Computer and Information Science; Vol. 14, No. 1; 2021
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

# An O(nlogn/logw) Time Algorithm for Ridesharing

Yijie Han[1], & Chen Sun[1]

[1] School of Computing and Engineering, University of Missouri at Kansas City, USA

Correspondence: Yijie Han, School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, USA.

Received: July 29, 2020 Accepted: December 28, 2020 Online Published: January 8, 2021

doi:10.5539/cis.v14n1p8 URL: https://doi.org/10.5539/cis.v14n1p8

**Abstract**

In the ridesharing problem different people share private vehicles because they have similar itineraries. The objective of solving the ridesharing problem is to minimize the number of drivers needed to carry all load to the destination. The general case of ridesharing problem is NP-complete. For the special case where the network is a chain and the destination is the leftmost vertex of the chain, we present an O(nlogn/logw) time algorithm for the ridesharing problem, where w is the word length used in the algorithm and is at least logn. Previous achieved algorithm for this case requires O(nlogn) time.

**Keywords:** data engineering, efficient algorithms, ridesharing, time complexity

## 1. Introduction

A road network is expressed by a (undirected) graph G connecting a set V(G) of vertices and a set E(G) of edges. Each edge (u, v), where u, v are vertices represents a road between u and v. G is weighted if each edge is assigned a weight (distance of the road). When G is unweighted we assume that each edge has weight 1. A path is a sequence of edges $e_1, e_2, \ldots, e_k$, where $e_i=(v_{i-1}, v_i) \in (G)$, $1 \le i \le k$ and no vertex is repeated in the sequence (i.e. there is no loops), where k is for $v_k$, the source. The length of the path P is the sum of the weights of its edges.

In our case we consider the situation that the road network is one line $v_0, v_1, .., v_n$ and the left most vertex $v_0$ is the destination for all trips. A trip t is from $v_i$ for some i to $v_0$, i.e. $v_i, v_{i-1}, \ldots, v_0$.

A trip t from $v_i$ to $v_0$ has load load (i) (which is a nonnegative number) at $v_i$ and capacity capacity(i) (which is a nonnegative number) at $v_i$. capacity (i) is the maximum load the driver can carry from $v_i$ to $v_0$. Thus load(i)≤capacity(i). free(i)=capacity(i)-load(i) is called the free load for trip t. When free(i) > 0 then on the path from $v_i$ to $v_0$ the driver of trip t can carry additional free(i) load from other trips at $v_{i-1}, v_{i-2}, \ldots, v_1$. When all the load at $v_i$ is carried by other trips with sources $v_k$, k>i, then the trip from $v_i$ to $v_0$ can be canceled. This is to say, when a driver at source $v_j$, j>i, pass by $v_i$ and is not fully loaded, he can carry some of the load at vi. If all load at $v_i$ are carried by such drivers, then the trip from $v_i$ to $v_0$ can be canceled. The objective is to remove as many trips as possible and to minimize the number of trips and thus keep the number of drivers needed at minimum.

The minimization problems in the general ridesharing problem (Gu, Q.-P., Liang, J. L. & Zhang, G., 2017) are complex and NP-hard because each trip may have many parameters. They can be solved as an Integer Programming (IP) or Mixed Integer Programming (MIP) problem and solves the IP or MIP problem by an exact method or heuristics (Alonso-Mora, J., Samaranayake, S., Wallar, A. , Frazzoli, E., D. Rus, D. 2017) (Baldacci, R., Maniezzo, V., Mingozzi, A. 2004) (Herbawi, W., Weber, M. 2012). The case considered in this paper is a situation where polynomial time algorithm can be obtained.

This version of the ridesharing problem has been studied by Gu, Liang and Zhang (Gu, Q.-P., Liang, J. L. & Zhang, G., 2019). (Gu, Q.-P., Liang, J. L. & Zhang, G., 2018). (Gu, Q.-P., Liang, J. L. & Zhang, G., 2017). In (Gu, Q.-P., Liang, J. L. & Zhang, G., 2017) they obtained $O(n^2)$ time algorithm for the problem but in (Gu, Q.-P., Liang, J. L. & Zhang, G.,, 2019) they achieved O(nlogn) time.

