Acceleration of Radix-Heap based Dijkstra algorithm by Lazy Update

Tomohiro Takahashi Yasuhiro Takashima
Department of Information Systems Engineering
University of Kitakyushu
Kitakyushu, Fukuoka, 808-0135
tomohiro.takahashi@is.env.kitakyu-u.ac.jp takasima@kitakyu-u.ac.jp

Abstract— This paper proposes a fast Dijkstra algorithm with radix-heap by lazy update which solves the single source shortest path problem (SSSP). The conventional Dijkstra algorithm chooses one vertex with the minimum tentative distance among the unvisited vertices. For the problem, the relaxation of the number of selected vertices not only one but also multiple under the guarantee of its optimality has been proposed, called *lazy update*. In this paper, we utilize this lazy update method to the radix-heap based Dijkstra which solves SSSP with the integer edge distances. The experimental results confirm the efficiency of the proposed method which executes 50 % faster than the conventional Dijkstra.

I. Introduction

In these decades, the problem size is still grown up as the computer performance evolution. In fact, the speed-up of the computer cannot catch up with the growing speed of the problem. Thus, the efficiency of the algorithm becomes much important.

In the large-sized problem, the graph theoretical problems are the most popular ones. Especially, the shortest path problem is widely used in the world [1–3]. In the VLSI design, the shortest path problem is also much important [4]. The impact of the high efficiency to solve the problem must be large. This paper focuses on the improvement of the algorithm solving it.

The shortest path problem includes the single source shortest path problem (SSSP) which calculates the shortest path length of each vertex from the source vertex in the given graph. To solve SSSP, Dijkstra's algorithm [5] has been proposed if the graph does not have any negatively weighted edges. The Dijkstra's algorithm iteratively executes the following two steps: 1) Update the tentative distances for all unvisited vertices from the source vertex through only visited vertices; 2) Select the vertex with the minimum tentative distance among the unvisited vertices and change its status to visited. Its optimality is proven and its runtime complexity is $O(|V|^2)$ with a naive implementation or $O((|E| + |V|) \log |V|)$ with a priority queue [6]. It means that $O(|V| \log |V|)$ if the given graph is sparse, that is $|E| = O(|V|)$. Furthermore, the radix-heap is more suitable than the ordinary priority queue [8].

As mentioned above, the optimality of Dijkstra's algorithm is proven. It claims that the minimum distance of the selected vertex cannot be changed subsequently. However, its optimality is retained even if not only one vertex but also all vertices with the minimum tentative distance are selected. Moreover, when the vertices with (the minimum tentative distance) + (the minimum edge distance) are selected, it is also optimal. We proposed the algorithm, called *lazy update*, which utilizes the above considerations [7]. It reported the lazy update achieves 1.2 times faster. But, it experiments a few benchmarks only. Furthermore, we implemented it with priority-queue only. However, for the acceleration of Dijkstra's algorithm, the radix-heap is also proposed [8]. We also need to confirm the efficiency of Lazy-update with radix-heap.

The main contributions of this paper are as follows:

- We implement the radix-heap based Dijkstra with lazy update and enhance it by the modification of data-structure of radix-heap.
- The experimental results show that the radix-heap based Dijkstra with lazy update improves 50 % to the conventional Dijkstra implementation.

The rest of this paper is as follows: Section II describes the single source shortest path problem and Dijkstra's algorithm; Section III introduces the proposed method; Section IV reports the experimental results; and Section V concludes this paper.

