An Efficient Hardware Architecture of the A-star Algorithm for the Shortest Path Search Engine

Woo-Jin Seo, Seung-Ho Ok
School of Electrical Eng. & Computer Science
Kyungpook National University
Daegu, Korea
{swj82, wintiger}@ee.knu.ac.kr

Jin-Ho Ahn
Dept. of Electronic Engineering
Hoseo University
Chungcheongnam-do, Korea
jhahn@hoseo.edu

Sungho Kang
Dept. of Electrical and Electronic Engineering
Yonsei University
Seoul, Korea
shkang@yonsei.ac.kr

Byungin Moon
School of Electrical Eng. & Computer Science
Kyungpook National University
Daegu, Korea
bihmoon@knu.ac.kr

Abstract—**There are several shortest-path search algorithms such as A-star, D-star and Dijkstra. These algorithms are widely used in automotive vehicles and mobile navigation systems. As the number of nodes is increased considerably, the shortest-path algorithms implemented in software produce heavily computational overhead. In this paper, in order to avoid computational overhead, we propose a hardware model of the A-star algorithm for the shortest-path search engine. Especially, we propose shift register based on efficient hardware model and show simulation results in comparison with previous works.**

Keywords - shortest-path search algorithm; A-star algorithm; shift register; priority queue; sorting;

I. INTRODUCTION

Transportation infrastructure is a highly complex network and there are huge traffics. For this reason, the telematics has been prospered in the automotive industry. More recently, it has been specifically applied to Global Positioning System(GPS) technology integrated with computers and mobile communications technology in automotive navigation systems.

One of the most important parts of the automotive navigation system is a shortest path search algorithm. In general, as the number of nodes is consistently increased, the shortest-path search algorithms implemented in software produce heavily computational overhead. Thus, the overall performance efficiency of the automotive navigation system could be decreased. Moreover, this performance degradation finally influence to the users of navigation because of increased searching time of the shortest path and disturbance of the telematics operation.

As mentioned above, a shortest path algorithm implemented in software produces huge overhead. But if the shortest-path algorithm is implemented with reasonable hardware, navigation system can efficiently avoid heavy overhead. Consequently, the overall performance of the navigation system can be increased.

The most important algorithm of the shortest path problem is the A* algorithm which finds single pair shortest-path using heuristic function to speed up the search. For this reason, we aim to design hardware model of the A* algorithm.

The hardware implementations of the A* algorithm have been studied in the previous researches. Z. K. Baker and M. Gokhale investigate a small bubble sort core to produce the extract-min function [2]. But its time complexity is O(N). Also pipelined heap can be adopted [5], but time complexity is O(log N).

Compared to the previous work, proposed architecture is based on a sorted shift register like a priority queue. And particularly, time complexity is O(1). In Section II, we explain the A* algorithm and the point of improvement. The details of the proposed architecture be described in Section III and Section IV shows a simulation results. Finally, we conclude our work in Section V.

II. RELATED WORK

There are many shortest-path search algorithms, Dijkstra, Bellman-Ford and A* algorithm [1, 6]. The one of them, A* algorithm, is usually used in the industry of transportation. The reason why A* algorithm is usually used is its flexibility that it can reduce the computational time. Fig. 1 represents pseudo code of A* algorithm.

```
▸    A* (start, goal)
1.   Closed set = the empty set
2.   Open set = includes start node
3.   G[start] = 0, H[start] = H_calc[start, goal]
4.   F[start] = H[start]
5.   While Open set ≠ ∅
6.       do CurNode ← EXTRACT-MIN- F(Open set)
7.       if ( CurNode  == goal ), then return BestPath
8.       For each Neighbor Node N of CurNode
9.           If ( N is in Closed set ), then Nothing
10.          else if ( N is in Open set ),
11.              calculate N's G, H, F
12.              If ( G[N on the Open set] > calculated G[N] )
13.                  RELAX(N, Neighbor in Open set, w)
14.                  N's parent=CurNode & add N to Open set
15.          else, then calculate N's G, H, F
16.                  N's parent = CurNode & add N to Open
```

Figure 1. A* Algorithm pseudo code

Initial conditions are represented from line 1 through line 4. The open set is defined as a set of already known nodes.