As an example, the following Figure 1 shows the initial situation of a ridesharing problem.

Figure 1. Initial situation of a ridesharing problem

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 1 | 3 | 5 | 1 | 1 |
| load : | | 7 | 2 | 10 | 5 | 5 | 4 | 2 |

For example, in Figure 1 shows that at $v_2$ the load is 2 and the free is 4. This means when the driver at $v_2$ can carry the load 2 and also can add additional load 4 when he drives to the destination $v_0$. If he pick additional load 4 at $v_1$, then at $v_1$ the load becomes 3 and the free becomes 2+4=6.

In this paper we show an O(nlogn/logw) algorithm for the ridesharing problem, where w is the word length, i.e. the number of bits used in a word. w is at least logn and therefore our algorithm has complexity no worse than O(nlogn/loglogn). Thus we improve the result achieved by Gu.et al.

**2. Preliminary**

As an example, we show a case in which the load at some sources can be removed:

Figure 1. Initial situation of a ridesharing problem

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 1 | 3 | 5 | 1 | 1 |
| load : | | 7 | 2 | 10 | 5 | 5 | 4 | 2 |

In this system, we can let the trip from $v_3$ carry load 1 from the load at $v_2$ and the trip from $v_4$ carry additional load 1 from $v_2$ and thus the load at $v_2$ can be carried by the trips starting at $v_3$ and $v_4$. The modified situation is shown here.

Figure 2. As modified in Figure 1

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 6 | 0 | 2 | 5 | 1 | 1 |
| load : | | 7 | 0 | 11 | 6 | 5 | 4 | 2 |

Thus the trip from $v_2$ can be removed. Now the load 7 at $v_1$ can be carried by trips starting at $v_3$ and $v_5$. The modified situation is shown here.

Figure 3. As modified in Figure 2

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 9 | 6 | 0 | 0 | 0 | 1 | 1 |
| load : | | 0 | 0 | 11 | 8 | 10 | 5 | 2 |

Thus we reduced 7 trips in the initial input to 5 trips.

**3. How This Works in Previous Papers**

In (Gu, Q.-P., Liang, J. L. & Zhang, G., 2018) it shows that there is an algorithm to reduce the number of trips to minimum in $O(n^3)$ time. This algorithm was improved to $O(n^2)$ time in (Gu, Q.-P., Liang, J. L. & Zhang, G., 2017). Here we describe the algorithm in (Gu, Q.-P., Liang, J. L. & Zhang, G., 2018) and explain how it works as some principles used in (Gu, Q.-P., Liang, J. L. & Zhang, G., 2018) are also used in our algorithm.

The algorithm in (Gu, Q.-P., Liang, J. L. & Zhang, G., 2018) computes GAP(i, j). GAP(i, j) is the remaining load after redistributing load at vertex i : load(i), to vertices k, i < k < j, provided that load(k) >0.

$GAP(i, j)=| S_j |-\sum_{a \in St, j<a<i} free(a)$

Find out the minimum GAP.

In our previous example:

Figure 1. Initial situation of a ridesharing problem

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 1 | 3 | 5 | 1 | 1 |
| load : | | 7 | 2 | 10 | 5 | 5 | 4 | 2 |

GAP(2, 3)=2, this is because when we redistribute load 2 at $v_2$ to $v_3$ and when we arrive at $v_3$ the load coming from $v_2$ is 2. And GAP(2, 4)=1, this is because when we redistribute load 2 at $v_2$ all the way to $v_4$ we can distribute load 1 from 2 to $v_3$ as $v_3$ has free 1, thus when we arraive at $v_4$ we have load 1 remaining.

