$$\alpha = 0.80 \times 10^{-6} \pm 1.39 \times 10^{-9}$$

$$\beta = 4.96 \times 10^{-6} \pm 1.03 \times 10^{-7}$$

$$\gamma = 1.12 \times 10^{-6} \pm 1.17 \times 10^{-8}$$

From these values the asymptote for the hybrid curve in the solution space of Figure 3 becomes:

$$\alpha / \gamma = 0.80 / 1.12 = 0.71$$

A plot of *(iv)* is shown in Figure 4, compared with the experimental results. The theoretical values for the hybrid point match the empirical results sufficiently well to justify our analysis. Thus to implement the hybrid on a different computer we need only to determine the constants from a single execution of the experiment and then use the equation to determine the hybrid curve, or the asymptote for large inputs. It should be noted here that the performance gain promised by the hybrid increases as we move farther away from the hybrid curve in the solution space on either sides. Thus even an approximate knowledge of the hybrid curve in the solution space can enable us to achieve higher algorithmic performance for uniformly distributed input.

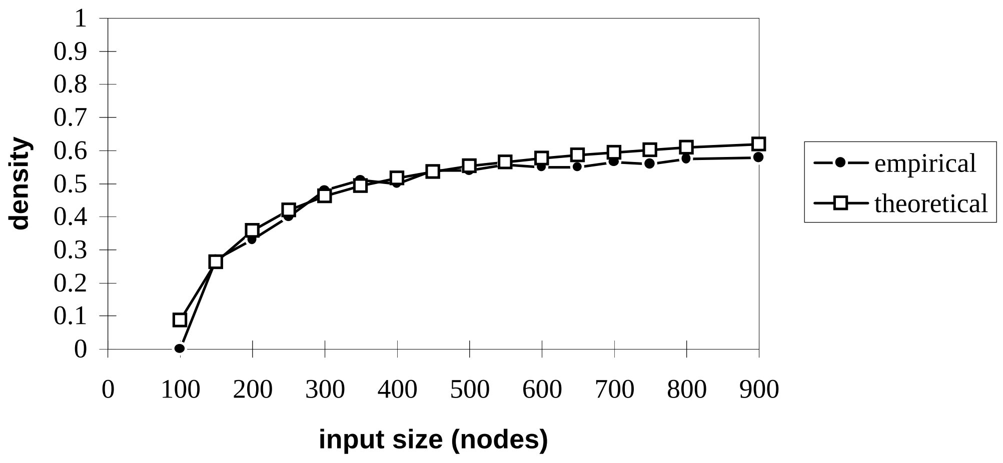

Figure 4: A comparison of theoretical data points ,obtained from *(iv)* with the mean of the constants, and empirical results for the hybrid curve in the solution space map.

Performance Gain Analysis

We calculated the percent improvement in performance for the hybrid algorithm relative to each parent method using the following equation:

$$G = (T_p - T_h)/ T_p \times 100$$

where

T_p = parent algorithm execution time

T_h = hybrid algorithm execution time

G = percent performance gain

To calculate the time for each algorithm we assumed a uniform distribution of *density* for a given n. This allowed us to obtain, for a given n, the total time from each algorithm as the area under its corresponding *density* curve. Such a curve is shown for n of 700 in Figure 2. The *density* curve for the hybrid is simply the curve of the faster algorithm at any point. Hence the *density* curve for the hybrid in Figure 2 consists of Dijkstra's curve preceding the hybrid point and of Floyd's curve succeeding it. We plotted the results of this performance gain analysis as a function of n in Figure 5.

