Faster Multiobjective Heuristic Search in Road Maps*

Georgia Mali[1,2], Panagiotis Michail[1,2] and Christos Zaroliagis[1,2]

[1] Computer Technology Institute & Press "Diophantus", N. Kazantzaki Str., Patras University Campus, 26504 Patras, Greece

[2] Dept of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece

Email: {mali, michai, zaro}@ceid.upatras.gr

***Abstract*— We present new implementations of heuristic algorithms for the solution of the multiobjective shortest path problem, using a new graph structure specifically suited for large scale road networks. We enhance the heuristics with further optimizations and experimentally evaluate the performance of our enhanced implementation on real world road networks achieving 10 times better performance with respect to the best previous study.**

***Index Terms*—multiobjective shortest path problem, goal-directed search, heuristic search, transportation networks**

I. Introduction

Multiobjective optimization is a key area attracting great interest and intense studies in the last 60 years [4,5]. This area studies decision making problems which operate on an input of multiple conflicting objectives or criteria, trying to determine their best possible combination. In such problems there is no concept that captures a single optimal solution but, rather, multiple solutions, each representing a different tradeoff between the input objectives. These form the so called Pareto set, or the set of non-dominated solutions of the problem.

One core problem in this area is the multiobjective shortest path problem, appearing in applications such as QoS routing in communication networks, traffic equilibria, transport optimization and route and itinerary planning [4,5,6,15]. It is an extension of the single criterion shortest path problem using a cost vector (instead of a scalar) per network edge.Even though numerous efficient algorithms exist for the single criterion shortest path problem, the multiobjective counterpart of the problem is much harder. In fact, it has been shown to be NP-complete [6] (as indeed is the case with almost all multiobjective optimization problems).

Two main approaches are followed in order to reduce the computation effort for solving the multiobjective shortest path problem; the first one uses *approximation* and computes optimal solutions up to a certain factor [15]. Approximation techniques do not necessarily yield exact solutions, but are adequately fast to be used in practice [1]. The second approach uses *heuristic improvements* to speed-up existing algorithms [10, 14]. Such techniques yield exact solutions, but it is considerably more difficult to achieve good performance.In this work we focus on the latter approach, motivated by the great demand in practical applications to achieve efficient and exact multiobjective shortest paths. The currently best heuristic algorithm for exact multiobjective shortest paths is NAMOA* [10] in combination with the bounded TC heuristic [16] as shown in [8]. This approach uses an extension of Dijkstra's algorithm [2] and A* search [7] to reduce the *search space* (visited part of the underlying network), which is exact, has very good performance and requires little or no preprocessing at all.

In this work, we present a new implementation of NAMOA* on a new graph structure [9] specifically suited for large-scale networks. Our implementation is enhanced with new heuristic optimizations that improve memory access and space efficiency. We evaluate our implementation using the same data sets as the best previous experimental study [8]. Our experimental results show a clear superiority of our enhanced implementation over previous results (roughly 10 times faster) using a hardware platform that is about 30% slower.

II. Preliminaries

Let $G=(V,E)$ be a directed graph. Each edge is assigned with a cost vector $L=(w_1,w_2,\ldots,w_k)$, where k is the number of different criteria. Let L,L' be two cost vectors. We say that L *dominates* L' if there is i, $1\le i\le k$, such that $w_i < w'_i$ and $w_j \le w'_j$ for each $1\le j\le k$, $i\neq j$. Accordingly, in this case we say that *is dominated by*. Given two cost vectors , the sum of these is defined as . The set of all non-dominated vectors is called *Pareto Set*.
A path in is a sequence of nodes such that there is an edge from each node to the next one in the sequence. The cost label of the path is defined as:

$$L(P_{uv})=L(u,u_1)\oplus L(u_1,u_2)\oplus\ldots\oplus L(u_i,v).$$

For two given nodes *s* and *t* in G, let P_{st} be the set of all *s*-*t* paths. In the multiobjective shortest path problems we wish to compute the Pareto set for P_{st}.

III. Packed-memory Graph Structure

A. Overview

For our implementation, we used the *packed-memory graph (PMG)* structure, proposed in [9]. This is a highly optimized graph structure, part of a larger algorithmic framework, specifically suited for large scale networks. It provides dynamic memory management of the graph and thus the ability to control the storing scheme of nodes and edges

*This work was supported by the EU FP7/2007-2013 (DG CONNECT (Communications Networks, Content and Technology Directorate General), Unit H5 - Smart Cities & Sustainability), under grant agreement no. 288094 (project eCOMPASS)

© 2012 ACEEE

DOI: 02.ICT.2012.3.16

in memory for optimization purposes. It supports almost optimal scanning of consecutive nodes and edges and can incorporate dynamic changes in the graph layout in a matter of *μs*.