In our previous example

Figure 2. As modified in Figure 1

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 6 | 0 | 2 | 5 | 1 | 1 |
| load : | | 7 | 0 | 11 | 6 | 5 | 4 | 2 |

GAP(1,3)=7 (note that load(2)=0), GAP(1,4)=7, GAP(1, 5)=5, GAP(1, 6)=0, this is because load 7 at $v_1$ can be redistributed to $v_4$ (redistribute load 2 as $v_4$ has free 2) and redistribute to $v_5$ (redistribute load 5 as $v_5$ has free 5).

The algorithm in (Gu, Q.-P., Liang, J. L. & Zhang, G., 2019) has a loop iterating through $v_1$, $v_2$, …, $v_n$. When working on $v_j$, the minimum of GAP(i, j), $1 \le i < j$, is found. Let $G(i_j, j)$ be the minimum, if $G(i_j, j) \le$ free(j) then load($i_j$) will be redistributed to $v_{i_j+1}$, $v_{i_j+2}$, …, $v_j$ to make free($i_j$+1), free($i_j$+2), …, free(j-1) to 0's. If $G(i_j, j)$>free(j) no redistribution of load will happen as redistribution, if taken, cannot remove any driver. Thus if $G(i_j, j) >$ free(j) then the loop will iterate to j+1.

Because GAP(i, j) takes O(j-i) time and therefore the computation of GAP(i, j), $1 \le i < j$, takes $O(j^2)$ time. Because the loop has n iterations and therefore the algorithm has $O(n^3)$ time.

**4. Our Algorithm**

We speed up the algorithm using the dynamic integer set and redesigned the algorithm so that dynamic integer set operations can be applied. Our algorithm time complexity is O(nlogn/logw), where $w \ge$ logn. We use dynamic integer sets (Phtrƅcu M. & Thorup M., 2014, 166-175) that support insert, delete, min operations among other operations in O(logn/logw) time. The reason we can use dynamic integer sets because load and capacity are integers (they are seats on the vehicle). We will name such a dynamic integer set as set H.

We intend to compute

$S_0=\sum_{i=1}^{n} free(i)$=2+4+1+3+5+1+1=17

$S_1=S_0$-free(1)=17-2=15, $S_2=S_1$-free(2)=15-4=11, $S_3=S_2$-free(3)=11-1=10

$S_4=S_3$-free(4)=10-3=7, $S_5=S_4$-free(5)=7-5=2, $S_6=S_5$-free(6)=2-1=1

And use load(i)-$S_i$ as a key. After put keys in a dynamic integer set and find the minimum key we can identify GAP(i, j) which is minimum.

In our case

Figure 1. Initial situation of a ridesharing problem

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 1 | 3 | 5 | 1 | 1 |
| load : | | 7 | 2 | 10 | 5 | 5 | 4 | 2 |

Our algorithm precedes as follows:

Precomputation: Compute $S_0$ =17 in our case) in O(n) time.

Step 1 : Works on $v_1$. Compute $S_1=S_0$-free(1) (=17-2=15 in our case). Enter $key_1$=load(1)-$S_1$ (=7-15=-8 in our case) into set H.

Step 2 : Works on $v_2$. Compute $S_2=S_1$-free(2) (=15-4=11 in our case). Find min in set H which is $key_1$. Compare load(1) (1 here because $key_1$) with $S_1$-$S_2$ (1 because of $key_1$ and 2 because of $v_2$) which is free(2). In our case load(1)=7 > 4=$S_1$-$S_2$ therefore load(1) cannot be redistributed. Thus we enter $key_2$=load(2)-$S_2$ (=2-11=-9 in our case) into set H.

Step 3 : Works on $v_3$. Compute $S_3=S_2$-free(3) (=11-1=10 in our case). Find min in set H which is $key_2$. Compare load(2) (2 here because $key_2$) with $S_2$-$S_3$ (2 because of $key_2$ and 3 because of $v_3$) which is free(3). In our case load(2)=2 > 1=$S_2$-$S_3$ therefore load(2) cannot be redistributed. Thus we enter $key_3$=load(3)-S3 (=10-10=0 in our

case) into set H.