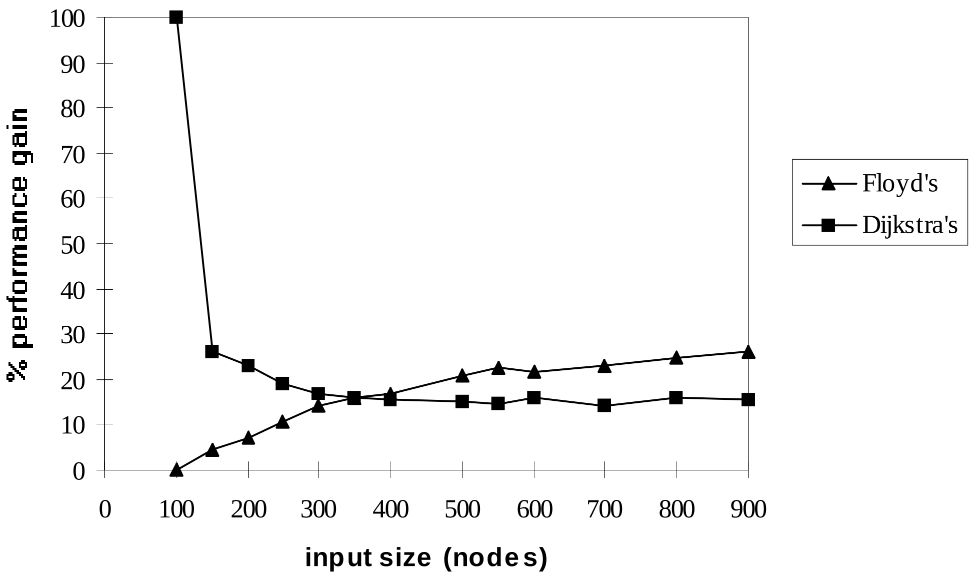

Figure 5: The percent increase in performance of the hybrid relative to each parent algorithm

As expected, the hybrid provides significant savings for small *n* relative to Dijkstra's algorithm since we can see from the solution space map in Figure 3 that this algorithm does not occupy the region for graphs less than 100 nodes. The hybrid for graphs with *n* less than 100 consists solely of Floyd's method. With increasing *n* we see the performance gain relative to both algorithms reaches an asymptotic value, approximately 26% for Floyd's and 16% for Dijkstra's algorithm. This is a considerable improvement in algorithmic performance for the shortest path problem given a large solution space.

Conclusion

We have presented a method of attaining higher algorithmic performance by introducing a deterministic hybrid technique. The technique presented in this paper is based on predicting the behavior of a set of algorithms on a well defined property of the input. We also showed how two algorithms can occupy separate regions in a solution space map that is a function of one or more input parameters. By using this map to select the faster algorithm for specific instances of a problem, we can create a new, more robust hybrid. Empirically, a solution space map was created for Dijkstra's and Floyd's algorithms for the shortest path problem, as a function of *density* and *n* of the input graph. It was followed by a theoretical analysis to supplement the experiment and to obtain an asymptotic bound for the empirical results. We were able to achieve, assuming uniform distribution of *density*, 16% and 26% improvement over Dijkstra's and Floyd's algorithm respectively, for large graphs. A point to note is that since our results were obtained experimentally, these are machine dependent and will vary slightly depending on the computer system organization.

For some deterministic problems it may be difficult to find a simple solution space map because of an algorithm's dependency on a less easily quantifiable input parameter. In such cases the parallel approaches of competing algorithms or divide-and-conquer, may prove more effective.

References

Malek, M., Guruswamy, M., Owens, H. and Pandya, M. 1989. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*. **21**, 59-84.

Malek, M., Guruswamy, M., Owens, H. and Pandya, M. 1989. A hybrid algorithm technique. Dept. of Computer Sciences Tech. Rep. 89-6, The University of Texas at Austin, Texas; translated and reprinted in Japanese book entitled H. Kitano ed. 1993. *Genetic Algorithm and its Applications*. Sangyo Tosho publishing Co., Ltd.

Kido, T., Kitano, H. and Nakanishi, M. 1993. A hybrid search for genetic algorithms: Combining genetic algorithms, tabu search, and simulated annealing. In *Proc. of 5th Intl. Conf. on Genetic Algorithms*. 641.

Hogg, T. and Williams, C. P. 1993. Solving the really hard problems with cooperative search. In *Proc. of 11th Natl. Conf. on Artificial Intelligence*. 231-236.

Dijkstra, E. W. 1959. A note on two problems in connection with graphs. *Numerische Math.* **1**, 269-271.

Floyd, R. W. 1962. Algorithm 97: Shortest paths. *Communications of ACM*. **5**, 345.

Thurimella, R. 1989. Techniques for the design of parallel graph algorithms. Ph.D. Thesis. Dept. of Computer Sciences, The University of Texas at Austin, Texas.

Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of ACM*. **34**, 596-615.

Horowitz, E., Sahni, S. and Freed, S. 1993. *Fundamentals of Data structures in C*. Computer Science Press, New York.

McHugh, J. A. 1990. *Algorithmic Graph Theory*. Prentice Hall, New Jersey.