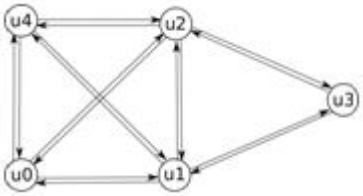

Fig. 1. An example graph.

B.Structure

The PMG structure consists of an array for storing the nodes, in an arbitrary order, and two arrays for storing the edges, one considering the edges as outgoing from the nodes and one considering them as incoming to the nodes. The storing order of the edges follows the order of their base node in the node array. In the outgoing (incoming) edge array, the edges are stored in sorted order by source (target) node. Thus, all outgoing (incoming) edges of a node *u* lie in consecutive cells in the outgoing (incoming) edge array. Each node keeps pointers to the respective range of cells containing its outgoing (incoming) edges in the outgoing (incoming) edge array. Each outgoing (incoming) edge keeps a pointer to its target (source) node and a pointer to its sibling incoming (outgoing) edge.

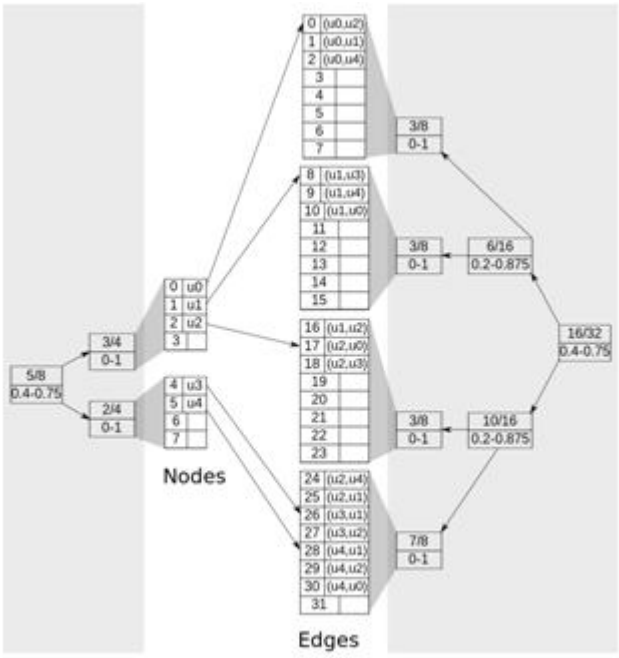

Fig. 2. Packed-memory Graph Structure

Finally, all three arrays reserve more memory than they need for their elements. The extra memory cells are evenly distributed throughout the arrays forming *holes* in them, which can then be used to efficiently add new elements. The maintenance of these holes is carried out by the use of a binary tree over each array which monitors the density of the holes within certain intervals of the array and redistributes the holes evenly when needed.An illustration of the PMG structure, for the example graph of Figure 1, is shown in Figure 2.

C. Internal Node Reordering

Given an arbitrary ordering of the nodes, the PMG can store them in consecutive memory addresses in the same order. This order can be changed at any time in an online manner. This operation is called *internal node reordering*. Clearly, any node reordering causes the edge arrays to be reordered as well.The power of this operation stems from the insight an algorithm might have about the sequence of its memory accesses. Such an algorithm can configure the ordering of the nodes in memory in such a way that will increase the locality of references and read/write operations will cause as few memory misses as possible.

D. Operations

The PMG structure can scan S consecutive nodes or edges in $O(S)$ time and $O(S/B)$ memory transfers, where B is the size of the block transferred between the memory layers. Hence, during Dijkstra's algorithm, it can access all outgoing edges of a node very efficiently.
Insert/Delete operations are very efficient as well. Since there exist holes in both the node and edge arrays, inserting an element involves finding a hole near the desirable insertion point and rearranging the elements in a small range around the hole. This rearrangement not only makes room for the new element but also keeps the elements ordered. The deletion operation is very similar, deleting an element and then rearranging the elements in a range around it. It is shown in [9] that the time and memory of the update operations are polylogarithmic in the size of the graph.