Step 4 : Works on $v_4$. Compute $S_4=S_3$-free(4) (=10-3=7 in our case). Find min in set H which is $key_2$. Compare load(2) (2 here because $key_2$) with $S_2$-$S_4$ (2 because of $key_2$ and 4 because of $v_4$) which is free(3)+free(4). In our case load(2)=2 ≤ 4=$S_2$-$S_4$ therefore load(2) can be redistributed. After redistribution the situation becomes :

Figure 2. As modified in Figure 1

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 0 | 2 | 5 | 1 | 1 |
| load : | | 7 | 0 | 11 | 6 | 5 | 4 | 2 |

Note up so far it works fine as when we are working on $v_j$ and find min $k_i$ in set H then GAP(i, j) is the minimum among all GAP(k, j) for k<j. However, it does not take us O($j^2$) time to find the minimum GAP, it takes us only O(logn/logw) time to find the min in set H to find the minimum GAP.

However, after redistribution, the $S_i$ values have been changed and thus it looks like that we have to recompute the key values for the keys already in set H. What we do instead is:

Because load(2)=0, it can be removed. We used 1 from free(3), 1 free(4) to carry the load at $v_2$.

Since now free(3) is decremented by 1 and free(4) is decremented by 1, we should increase key1 by 2. However, to change the value for the keys already in the dynamic integer set will make the O(nlogn/logw) time for our algorithm unachievable. What we do is to decrement 2 for all the keys entered starting form $v_4$. That is for every key starting from $key_4$, we will subtract 2 from the key value before entering it into set H. Here we call -load(2)=-2 as the adjusting value.

Redistributed:

Figure 2. As modified in Figure 1

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 2 | 4 | 0 | 2 | 5 | 1 | 1 |
| load : | | 7 | 0 | 11 | 6 | 5 | 4 | 2 |

$v_2$ is removed. Since free(4) has been decremented by 1, we need to change value of key3. We continue working on Step 4. Because $v_2$ is removed we need change adjusting value=adjusting value -free(2)=-2-4=-6

Step 4 (continued): All vertices v with free(v) set to 0 needs to be worked on. Therefore we work on $v_3$. Remove $key_3$ from set H. $key_3=key_3$+load(2) -free(3) (before load(2) and free(3) were set to 0, load(2) because load(2) is redistributed, free(3) because load(2)-free(3) is the value taken out from the free's starting from $v_4$) +adjusting value = 0+2-1-6= -7. Insert $key_3$ into set H.

Find min in set H again, which is $key_1$= -8. $S_1$-$S_4$+adjusting value=15-7-6=2 < load(1)=7, thus no redistribution can happen. Thus we put $key_4$=load(4)-$S_4$+adjecting value=6-7-6= -7 into set H.

Step 5 : Working on $v_5$. Computing $S_5=S_4$-free(5) = 7-5=2. Find min in set H which is $key_1$=-8. $S_1$-$S_5$+adjusting value=15-2-6=7 ≥ 7=load(1). Thus redistribute load(1). After redistribution the situation becomes:

Figure 4. As modified in Figure 2 (use our algorithm redistribution the situation)

| source, v : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| free : | | 9 | | 0 | 0 | 0 | 1 | 1 |
| load : | | 0 | | 11 | 8 | 10 | 4 | 2 |

In the remaining steps redistribution cannot happen. Thus our algorithm ends up with 5 drivers.