Nodes in the open set have been neighbor nodes of certain current nodes and they can be current nodes. And the closed set is defined as a set of nodes that have been current nodes.

In order to find the shortest path, the value of G, H and F are used. The value of the G is the actual shortest distance from the initial node to the current node. The value of the H is the estimated distance from the current node to the goal node, and the value of the F is the sum of the G and H.

As shown in line 5, this A* algorithm is executed until the open set is not empty. If the open set is not empty, then EXTRACT-MIN-F function is executed. It makes a current node which has minimum value of F in the open set. Then this current node is deleted from the open set and added to the closed set.

After that, the next step is checking whether the current node and the goal node are same or not. If it is the goal node, it means that we get the shortest path and return the solution. On the contrary, if it is not the goal node, its neighbor nodes Ns are in three cases, line 9, 10 and 15.

If the neighbor node N is in the closed set, then nothing to do (line 9). Two other cases have a common function of calculating the values of G, H and F. The difference of two cases is whether it needs to execute a relaxation or not. If the neighbor node is in the open set, the relaxation is needed. The relaxation is checking the possibility to improve the value of G. After the relaxation, the value of G can be changed to lower value, if the calculated value of G in the relaxation is lower than the previous value of G (lines 12~13).

The point of design in hardware is a EXTRACT-MIN-F function. Its function includes searching the open set and finding a node having minimum value of the F. But the open set is controlled independently, so searching the open set isn't a problem. Only finding a minimum value is considerable. In order to finding a minimum value, many comparisons are needed. Therefore it takes long time relatively.

$$\text{EXTRACT-MIN-F time} \propto N_{open} * T_{access} \qquad (1)$$

The computational time satisfies (1). N_{open} represents the number of nodes in the open set and T_{access} is the time of memory access. T_{access} can be neglected, because it is affected by what kind of memory is used. It is system's specification, so we are not interested in that term. Therefore, its performance is determined by the number of nodes in the open set, N_{open}.

Reference [2] is a case of comparing all nodes. It proposes bubble sort and computational time is determined by N_{open}. But its time complexity is $O(N_{open})$. Also heap is possible to be implemented ([5]). Its time complexity to sort is nearly $O(\log N_{open})$. But our proposed architecture shows better performance, O(1).

III. Proposed Architecture

A shift register is a group of flip flops set up in a linear fashion which have their inputs and outputs connected together. If we can sort shift register, then it is possible to extract the node having the minimum value of the F. The priority queue is one of the examples.

Fig. 2 represents the sorted shift register. Every register corresponds to the comparator. The comparator transmits control signals to the controller. According to control signals, the controller gives signals to shift register and shift registers are sorted. On this account, time complexity of the sorted shift register is O(1).

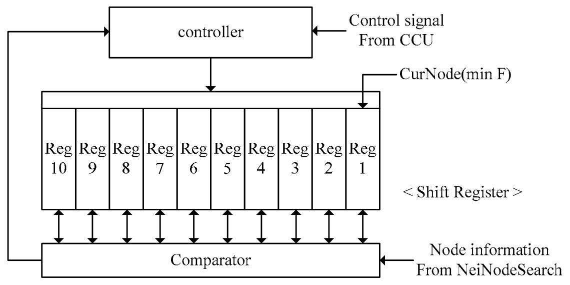

Figure 2. Sorted Shift Register

Now, we would consider that shift register is finite. As the loop is repeated, the number of elements of the open set is increased. If it is bigger than the number of shift register, then proposed architecture isn't reliable any more. Reference [2] mentioned how the number of registers is determined. They experiment on the specific map, Los Angeles street graph. As a result, they implement queues having the size of 16 elements. In this case, 16 elements are determined by specific map. The size of queues should be changed, if the map is not about Los Angeles street graph. It means that additional experiment is needed, whenever you adopt other maps. But it is not our goal. Our proposal is universally applicable architecture.