IV. Multicriteria Shortest Paths

In this section, we review the most important methods for finding all Pareto-optimal solutions in the multiobjective shortest path problem.

A. Multiobjective Dijkstra's Algorithm

In order to find Pareto-optimal paths with respect to given criteria and a given *s-t* query, an extension of Dijkstra's algorithm [11] can be applied. Each node keeps a list of labels (cost vectors) containing an entry for each criterion and a reference to its predecessor on the path. These labels represent paths that are not dominated by any other label on the node. During an iteration of the algorithm, the lexicographically minimal label is extracted from the queue, and the node *u* associated with it is processed. The process starts with scanning all outgoing edges of *u* and updating the labels on the neighboring nodes provided an update is feasible. For each neighboring node *v*, a new label is created representing the path to *v* through *u*. This label is then compared to the existing list of labels of *v*, and is inserted in the queue and the list of labels of *v*, only if it is not dominated

© 2012 ACEEE
DOI: 02.ICT.2012.3.16

by any other label. Furthermore, previously existing labels of *v* that are dominated by the newly computed label are discarded.The algorithm stops when the queue is empty. Then, the labels of *t* represent all non-dominated paths from *s* to *t*. The paths can be reconstructed by following the predecessor pointers of each label.

B. NAMOA Algorithm*

Proposed in [10], the NAMOA* algorithm incorporates the A* search technique with the multiobjective Dijkstra's algorithm along with several optimizations.

The core *s-t* query is similar to the multiobjective Dijkstra's algorithm with some extensions. The main difference is that the priority of a label in the queue is modified according to a *heuristic function* $h_t : V \rightarrow \Re^k$. This function gives a lower bound estimate $h_t(u)$ for the cost of a path from a node *u* to a target node *t*. By adding this heuristic function to the priority of each generated label of a node, the search is pulled faster towards the target. The tighter the lower bound is, the faster the target is reached. Hence, different heuristic functions yield different performances.

In Section IV.D we describe the heuristic functions used for our evaluation.In order for the A* search extension to have as large an effect as possible, more modifications have been introduced to the core multiobjective Dijkstra's algorithm [10]. First, the list of labels on a node *u* are split into two sets, $G_{op}(u)$ and $G_{cl}(u)$ where the first contains the labels that are also present in the queue, and the second contains the rest. This way, when discarding a label from the list that is also present in the queue, it is discarded from the queue as well.

Moreover, as soon as the target node is reached through a non-dominated path P_{st}, this path gets recorded and the search is pruned by it. On each iteration, when a label representing a path P_{su} is extracted from the queue, a check takes place; if the label representing is dominated by the label representing , then there can be no path to *t* consisting of that is not dominated. Therefore, the label representing is discarded, and the search is pruned at this point.
It is clear that the fastest a first non-dominated path to *t* is discovered, the earliest the search will be pruned. Hence, the heuristic function that pulls the search towards the target is a very important factor affecting the overall performance of the algorithm.

C. Optimizations

We have incorporated our own optimizations in the NAMOA* algorithm, mainly to increase the efficiency of the memory accesses. First, we do not keep G_{op} and G_{cl} as different entities on each node. Instead, we have combined them into one list of labels, and have extended the actual labels to contain a flag determining whether a particular label is in the queue or not. This way, all labels of a node, either open or closed, reside on consecutive memory addresses, yielding less cache misses. Second, we keep a pointer on each label, pointing to the predecessor node that generated it. Thus, the predecessor graph, and all non-dominated paths can be induced by following these predecessor pointers.
Third, we have made the following observation. Any label *L* residing on a node *u* during an iteration of the algorithm represents a currently non-dominated path P_{su}. This path might have been the prefix of more non-dominated paths P_{sv} towards a node *v*. The paths are a concatenation of and some path. In case becomes dominated by another path , then all paths will be dominated by consisting of and the original. Hence, when discarding a dominated label *L* from the list of a node *u*, we search forward for all subsequent labels of other nodes *v* generated by *L* and discard them both from the lists and from the queue.

D. Heuristics

Great Circle distance heuristic. Since our focus is on road networks, a straightforward lower bound for the distance of a route *s-t* is the "flying" distance from *s* to *t*. In Cartesian coordinates this would be the Euclidean distance, as in the length of the straight line connecting *s* and *t*.

However, this does not take into consideration the Earth's curvature, which results in incorrect lower bounds. Instead, we use the Great-circle distance which measures the distance between two points along the surface of a sphere, since the Earth's shape resembles a sphere.