In general, suppose the first time we redistribute load it is to redistribute load(i) to $v_{i+1}$, $v_{i+2}$,…, $v_{i+j}$. Then free(i+1), free(i+2),… free(i+j-1) all become 0's. load(i)=0 and $v_i$ can be removed. The adjusting value= -load(i)-capacity(i) (capacity(i)=load(i)+free(i)). We will remove $key_{i+1}$, $key_{i+2}$, …, $key_{i+j-1}$ from set H and build another dynamic integer set $H_i$ with capacity(i+1), capacity(i+2), …, capacity(i+j-1) inserted in to set $H_i$. Here capacity(i+k) is equal to load(i+k) as free(i+k) = 0, 1 ≤ k < j. In our algorithm the new key(i+k)=capacity(i+k)-$S_{i+j-1}$ + (load(i) (before redistribution) –($S_i$-$S_{i+j-1}$))(this is the free value amount at free(i+j) that is used for redistributing load(i))+adjusting value, 1 ≤ k < j. In this quantity the only value relevant to k is capacity(i+k). We will call $S_{i+j-1}$ + (load(i) (before redistribution) –($S_i$-$S_{i+j-1}$))+adjusting value as the adjusting

value for $H_i$. Thus we build $H_i$ and insert the min key min(k) in $H_i$ into H. If min(k) in H is deleted then we delete min(k) from $H_i$ and find the new min(k) in $H_i$ and insert it into H. We also have to build a set $T_i$ and put vertices $v_i$, $v_{i+1}$, …, $v_{i+j-1}$ into from $T_i$ in O(j-i) time The use $T_i$ is to let every vertex $v_{i-1}$, $v_i$, …, $v_{i+j-1}$ to find $v_{i+j}$ quickly. Here every vertex $v_{i-1}$, $v_i$, …, $v_{i+j-1}$ can find the set $T_i$ which is pointing to $v_{i+j}$. Because if a load(k), $i-1 \leq k \leq i+j-1$, is to be redistributed, we have to jump through $v_{k+1}$, $v_{k+2}$, $v_{i+j-1}$ without visiting each of them (to keep the O(logn/logw) time for each vertex), tree $T_i$ will allow us find $v_{i+j}$ from $v_k$ in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function, because we will use Union-Find (T.H. Corman, C.E. Leiserson, R.L. Rivest & C. Stein, 2009) to implement it.

We then start working on $v_{i+j}$, and we compute $key_{i+j}$=load(i+j)-$S_{i+j}$+adjusting value.

Suppose we are going to do another redistribution from $v_l$ to $v_m$. Consider the following cases :

1. $l >= i+j$. This is the most simply case. Because key values for keyk, $k<i+j$, need not be changed and we need only add adjusting values to later keys. Note if $l > i+j$ then we will build another dynamic integer set $H_l$ for the nodes $v_l$ to $v_{m-1}$. $v_l$ will be removed. The new adjusting value=adjusting value-load(l)-capacity(l). The adjusting value for $H_l$ is $S_m$ + (load(l) (before redistribution) –($S_l$-$S_{m-1}$))+adjusting value. If $l > i+j$ we will build set $T_l$ in O(m-l) time. If $l=i+j$ we will build set $T_l$ in O(m-l) time and then union $T_i$ and $T_l$ into one set with the Union-Find algorithm (T.H. Corman, C.E. Leiserson, R.L. Rivest & C. Stein., 2009) and name it $T_i$.

2. $l < i$. In this case we will redistribute load(l) to $v_{l+1}$, $v_{l+2}$, …, $v_{i-1}$, then jump over from $v_i$ to $v_{i+j-1}$ and then start redistribute at $v_{i+j}$, $v_{i+j+1}$, …, $v_m$. capacity(l+1), capacity(l+2), …, capacity(i-1), capacity(i+j), capacity(i+j+1),…, capacity(m-1) will be entered as keys into $H_i$.The original min(key) in $H_i$ that was entered into H will be withdrawn from H. We then rename $H_i$ as $H_l$ and find min(key) in $H_l$ and insert it into H. $v_l$ will be removed. The new adjusting value is updated as adjusting value – load(l)-capacity(l). The adjusting value for $H_l$ is $S_m$ + (load(l) (before redistribution) – ($S_l$-$S_i$+ $S_{i+j-1}$-$S_{m-1}$)) + adjusting value. We will build a set for $v_l$, $v_{l+1}$, …, $v_{i-1}$ in O(i-l) time, and set for $v_{i+j}$, $v_{i+j+1}$,…, $v_{m-1}$ in O(m-i-j) time. Then union these two sets with $T_i$ and then rename $T_i$ to $T_l$.