II. Single Source Shortest Path Problem

A. Problem definition

The single source shortest path problem (SSSP) is the problem to find the shortest length of each vertex from the source vertex. The detail is shown as follows:

Input: Graph $G = (V, E)$ where each edge $e \in E$ has each non-negative distance, the source vertex $s \in V$

Output: The shortest path length of each vertex from the source vertex

B. Dijkstra's Algorithm

To solve SSSP, Dijkstra's algorithm has been proposed [5]. The detail flow of Dijkstra's algorithm [6] is shown in Fig.1. In the description, the tentative distance corresponds to the distance from the source vertex through only the *visited* vertices.

step 1: Mark all vertices *unvisited*.
step 2: Set the length of the source vertex to 0 and mark it *visited*. For the other vertices, the tentative distances are set to ∞.
step 3: Update the tentative distances of the unvisited vertex through the visited vertices.
step 4: Select the unvisited vertex with the minimum tentative distance and mark it visited.
step 5: If all vertices are marked visited, then finish this algorithm. Otherwise, return step 3.

Fig. 1. Dijkstra's algorithm

C. Priority Queue Based Acceleration of Dijkstra's Algorithm

To accelerate Dijkstra's algorithm, the priority queue based methods are widely used. In step 4 in Fig.1, the time complexity of the selection of the minimum distance vertex is $O(|V|)$, if a naive implementation is employed. Then, by replacing it to the priority queue, its complexity may decrease. Usually, the priority queue is based on the binary heap. Thus, the time complexity is $O(\log|V|)$ for each selection with the modification of heap-tree. But, to utilize the priority queue, changing step 4 is not enough. Step 3 also must be changed. This modification effects that the time complexity of step 3 is $O(|E|\log|V|)$ for whole execution. Thus, whole time complexity is $O((|E|+|V|)\log|V|)$.

D. Radix-Heap

There is another way of implementation of priority queue, called *Radix-heap* [8]. Radix-heap uses the most significant bit (MSB) of the binary representation of numbers. Its characterizations are 1) bucket based data-structure where the size of the bucket is the bit-width of the values, 2) each element x is classified into the $(\texttt{MSB}(last \hat{} x)+1$-th bucket, and 3) monotone priority queue: the inserted value must be equal to or more than *last*. In the above description, several notations are as follows:

- The function $\texttt{MSB}(x)$ returns the MSB value of x if $x! = 0$. If $x == 0$, then return -1;
- The variable *last* is the last extracting value, that is, the minimum value among the stored values in the buckets.
- $\hat{}$ corresponds to the bit-operation xor.

There are the basic operations, push and pop, for the radix-heap. The detail of each operation is as follows, where the array of the buckets is $b[i]$:

- push(x): The element x is placed into $b[(\texttt{MSB}(last \hat{} x))+1]$.
- pop: The minimum element is extracted. If $b[0]$ is not empty, the elements in $b[0]$ correspond to the minimum element. Then, output one element from them. Otherwise, search the first non-empty bucket among $b[1]$, $b[2]$, ... If such a bucket is found, then the minimum element is searched in the bucket and modify *last* to the searched value. Finally, re-distribute all elements in the bucket into $b[(\texttt{MSB}(last \hat{} x))+1]$ and output one element form $b[0]$.

To explain this data structure, a simple example is shown as follows, where the bit-width of the values is 4. The process of the example is:

1. An initial input data, $\{(1,a),(4,b),(6,c),(1,d)\}$;
2. three times pop;
3. $\{(5,e)\}$ is added; and
4. pop and finish.

In this example, the element (n,x) corresponds to the value (n) and the name (x) of element, respectively. After the process 1, the data structure is shown in Fig. 2 where $last = 0$. For the first pop in the process 2, search the minimum non-empty bucket and find $b[1]$ since $b[0]$ is empty. Next, find the minimum element in the bucket $b[1]$ and, as a result, $last = 1$. Then, re-distribute all elements in $b[1]$ into $b[(\texttt{MSB}(last \hat{} x))+1]$, shown in Fig. 3. For two pop's, output $(1,d)$ and $(1,a)$, and $b[0]$ is empty again (shown in Fig. 4). Then, search the minimum non-empty bucket again and find $b[3]$. In $b[3]$, the minimum element is 4, that is, $last = 4$. Thus, the result of re-distribution is shown in Fig 5. After the last pop in the process 2, the data structure is shown in Fig 6. The result of the process 3 is shown in Fig. 7. For the process 4, *last* is modified since $b[0]$ is empty (shown in Fig. 8). Finally, when all processes are finished, the data structure is shown in Fig. 9.