Even though this is adequate for approximating a lower bound on the distance between two points, it cannot be used for metrics that are not correlated to the distance. However, computing a lower bound for travel time is still straightforward. Since a lower bound for the distance exists, as well as upper bounds for the speed of travelling (speed limits), the lower bound for the travel time between two points can be easily deduced.

TC heuristic. Tung and Chew in [16] have proposed the following heuristic. Let be the heuristic function of a node *u* during a search towards a target node *t*. The heuristic function consists of the shortest distances from *u* to *t* with respect to only one criterion at a time. For each criterion *i*, a single-criterion shortest path tree is grown from *t* on the reverse graph , and each shortest path distance is recorded. Then, the heuristic is the combination of the shortest path distances for each criterion, $h_t(u) = (c_1^*(u,t), c_2^*(u,t), \ldots, c_k^*(u,t))$. Clearly, this is a lower bound for any generated distance label of node *u* using the NAMOA* algorithm.Bounded calculation for the TC heuristic. The TC heuristic must build a full reverse single-criterion shortest path tree for each criterion of the problem. Even though the single-criterion search is efficient, this process is executed during the query, which clearly must be as fast as possible. There is a way to reduce the search space according to [8]. For simplicity purposes, we assume that the number of different criteria is two. The following reasoning can be extended to multiple

© 2012 ACEEE
DOI: 02.ICT.2012.3.16

criteria. Let $c_1^*(u,t)$ be the shortest path cost from *u* to *t* with respect to the first criterion. The cost of this path using the second criterion is denoted as $c'_2(u,t)$ and is clearly not the minimum. Accordingly, $c_2^*(u,t)$ is defined as the shortest path cost from *u* to *t* with respect to the second criterion, and the cost of this path using the first criterion is denoted as $c'_1(u,t)$. It has been shown in [10] that NAMOA* does not consider paths with costs that are dominated by $(c'_1(u,t), c'_2(u,t))$ since this can never lead to non-dominated solutions. Hence, the single-criterion shortest path tree search can be stopped as soon as it reaches these bounds.

In particular, given an *s-t* query, the following steps are carried out:

1. A reverse single-criterion shortest path tree is grown from *t* using the first criterion. During the search, for each node, the shortest path distance towards *t* is assigned as the first criterion heuristic for this node. The search is stopped as soon as it reaches *s* with cost $c_1^*(s,t)$. The cost of P_{st} using the second criterion is recorded as $c'_2(s,t)$ and the search is paused at this point.
2. A reverse single-criterion shortest path tree is grown from *t* using the second criterion. In the same manner as in the first step, each node *u* gets assigned its lower bound for the second criterion. The search is stopped as soon as the minimum cost in the queue is greater than $c'_2(s,t)$. Clearly, node *s* is settled before quitting the search, and gets assigned the shortest path cost $c_2^*(s,t)$, with $c_2^*(s,t) \leq c'_2(s,t)$. The cost of this path using the first criterion is recorded as $c'_1(s,t)$.
3. The first search continues from the same point it was paused in Step 1. The search stops as soon as the minimum cost in the queue is greater than $c'_1(s,t)$.

V. Experimental Evaluation

To assess the performance of our graph structure and algorithmic implementations, we conducted a series of experiments on the proposed shortest path routing algorithms on real world large-scale transportation networks (USA road networks) with two criteria. The first criterion is the actual distance and the second criterion is the travel time between two intersections. The travel time is not always relative to the actual distance, since different roads have different speed limits in practice.