We'll describe our main idea in the sorted shift register. There are two shift registers in Fig. 3.

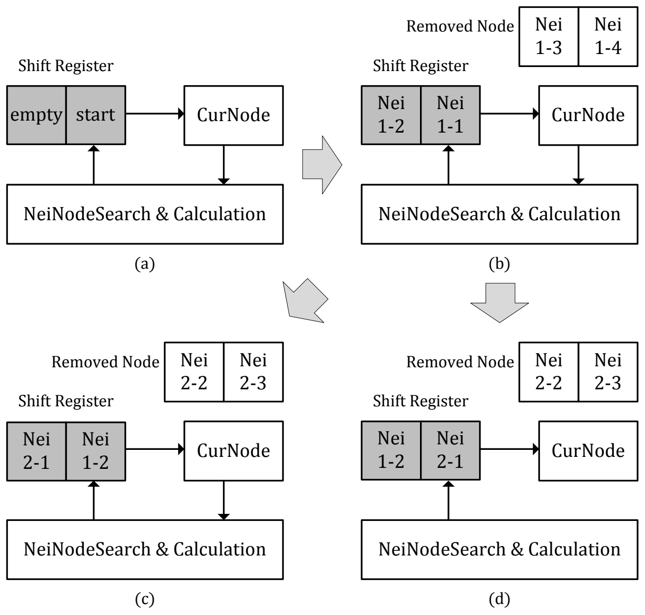

Figure 3. Sorted shift register function on A* algorithm

First time, start node is stored in shift register in (a), Fig. 3. The open set's element is only start node, so start node is extracted to the CurNode. Because of changing the CurNode, the NeiNodeSearch & Calculation module figures out information of each neighbor node of the CurNode. Neighbor nodes are defined like Nei X-Y. The X means Xth loop and the Y is numbered in the order of smaller value of F. In (b), Fig. 3, four neighbor nodes are calculated and two minimum of them, Nei 1-1 and Nei 1-2, are stored and others, Nei 1-3 and Nei 1-4, are removed.

On the next loop, we can execute EXTRACT-MIN-F, because shift register guarantees an element having a minimum value of F. Then the Nei 1-1 is extracted, and the NeiNodeSearch & Calculation module outputs second loop set of neighbor nodes, the Nei 2-Y. In (c), (d), Fig. 3, it shows possible array of the shift register. The Nei 1-2 having a minimum value of F and the Nei 2-Y are compared and it makes a candidate node of the next current node. This node is guaranteed to be a current node until next loop, because next loop is not affected by removed node in (b), Fig. 3.

In other words, two nodes in the shift register in (b), Fig. 3 are candidate nodes which can be the CurNode until next two loop and two nodes removed from the shift register can be the CurNode after three loops later.

From this, we can analogize the fact that the number of registers determines the number of valid loops. And valid loop is started from the event that the shift register is full of node information. Therefore the structure in Fig. 2 is valid until 10 loops.

But after 10 cycles, this architecture is not reliable, so we need an additional module. This additional module has to sort the open set and update the sorted shift register. It is nearly same to Fig. 2, but it has more registers and can access to memory.

$$\text{The number of registers} \fallingdotseq 10 * 4 - 10 = 30 \qquad (2)$$

Equation (2) represents a approximate calculation. First number 10 means the number of loops. The number 4 is the maximum number of neighbor nodes(we assume that an intersection has four edges), and second number 10 means the number of nodes is changed from the open set to the closed set. Therefore additional module's number of registers is determined by 30.