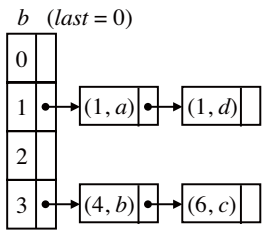

Fig. 2. After Process 1: initial status

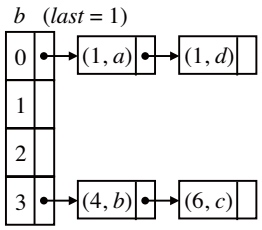

Fig. 3. During Process 2: find the minimum non-empty bucket and re-distribute the elements in the bucket

Note that $b[0]$ holds the elements with the minimum value, if not empty. The time complexity of push operation is $O(1)$. On the other hand, the time complexity of pop operation is $O(B)$, where B is the bit-width of the value.

III. Lazy Update Application to Radix-heap based Dijkstra's algorithm

[7] proposed the lazy update method. Lazy update is to select multiple vertices in step 3 of Dijkstra's algorithm. To guarantee the optimality, the selection of all vertices with the

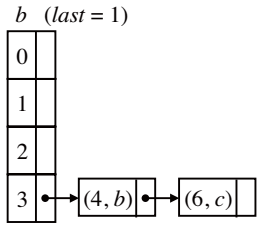

Fig. 4. During Process 2: two pop's are finished

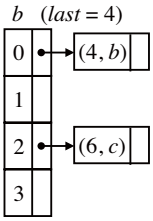

Fig. 5. During Process 2: re-distribute the elements in bucket[3]

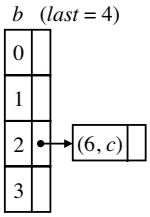

Fig. 6. After Process 2: the last pop is finished

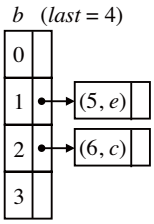

Fig. 7. After Process 3: element $(5, e)$ is added

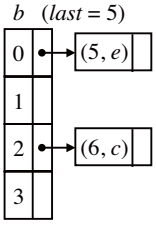

Fig. 8. During Process 4: find the minimum non-empty bucket and re-distribute the elements in the bucket

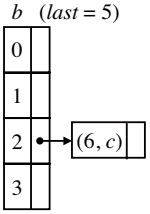

Fig. 9. Final Result

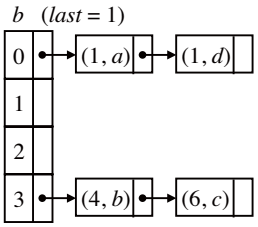

Fig. 10. An example of data-structure of radix-heap

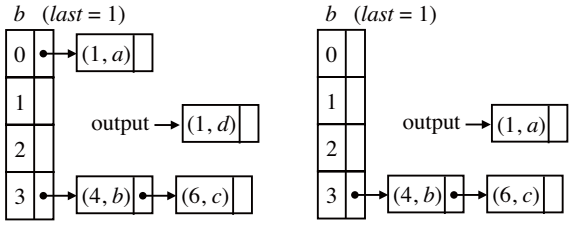

Fig. 11. Naive implementation

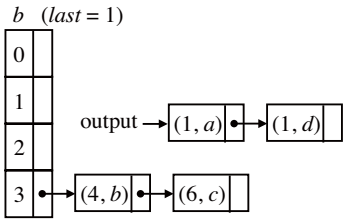

Fig. 12. Simultaneous Extraction method

minimum tentative distance is simple enhancement. But, to widen the selection range to (the minimum tentative distance) + (the minimum length of the edges) also achieves the optimal solution. In this paper, we try above two selection schemes. One is a naive implementation, the other is a utilization of the data-structure of radix-heap. The naive implementation is similar way mentioned in Section II. The second implementation is based on the data-structure of radix-heap. As mention in Section II, the bucket $b[0]$ must hold the minimum value's elements. Thus, for the pop operation, all elements in $b[0]$ can be extracted simultaneously instead of only one element. We utilize this observation.