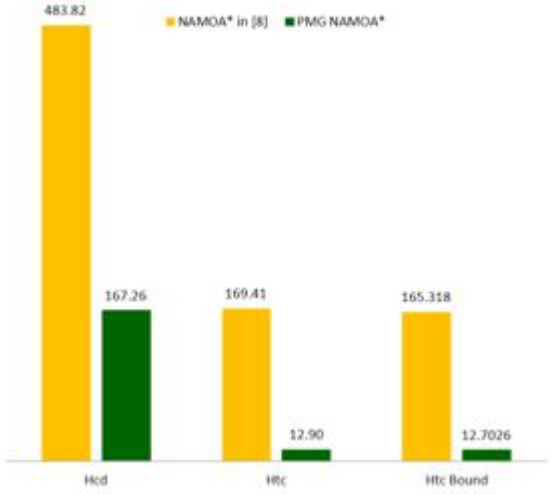

Fig. 3. Mean running times (sec) in New York City

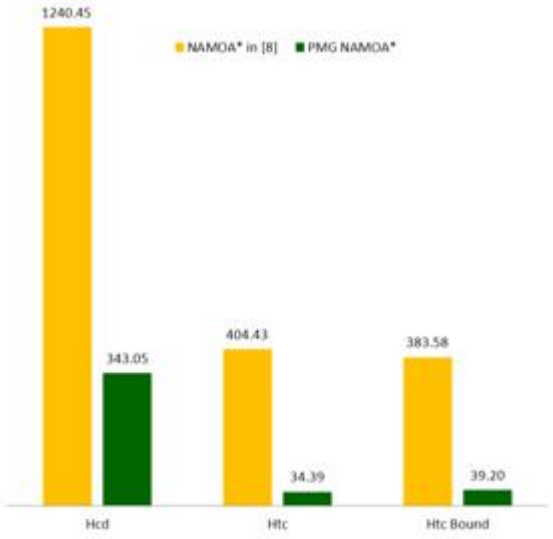

Fig. 4. Mean running times (sec) in Florida

A. Setup and Data

All experiments were conducted on an Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz with a cache size of 6144Kb and 8Gb of RAM. Our implementations were carried out in C++ and compiled by GCC version 4.4.3 with optimization level 3. According to [12, 13] our CPU is about 30% slower than the CPU used in [8] and our RAM size is 8 times smaller.

The road networks for our experiments were acquired from [3] and consist of the road networks of New York City and the state of Florida. The provided graphs are strongly connected and undirected. Hence, we consider each edge as bidirectional. We use these networks in order to directly

© 2012 ACEEE
DOI: 02.ICT.2012.3.16

compare the performance of our implementations to the results of previous works that were evaluated on the same networks.

B. Experimental Results

We directly compare the heuristic that has the best running times in the implementation in [8] with all our heuristic implementations. We denote as H_{cd} the great circle heuristic, as H_{tc} the TC heuristic and as H_{tc} *Bound* the bounded TC heuristic. We have used the same query set as in [8] in order to be directly comparable. We have measured the running times for each query using every heuristic. The running times reported here are the mean values of 10 query repetitions. For each query we have cross-referenced the number of non-dominated solutions in order to assess the correctness of our implementation, in each case being the same as in [8].

Tables I and II show the running times for each single query on the road maps of New York City and Florida respectively. The time is measured in seconds and values that are omitted are running times that exceed the one-hour limit. The last column, namely *Ratio*, is the ratio of the running times of the bounded TC heuristic in [8] to the running times of the bounded TC heuristic in our implementation, which are directly comparable.

The results in [8] are confirmed by our evaluation. The best running times are achieved using the TC heuristic, either bounded or not. The difference between these two heuristics is the initial computation of the heuristics, not the actual running time of NAMOA*.Our running times are much better than the running times in [8]. This is apparent by the ratio,shown in the last column of each table. There are queries where our implementation is 40 to 50 times faster (bold letters denote a ratio grater than 20). Even with the worst heuristic,the great circle heuristic, in some cases (italic letters) our implementation can outperform the bounded TC heuristic in [8], especially in smaller networks.

In order to have a more illustrative indication of our performance gains, we have plotted the mean times of the query set using each heuristic in both our implementation and in [8]. These can be seen in Figures 3 and 4. Our implementation is denoted as *PMG NAMOA** due to the use of the PMG structure.

It is apparent that the PMG NAMOA* is much faster in each heuristic. The mean ratio between the bounded TC heuristic implementations is 13 in New York City, and 9.7 in Florida. Therefore, we can safely claim that our implementation is roughly 10 times faster on these large-scale networks, on the given query set.

Conclusions

We have presented a new implementation of NAMOA* on a new efficient graph structure. We have suggested not only implementation optimizations, but also heuristic enhancements of the algorithm. Finally, we have assessed the superiority of our implementation through an experimental evaluation.

References

[1] D. Delling and D. Wagner. “Pareto Paths with SHARC”. In Proc. SEA’09, LNCS vol. 5526, pp. 125–136, 2009.

[2] E.K. Dijkstra, “A note on two problems in connexion with graphs,” Numerische Mathematik 1 (1959), pp.269-271

[3] 9th DIMACS Implementation Challenge - Shortest Paths, http://www.dis.uniroma1.it/challenge9/download.shtml

[4] M. Ehrgott, “Multicriteria Optimization,” Springer, Berlin 2000.