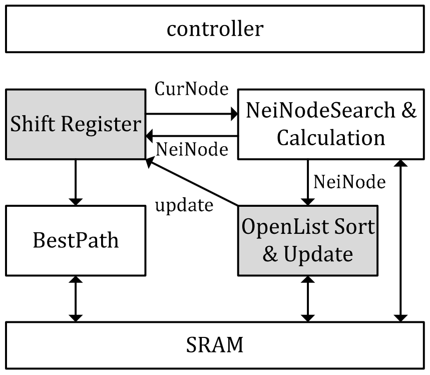

Figure 4. The top-level block diagram of proposed A* algorithm

In according to the map, the number of nodes is a striking contrast. Because of that reason, architecture included memory of enough size. The format of the memory is not only important, but also reducing memory access is a key of implementation. The OpenListSort & Update module is using linked list after valid loops of the shift register. Linked list minimize memory access, so it is suitable to our architecture.

Fig. 4 represents the top-level block diagram. There is the central controller to control each module, and the Bestpath module returns the shortest path. The SRAM stores node information and the open set. Two shaded modules in Fig. 4 manage the open set.

IV. Simulation Results

In order to verify, we made maps having variable nodes, 16, 64, 256 and 1024. They are quite reliable, because each node is located in rectangular coordinates and the cost (or distance) of each edge is calculated in that coordinate systems. And we design an A* algorithm model by using C language.

Fig. 5 represents the comparison of each way with different number of nodes.

$$\text{The clock cycles to sort} = T_{sort} + T_{access} \qquad (3)$$

In order to sort, the clock cycles are calculated like (3). T_{sort} is the time to sort and T_{access} is the time to access memory. The sort is executed after 10 loops. In order to simulate the proposed model, we randomly choose 15 pairs of nodes, the start and goal nodes.

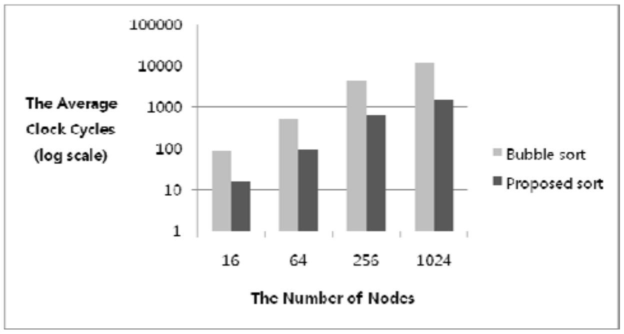

Figure 5. Average clock cycles as a function of the number of nodes and sorting algorithm

The time complexity of the bubble sort algorithm is $O(N^2)$. But our architecture's time complexity is only O(1), because every register corresponds to each one of comparators. For that reason, we can reduce average clock cycles.

Fig. 5 represents log scale on y-axis. As mentioned above, the average clock cycles include memory fetch time and the time to sort each list. The clock cycles of bubble sort need approximately 10 times more than the proposed sort time. As the number of nodes is increased, the increment of the average clock cycles to sort is reduced. It means proposed architecture is efficient as the number of nodes is increased.

Fig. 6 shows memory fetch clock cycles with specific start and goal nodes. This simulation is carried out with a map having 256 nodes. The start node and goal node are randomly selected.

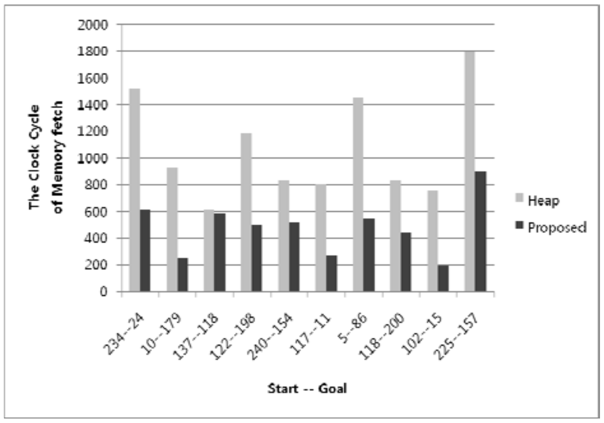

Figure 6. The comparison of clock cycles of memory fetch between heap and proposed architecture with specific node

To sort with heap architecture, it should execute many comparisons between parent node and two children nodes. So it causes more memory access. On the other hand, proposed architecture needs short time to sort, O(1). Memory access is occurred after 10 loops, and it is minimized by linked list architecture. Proposed architecture needs nearly half of time to memory access. The more memory accesses cause the more power consumption. Accordingly, proposed architecture is suitable for the hardware implementation.