We show the flow of both methods. Fig. 10 shows an example of the data-structure of radix-heap, again. For this situation, the naive implementation runs as shown in Fig. 11. On the other hand, the simultaneous extraction runs as shown in Fig. 12.

For both implementations, we must consider holding the monotone priority queue, that is, once some value is extracted, then the smaller value cannot be inserted. For example, for the case shown in Fig., we assume that pop operation with the limit value being 2 is executed. After the extraction of all elements valued 1, re-distribution must be done since $b[0]$ is empty (shown in Fig. 14). As a result, we cannot push the data valued 3 since *last* is equal to 4.

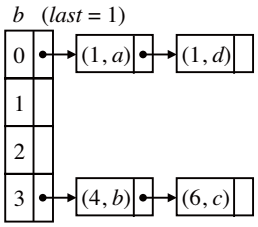

Fig. 13. Considering monotone priority queue

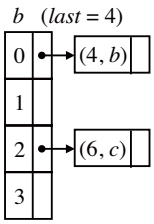

Fig. 14. Re-distribution result

To solve this issue, we insert a sentinel data with the limit value. As a result, it ensures that the last extracted value is equal to the limit value. Thus, it does not need to consider that the monotone priority queue must be held. For the case

shown in Fig. with pop operation with the limit value being 2, a sentinel data valued 2 is pushed (Fig. 15). Then, the situation after the extraction is shown in Fig. 16. For this case, we can push the data valued 3.

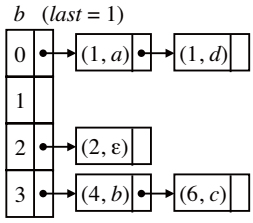

Fig. 15. Insertion of Sentinel Data

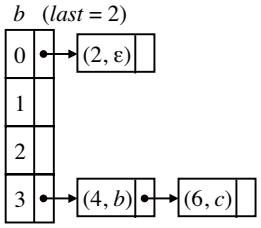

Fig. 16. After the Extraction

IV. Experiment Results

To confirm the efficiency of the proposed method, we implement the several following methods: Dijkstra's algorithm with ordinary priority queue (H), Dijkstra's algorithm with radix-heap (R), Dijkstra's algorithm with lazy update and ordinary priority queue (LH), and Dijkstra's algorithm with lazy update and radix-heap(LR). Furthermore, the implementation of LR has two types, a naive method (LRN) and a simultaneous extraction (LRS). For lazy update, the limit value varies the minimum tentative distance version (*0) and (the minimum tentative distance) + (the minimum edge distance) (*1). Thus, the variation of implementation is eight.

Input graph is constructed by random, where the probability of edge is given. In the experiments, the probability set is $\{0.2, 0.02, 0.002, 0.0002\}$, denoted as p. The number of vertices is spread across $[10000, 500000]$. The weight of edge is decided by random under the uniform distribution under $[1, 2000]$.

The experimental environment is as follows: priority queue is STL library; radix-heap is open source library [8]; CPU is Intel Core 3.2 GHz; Memory size is 32 GB; OS is macOS Mojave.

TABLE I, II, III, and IV show the experimental results of $p = 0.2$, $p = 0.02$, $p = 0.002$, and $p = 0.0002$, respectively, where each line corresponds to the run-time and the ratio to the run-time of Dijkstra's algorithm with radix-heap and # v.s line corresponds to the number of vertices. Each element corresponds to the average of the run-time among five trials where its unit is msec. For $p = 0.2$, the result with 200000 vertices or more is not presented since the experiment cannot be finished due to the lack of memory. On the other hand, the lack of data with 10000 vertices for $p = 0.0002$ since too short run-time.

From the experimental results, several observations are clear.