[5] M. Ehrgott and X. Gandibleux, “Multiple Criteria Optimization—State of the Art Annotated Bibliographic Surveys,” Kluwer Academic, Boston 2002.

[6] P. Hansen, “Bicriterion path problems,” Lecture notes in economics and mathematical systems, vol. 177, 1979, pp.109-127, Springer.

[7] P. E. Hart, N. J. Nilsson and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” IEEE Transactions on Systems Science and Cybernetics SSC-4, 2, pp 100–107, 1968.

[8] E. Machuca, and L. Mandow, “Multiobjective heuristic search in road maps,” Expert Systems with Application. 2012.

[9] G. Mali, P. Michail and C. Zaroliagis, “A new dynamic graph data structure for large-scale transportation networks,” eCOMPASS Technical Report, TR-003, July 2012.

[10] L. Mandow, and J. L. Perez de la Cruz, “Multiobjective A* search with consistent heuristics,” Journal of the ACM, 57, 27, 2010, pp. 1-25.

[11] E. Martins, “On a multicriteria shortest path problem,” European Journal of Operational Research, 16, 1984, pp.236-245.

[12] Passmark CPU Benchmark, Six-Core AMD Opteron 2435, http://www.cpubenchmark.net/cpu_lookup.php?cpu=[Dual+CPU]+Six-Core+AMD+Opteron+2435

[13] Passmark CPU Benchmark, Intel Core i5-2500 @ 3.30GHz, http://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i5-2500K+%40+3.30GHz

[14] B. S. Stewart and C. C. White, “Multiobjective A*,” Journal of the ACM 38 (4), pp.775-814.

[15] G. Tsaggouris and C. Zaroliagis, “Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications”, Theory of Computing Systems, Volume 45 Issue 1, April 2009, pp 162-186.

[16] C. T. Tung and K. L. Chew, “A multicriteria Pareto-optimal path algorithm”, in European Journal of Operational Research, 62, 1992, pp.203-209.

© 2012 ACEEE
DOI: 02.ICT.2012.3.16

TABLE I. RUNNING TIMES ON NEW YORK CITY

Source id	Target id	H_{sz} Bound [8]	H_{cd}	H_{sz}	H_{sz} Bound	Ratio
33502	163335	9.22	10.62	8.24	0.98	9.37
198561	195430	0.34	0.40	0.23	0.04	7.64
40851	4310	525.87	791.49	36.11	35.91	14.64
19103	95503	4.98	7.73	0.86	0.75	6.60
65190	57030	0.09	0.10	0.19	0.01	6.63
172882	189944	44.17	*22.01*	2.14	2.13	**20.78**
181176	151910	89.46	*36.88*	5.17	5.13	17.43
177414	103345	30.72	97.39	2.08	2.09	14.69
186166	71968	862.22	*546.61*	60.16	60.48	14.26
50616	76333	7.27	37.94	0.96	0.85	8.54
56699	159358	114.02	154.06	8.62	8.56	13.32
103987	175817	77.33	139.46	6.93	6.89	11.22
75533	165171	129.05	216.17	8.28	8.34	15.47
191865	72103	128.17	281.28	7.96	7.97	16.09
35170	237017	5.31	10.52	0.39	0.28	19.15
207442	156433	15.08	*7.43*	0.95	0.79	19.19
62306	134007	22.05	91.86	1.34	1.36	16.22
58427	135252	76.63	*56.92*	4.31	4.32	17.73
91985	200812	97.36	*63.15*	6.10	6.10	15.96
242644	163590	12.83	20.22	0.91	0.87	14.68
40180	100359	5.54	*3.72*	0.50	0.36	15.40
38497	207344	199.53	*77.63*	12.00	12.05	16.56
129948	7003	135.24	*81.44*	11.13	11.14	12.15
259195	173121	12.87	53.57	0.74	0.74	17.40
147806	136543	63.93	67.05	4.38	4.32	14.80
189934	31336	25.34	66.77	1.82	1.77	14.29
138263	253856	0.64	1.71	0.24	0.07	9.39
246144	166336	6.84	*4.00*	0.45	0.34	20.03
25610	143842	10.71	12.82	0.67	0.58	18.54
228779	167251	18.6	83.95	1.11	1.10	16.90
78936	34136	39.86	141.19	2.22	2.24	17.77
124173	138439	108.12	*86.41*	9.82	9.84	10.99
260563	233292	3.51	*2.66*	0.44	0.29	12.16
193168	66816	94.74	180.55	6.73	6.79	13.95
29432	29834	20.5	36.36	1.52	1.45	14.11
193241	144927	209.09	533.64	16.54	16.46	12.70
161522	171446	0.8	4.18	0.29	0.52	1.54
176910	109129	23.21	66.22	1.30	1.34	17.27
251416	53900	19.09	*14.72*	1.13	1.10	17.29
201505	262626	7.32	45.08	0.45	0.35	**20.93**
35252	18638	1.53	*1.24*	0.21	0.05	**32.60**
92562	65120	33.85	75.66	2.56	2.55	13.28
230423	2724	4.78	*3.18*	0.44	0.32	14.91
17285	92411	3.35	*2.04*	0.27	0.11	**29.29**
177037	199832	5.44	*1.04*	0.24	0.12	**45.94**
68330	206280	114.12	*85.98*	6.24	6.27	18.20
61414	50367	11.32	*4.61*	0.49	0.46	**24.40**
180834	83150	1948.15	*1153.73*	153.37	153.26	12.71
179874	57536	1495.73	*1172.39*	127.70	127.96	11.69
86937	190907	1389.98	1707.36	118.30	117.29	11.85

