

FPGA-Accelerated Pathfinding System (FAPS)

1st Nikita Agrawal

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
nagrawal@mit.edu*

2nd Shravika Pendyala

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
shravika@mit.edu*

3rd Kenneth Chap

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
kchap@mit.edu*

Abstract—Real-time navigation and path finding in large-scale road networks requires efficient shortest-path computation to ensure timely and accurate routing. Conventional navigation systems running on CPUs can become bottlenecked when operating on graphs with millions of nodes. We propose FPGA-Accelerated Path finding System (FAPS), a hardware-optimized solution designed to compute shortest paths with low latency. FAPS implements an A* search using a min-heap priority queue, visited list, and cost table, while storing large graph structures in DRAM. Our goal is to experimentally measure FAPS and CPU computation time on a Vermont road network, to evaluate their suitability for real-time navigation and other latency-critical applications.

Index Terms—Digital systems, Field programmable gate arrays, Graph traversal, Navigation systems

I. INTRODUCTION

Navigation systems use shortest-path algorithms to find the fastest route between two points on a road network [1]. These algorithms use a graph representation where nodes correspond to intersections and edges correspond to roads with weights based on distance. However, computing the fastest path can be time and memory-resource consuming for large graphs [2] and requests with large distances between start and destination nodes. This latency can lead to delays in real-time navigation applications, impacting user experience and system efficiency. By offloading computation to FPGAs, navigation systems can dramatically reduce latency [3], handle higher user loads, and maintain real-time responsiveness. The FPGA-Accelerated Pathfinding System (FAPS) will provide low-latency shortest-path computation between an input start and end destination on large-scale road networks. FAPS leverages the custom memory access capabilities of FPGAs to accelerate graph-search algorithms, targeting real-time navigation.

II. A* ALGORITHM

A* is a graph traversal and pathfinding algorithm designed to find the shortest path between a start node and a goal node. It combines features of Dijkstra's algorithm and greedy best-first search by considering both the cost to reach a node from the start, called the thus-far cost ($g(n)$), and an estimated cost

from that node to the goal, called the heuristic ($h(n)$). The heuristic is a function that provides an informed guess of the remaining cost to reach the goal to indicate whether the node traversal would go closer to the goal node or farther away from the goal node. A* maintains a priority queue to store nodes sorted by their total estimated cost $f(n) = g(n) + h(n)$. At each step, it explores the node with the lowest $f(n)$, updates cost information for its neighbors if a shorter path is found, and continues until the goal node is reached. Using a heuristic distinguishes A* from other search algorithms because it allows A* to focus its search, significantly improving efficiency.

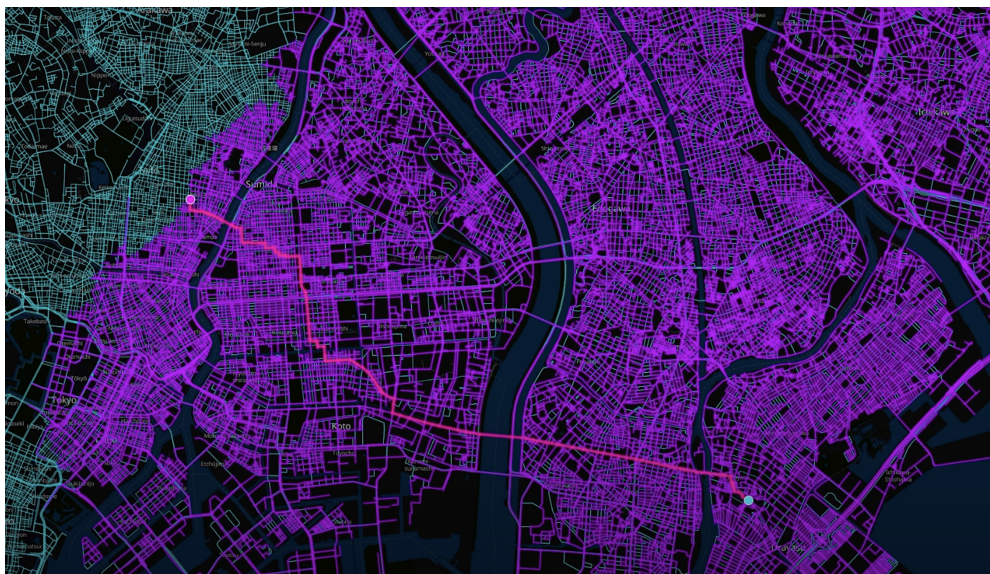
Bi-directional A* significantly reduces the number of nodes explored compared to standard A* and Dijkstra's algorithm by simultaneously searching from both the start and goal nodes and meeting in the middle. This approach effectively halves the search space, allowing the algorithm to focus only on the regions most likely to contain the shortest path. Standard A* still explores more nodes than bi-directional A* because it searches outward from the start node alone, potentially traversing irrelevant paths before reaching the goal. In contrast, Dijkstra's algorithm, which does not use a heuristic, explores all nodes in order of increasing distance from the start, often visiting a much larger portion of the graph. This comparison is shown in Figure 1.



(a) Bi-directional A* searches simultaneously from start and goal, meeting in the middle, effectively halving the search space.



(b) Standard A* uses a heuristic to guide its search from the start node, reducing nodes explored compared to Dijkstra but still visiting irrelevant paths.



(c) Dijkstra's algorithm explores all nodes without a heuristic, resulting in the largest search space.

Fig. 1: Comparison of node exploration for Dijkstra, standard A*, and bi-directional A* algorithms [4].