- Lazy update improves the run-time.
- As a comparison between the ordinary priority queue and the radix-heap, the method with radix-heap is faster than that with the ordinary priority queue.
- For the limit value, the case 1, that is, (the minimum tentative distance) + (the minimum edge distance), is faster.
- For the implementation of LR, a simultaneous extraction achieves the better solutions.

For $p = 0.0002$, Dijkstra's algorithm with lazy update and radix-heap which employs a simultaneous extraction until the sum of the minimum tentative distance and the minimum edge distance (LRS1) is 13% faster than Dijkstra's algorithm with radix-heap and 50% faster than Dijkstra's algorithm with ordinary priority queue. Thus, we confirm that the proposed method is efficient.

V. Concluding Remarks

In this paper, we discussed the acceleration of Dijkstra's algorithm by lazy update. In the discussion, we confirmed the efficiency of lazy update even with ordinary priority queue or radix-heap. Especially, we enhanced the radix-heap by the utilization of data-structure. Experimental results show that 50% run-time improvement from the ordinary implementation or 13% run-time improvement from the radix-heap based algorithm. Thus, the proposed method is efficient.

For the future works, we need to achieve more acceleration. Especially, the consideration of parallel implementation must be done. The dynamic consideration of the minimum edge distance may also be promising.

References

[1] GoogleTechTalks,"Fast Route Planning," `https://www.youtube.com/watch?v=-0ErpE8tQbw`.

[2] D. Z. Chen, "Developing algorithms and software for geometric path planning problems," *ACM Computing Surveys*, Vol. 28, Article No. 18, 1996.

[3] I. Abraham, A. Fiat, A. Goldberg, and R. Werneck, "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms," *ACM-SIAM Symposium on Discrete Algorithms (SODA10))*, pp.782–793, 2010.

[4] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, "Handbook of Algorithms for Physical Design Automation", *CRC Press*, 2009.

[5] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, 1, pp.269–271, 1959.

[6] "Dijkstra's algorithm,", `https://en.wikipedia.org/wiki/Dijkstra\%27s_algorithm`.

[7] T. Takahashi and Y. Takashima, "Fast approximate algorithm for the single source shortest path with Lazy update," *NGCAS 2018*, pp.94–97, 2018.

[8] T. Akiba, "Radix-Heap," `https://github.com/iwiwi/radix-heap`.

TABLE I
EXPERIMENTAL RESULT OF $p = 0.2$

# v.s	H	ratio	R	LH0	ratio	LH1	ratio	LRN0	ratio	LRN1	ratio	LRS0	ratio	LRS1	ratio
10000	65.0	1.031	63.0	62.6	0.994	61.0	0.968	60.8	0.965	60.0	0.952	60.4	0.959	58.6	0.930
20000	284.2	0.934	304.2	273.0	0.897	265.0	0.871	262.6	0.863	267.0	0.878	265.6	0.873	272.0	0.894
30000	627.4	0.935	671.2	563.2	0.839	557.2	0.830	562.0	0.837	547.0	0.815	547.8	0.816	536.0	0.799
40000	1035.4	0.918	1127.6	922.0	0.818	924.6	0.820	932.8	0.827	944.6	0.838	934.4	0.829	916.2	0.813
50000	1594.4	0.899	1773.4	1427.2	0.805	1357.0	0.765	1477.2	0.833	1390.8	0.784	1462.0	0.824	1372.2	0.774
60000	2244.8	0.896	2504.0	2086.8	0.833	2134.6	0.852	2060.2	0.823	2121.6	0.847	2056.2	0.821	2074.0	0.828
70000	3073.4	0.889	3456.4	2474.6	0.716	2272.8	0.658	2466.6	0.714	2334.8	0.676	2490.2	0.720	2324.2	0.672
80000	4102.6	0.888	4620.8	3626.2	0.785	2823.4	0.611	3631.6	0.786	2837.2	0.614	3545.4	0.767	2769.6	0.599
90000	5280.0	0.901	5859.6	4732.0	0.808	3792.0	0.647	4774.0	0.815	3853.0	0.658	4729.0	0.807	3783.6	0.646
100000	6065.2	0.829	7315.8	5302.2	0.725	5172.6	0.707	5381.8	0.736	5308.0	0.726	5361.8	0.733	5304.2	0.725
Aver.	—	0.913	—	—	0.839	—	0.779	—	0.828	—	0.787	—	0.817	—	0.771