There are simulation results with variable number of shift registers in Fig. 7. As the number of shift registers is increased, the average clock cycles are reduced, because the number of registers indicates the number of valid loops. The number of valid loops means the period of sorting the memory. As the period is longer, the number of times to sort the memory is less needed. It can be expected to reduce sorting time and to improve performance with variable number of them.

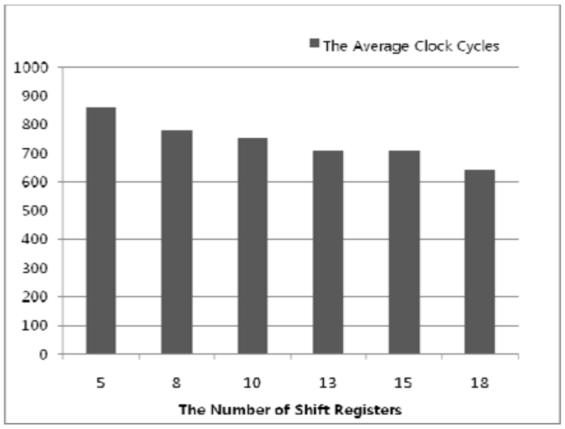

Figure 7. The comparison of the average clock cycles with variable number of shift registers

But you should keep in mind that the relation between the number of shift registers and cost is a trade-off. The more registers are implemented, the more size of area is needed.

V. Conclusion and Future Work

This paper gives novel idea to design a shortest path search engine. The essential point of our work is reducing memory access and managing the open set efficiently. Many times of memory accesses cause large power consumption in the system. Thus we propose a sorted shift register, because it is appropriate to be implemented as hardware.

Sorted shift register's EXTRACT-MIN-F is constant time complexity, O(1). And sorting executes each valid loops later, not every loop. Those features make possible to implement pipeline architecture.

Also linked list data structure is efficient to our hardware model. It minimizes memory access, so our architecture has a feature of low power consumption.

Our future work is designing in hardware using proposed architecture, not modeling. If it is implemented as hardware, there are some ways to reduce cost. For example, we can put two modules managing the open set together. Because both of them are formed with an array of shift registers. And Implementing pipeline is one of them. Pipeline architecture improves the system significantly. Also [2], [3] and [4] commonly said parallel architecture improves performance efficiently. Our proposed architecture is also possible to be implemented in parallel.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2rd ed., The MIT Press, 2001, pp. 580-619.

[2] Z. K. Baker and M. Gokhale, "On the Acceleration of Shortest Path Calculations in Transportation Networks", Proc. The Symposium on Field-Programmable Custom Computing Machines (FCCM'07), April 2007, pp 23-34, doi:10.1109/FCCM.2007.46

[3] I. Fernandez, J. Castillo, C. Pedraza, C. Sanchez and J. I. Martinez, "Parallel Implementation of The Shortest Path Algorithm on FPGA", Proc. The Southern Conference on Programmable Logic, March 2008, pp. 245-248, doi : 10.1109/SPL.2008.4547768

[4] M. Tommiska and J. Skyttä, "Dijkstra's Shortest Path Routing Algorithm in Reconfigurable Hardware", in Lecture Notes in Computer Science(LNCS), vol. 2147/2001, Springer Berlin / Heidelberg, 2001, pp. 653-657

[5] A. Ioannou and M. Katevenis. "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High Speed Networks". In IEEE/ACM Transactions on Networking, vol. 15, issue 2, April 2007, pp. 450-461, doi: 10.1109/TNET.2007892882

[6] A. Patel, Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/