TABLE II. RUNNING TIMES ON FLORIDA

Source id	Target id	H_{sz} Bound [8]	H_{cd}	H_{sz}	H_{sz} Bound	Ratio
361739	698672	21.61	173.78	2.86	3.89	7.53
546667	1044042	41.77	662.16	5.23	5.88	7.13
115105	421966	96.59	-	7.38	9.24	10.55
742805	335320	804.18	-	62.99	95.10	8.46
88673	333047	1163.91	-	83.02	109.60	10.62
766579	263017	125.11	800.65	11.46	13.30	9.41
28100	848660	835.32	-	64.86	84.77	9.86
134765	11866	279.88	-	20.02	22.12	12.66
158576	949455	15.5	22.52	1.12	0.63	**24.63**
659282	327441	18.53	*17.10*	1.74	1.25	14.82
539004	639594	10.02	133.24	1.15	0.52	19.26
489044	463492	127.43	-	8.48	9.05	14.22
481860	1046443	19.87	108.05	3.03	2.50	8.00
273776	154436	50.86	158.14	2.78	2.72	18.97
946451	513773	62.32	869.99	6.57	6.61	9.48
90921	359195	1160.1	-	78.11	101.91	11.38
783218	996886	388.91	-	30.97	35.18	11.06
646797	149214	104.12	606.01	6.89	7.37	14.36
398569	982263	18.25	49.96	3.53	3.03	6.03
809772	870827	871.37	-	69.67	84.67	10.32
234560	955775	23.02	52.17	2.10	1.89	12.16
716892	344531	756.3	-	72.43	82.21	9.22
516174	154020	68.99	628.38	7.35	7.45	9.36
129998	118211	32.44	569.11	3.28	2.89	11.39
905861	756883	9.08	24.22	2.39	1.70	5.33
41614	404340	5.53	32.06	0.92	0.20	**27.63**
933700	561390	28.18	1274.26	0.88	0.55	**51.56**
237886	310889	864.33	-	69.35	81.12	10.74
257739	652062	2.86	20.72	1.47	0.90	3.18
478200	1062969	134.38	151.89	16.21	16.55	8.17
173720	246425	22.18	-	3.04	2.61	8.51
43974	803673	56.61	650.99	6.12	5.98	9.47
382275	1044332	68.37	-	6.76	6.61	10.38
462808	85391	13.72	*11.72*	1.60	1.05	13.12
818016	667330	74.78	349.52	9.11	9.03	8.28
257389	17759	916.32	-	74.27	86.82	10.55
16738	751658	269.02	-	28.83	32.17	8.36
364903	404709	119.92	698.98	11.81	12.00	9.99
472495	193187	26.99	132.33	2.52	2.11	12.79
131865	294161	10.91	*9.22*	2.16	1.53	7.11
363001	263258	195.72	712.02	16.16	16.57	11.81
310505	612278	3237.24	-	290.39	336.04	9.90
401809	616933	3341.38	-	377.59	398.69	8.56

© 2012 ACEEE
DOI: 02.ICT.2012.3.16