TABLE II
EXPERIMENTAL RESULT OF $p = 0.02$

# v.s	H	ratio	R	LH0	ratio	LH1	ratio	LRN0	ratio	LRN1	ratio	LRS0	ratio	LRS1	ratio
10000	12.8	1.306	9.8	12.2	1.245	11.6	1.184	9.6	0.980	9.8	1.000	10.0	1.020	9.4	0.959
20000	42.4	1.225	34.6	41.2	1.191	41.2	1.191	34.6	1.000	33.8	0.977	36.6	1.058	41.4	1.197
30000	91.4	1.163	78.6	84.6	1.076	84.8	1.079	76.2	0.969	76.2	0.969	76.6	0.975	74.6	0.949
40000	153.4	1.136	135.0	146.6	1.086	143.8	1.065	131.6	0.975	130.8	0.969	130.0	0.963	129.6	0.960
50000	229.4	1.120	204.8	225.4	1.101	222.0	1.084	204.0	0.996	218.0	1.064	203.4	0.993	200.2	0.977
60000	316.6	1.091	290.2	308.6	1.063	307.8	1.061	284.8	0.981	284.0	0.979	287.4	0.990	283.4	0.977
70000	429.8	1.094	392.8	409.6	1.043	402.4	1.024	383.8	0.977	380.0	0.967	384.4	0.979	385.0	0.980
80000	530.4	1.056	502.2	528.8	1.053	517.2	1.030	500.4	0.996	485.2	0.966	494.0	0.984	480.6	0.957
90000	669.4	1.002	668.2	678.4	1.015	669.6	1.002	662.8	0.992	624.0	0.934	617.8	0.925	611.8	0.916
100000	786.0	1.042	754.0	764.0	1.013	752.7	0.998	731.0	0.969	723.0	0.959	727.3	0.965	716.0	0.950
200000	3005.0	1.012	2968.3	2748.0	0.926	2753.0	0.927	2729.7	0.920	2723.7	0.918	2698.3	0.909	2695.0	0.908
300000	6916.3	1.032	6699.0	5887.7	0.879	5884.7	0.878	5910.3	0.882	5909.3	0.882	5867.3	0.876	5866.3	0.876
400000	37792.3	0.852	44376.0	47908.3	1.080	49339.7	1.112	45538.3	1.026	50365.3	1.135	42476.3	0.957	49279.3	1.110
Aver.	—	1.087	—	—	1.059	—	1.049	—	0.974	—	0.978	—	0.969	—	0.978