3. $i \leq l < i+j$. In this case redistribute load(l) to $v_{i+j}$, $v_{i+j+1}$, …, $v_m$. capacity(i+j), capacity(i+j+1), …, capacity(m-1) will be entered into set $H_i$. $v_l$ will be removed. New adjusting value=adjusting value – load(l) -capacity(l). The new adjusting value for $H_i$ is $S_m$ + y(load(l) (before redistribution) –($S_{i+j-1}$-$S_{m-1}$))+adjusting value. We build a set for $v_{i+j}$, $v_{i+j+1}$, …, $v_{m-1}$ in O(m-i-j) time and then union this set with $T_i$.

Thus in our algorithm, each vertex $v_i$ and load(i) is processed at most twice. Once is load(j) redistribute to free(i) for $j< i$. We count vi as processed for this time only if free(i) changed from nonzero to zero. If part of load(i) is redistributed to free(i) but free(i) did not become 0 then $v_i$ is the last vertex for the distribution. In this case we attribute the time for processing $v_i$ to the time for $v_j$. Another time $v_i$ and load(i) is processed is when load(i) is redistributed and therefore $v_i$ is removed. The remaining time for $v_i$ is insert $key_i$ into dynamix integer set and delete $key_i$ from dynamic integer set. It is known that these operation has time O(logn/logw) for processing each vertex.

Because we spend O(logn/logw) time per vertex and therefore the time complexity of our algorithm is O(nlogn/logw).

**Main Theorem:** There is an O(nlogn/logw) time algorithm that finds the minimum number of drivers for the ridesharing problem.

## 5. Conclusion

We proposed an O(nlogn/logw) time algorithm to minimize the number of drivers for the ridesharing problem. It is also worth developing algorithms for rider cases and exploring the algorithmic complexity of other simplified variants of this problem.

Note that the load and capacity in the algorithm are nonnegative integers and therefore we can use the dynamic integer set for them. If load and capacity are real values then a different algorithm has to be designed.

## References

Alonso-Mora, J., Samaranayake, S., Wallar, A., Frazzoli, E., & D. Rus, D. (2017) On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proc. Natl. Acad. Sci., 114*(3), 462-467. https://doi.org/10.1073/pnas.1611675114

Baldacci, R., Maniezzo, V., & Mingozzi, A. (2004). An exact method for the car pooling problem based on Lagrangean column generation. *Oper. Res., 52*(3), 422-439. https://doi.org/10.1287/opre.1030.0106

Corman, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. Third Edition, *The MIT Press*.

Gu, Q. P., Liang, J. L., & Zhang, G. (2019). Efficient algorithms for ridesharing if personal vehicles. *Theoretical Computer Science, 788,* 79-94. https://doi.org/10.1016/j.tcs.2019.05.027

Gu, Q. P., Liang, J. L., & Zhang, G. (2018). Algorithmic analysis for ridesharing of personal vehicles. *Theoretical Computer Science, 749,* 36-46. https://doi.org/10.1016/j.tcs.2017.08.019

Gu, Q. P., Liang, J. L., & Zhang, G. (2017). Efficient algorithms for ridesharing of personal vehicles. *Proceedings of COCOA'2017, LNCS 10627*, 340-354. https://doi.org/10.1007/978-3-319-71150-8_29

Herbawi, W., & Weber, M. (2012). The ridematching problem with time windows in dynamic ridesharing: a model and a genetic algorithm, in: Proceedings of ACM Genetic and Evolutionary Computation Conference (GECCO), pp. 1–8. https://doi.org/10.1109/CEC.2012.6253001

Phtrbcu, M., & Thorup, M. (2014). Dynamic integer sets with optimal rank, select, and predecessor search. *Proc. 2014 IEEE Foundations of Computer Science*. 166-175.