TABLE III
EXPERIMENTAL RESULT OF $p = 0.002$

# v.s	H	ratio	R	LH0	ratio	LH1	ratio	LRN0	ratio	LRN1	ratio	LRS0	ratio	LRS1	ratio
10000	4.6	1.533	3.0	4.0	1.333	4.0	1.333	2.2	0.733	2.2	0.733	2.0	0.667	2.2	0.733
20000	13.0	1.413	9.2	12.4	1.348	12.2	1.326	8.0	0.870	8.2	0.891	7.6	0.826	7.4	0.804
30000	24.2	1.513	16.0	24.2	1.513	24.2	1.513	16.2	1.013	15.8	0.988	15.8	0.988	15.2	0.950
40000	37.2	1.420	26.2	37.0	1.412	36.0	1.374	25.0	0.954	25.0	0.954	24.2	0.924	24.0	0.916
50000	53.0	1.432	37.0	53.4	1.443	52.2	1.410	36.4	0.984	35.8	0.968	35.0	0.946	35.0	0.946
60000	66.8	1.386	48.2	65.0	1.349	72.4	1.502	47.2	0.979	47.0	0.975	47.2	0.979	47.2	0.979
70000	87.2	1.402	62.2	86.2	1.386	87.4	1.405	62.4	1.003	61.2	0.984	62.0	0.997	61.2	0.984
80000	109.4	1.347	81.2	110.2	1.357	110.2	1.357	79.4	0.978	79.0	0.973	79.2	0.975	77.8	0.958
90000	131.6	1.285	102.4	131.8	1.287	130.4	1.273	99.2	0.969	97.6	0.953	98.0	0.957	97.2	0.949
100000	171.0	1.289	132.7	162.0	1.221	162.7	1.226	121.3	0.915	120.0	0.905	124.0	0.935	121.3	0.915
200000	520.3	1.188	438.0	518.7	1.184	520.7	1.189	429.0	0.979	426.3	0.973	422.0	0.963	423.7	0.967
300000	1107.0	1.194	927.0	1062.7	1.146	1051.0	1.134	906.3	0.978	922.0	0.995	921.0	0.994	914.0	0.986
400000	1855.0	1.156	1604.0	1785.7	1.113	1769.7	1.103	1580.7	0.985	1563.3	0.975	1554.7	0.969	1547.3	0.965
500000	2849.0	1.133	2515.7	2718.7	1.081	2722.3	1.082	2467.0	0.981	2452.3	0.975	2445.3	0.972	2441.0	0.970
Aver.	—	1.335	—	—	1.298	—	1.302	—	0.951	—	0.946	—	0.935	—	0.930

TABLE IV
EXPERIMENTAL RESULT OF $p = 0.0002$

# v.s	H	ratio	R	LH0	ratio	LH1	ratio	LRN0	ratio	LRN1	ratio	LRS0	ratio	LRS1	ratio
20000	5.2	2.363	2.2	4.4	2.000	3.4	1.545	2.0	0.909	2.0	0.909	2.0	0.909	2.0	0.909
30000	10.2	1.889	5.4	8.8	1.630	7.8	1.444	4.0	0.741	3.8	0.704	3.2	0.593	3.0	0.556
40000	17.4	1.740	10.0	14.4	1.440	13.6	1.360	7.6	0.760	7.2	0.720	6.6	0.660	6.8	0.680
50000	23.6	1.761	13.4	21.0	1.567	20.6	1.537	13.2	0.985	13.2	0.985	11.6	0.866	11.4	0.851
60000	31.8	1.710	18.6	29.6	1.591	28.0	1.505	17.4	0.935	16.6	0.892	15.8	0.849	15.0	0.806
70000	40.4	1.656	24.4	38.8	1.590	37.8	1.549	23.0	0.943	23.0	0.943	21.4	0.877	21.0	0.861
80000	50.2	1.651	30.4	47.8	1.572	47.4	1.559	28.2	0.928	29.2	0.961	27.2	0.895	26.8	0.882
90000	59.0	1.715	34.4	58.2	1.692	56.0	1.628	34.4	1.000	34.8	1.012	33.0	0.959	32.6	0.948
100000	65.3	1.633	40.0	66.0	1.650	64.3	1.608	38.7	0.967	39.7	0.992	37.3	0.933	38.0	0.950
200000	178.7	1.629	109.7	178.3	1.626	185.7	1.693	108.3	0.988	105.7	0.964	109.7	1.000	107.7	0.982
300000	335.7	1.645	204.0	340.0	1.667	338.7	1.660	201.0	0.985	202.0	0.990	195.3	0.958	196.0	0.961
400000	510.3	1.554	328.3	515.3	1.570	515.3	1.570	316.	0.962	317.7	0.968	310.3	0.945	307.3	0.936
500000	706.3	1.530	461.7	716.7	1.552	718.7	1.557	450.7	0.976	451.3	0.978	452.0	0.979	451.3	0.978
Aver.	—	1.729	—	—	1.627	—	1.555	—	0.929	—	0.924	—	0.879	—	0.869