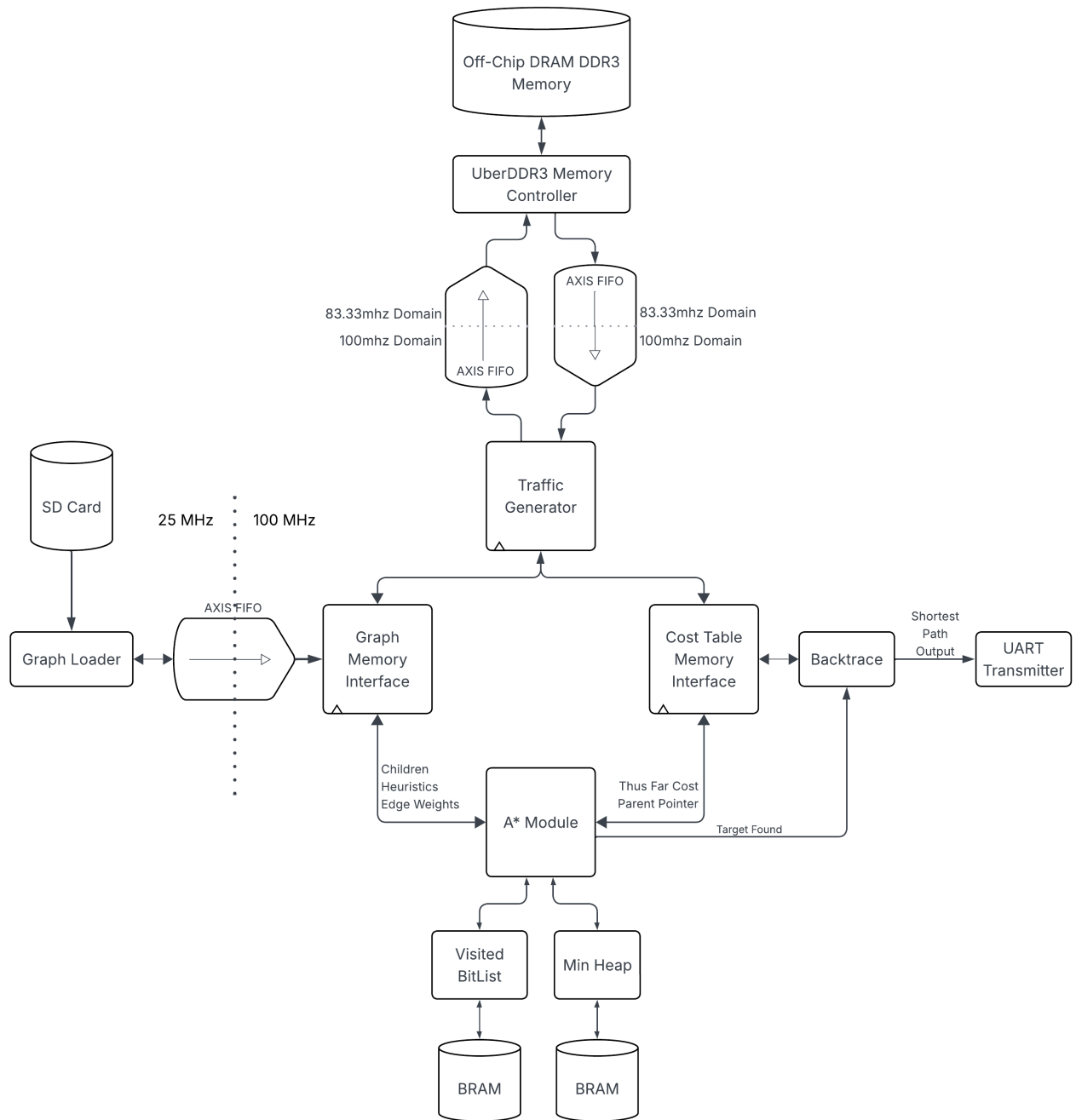


Fig. 2: System Architecture and Data Flow Diagram

III. GRAPH PRE-PROCESSING

The decision to structure the DRAM and all bit-widths was based on the graph chosen to be preprocessed in Python 3.12. In this design, a map of Vermont provided by OpenStreetMap (OSM) was utilized. In the future, other maps in the future may be tested to test the design's robustness and flexibility to various inputs. The start node s was determined to be Santa's Land USA, and destination node d was determined to be the original Ben and Jerry's location. Reasoning for picking these locations will be explained further.

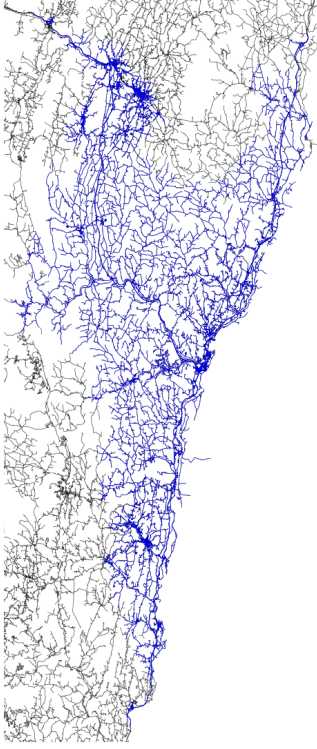


Fig. 3: Blue indicates the portion of Vermont this design will run on.

Driving network information from the OSM Vermont map was first extracted using pyrosm, which contained all node IDs and distances in meters for each road. However, to come up with the necessary components for the graph that could fit onto the DRAM of the FPGA, the following was done:

- 1) Each **weight** of the graph is the amount of seconds t it takes to get from node u to node v . t was calculated by dividing the given distances in meters by the given speed limits in meters per second. For missing speed limits, a survey of all Vermont road types were taken to determine default speeds per road type. It is important to use time instead of distance for weights as the purpose of GPS systems is to get from u to v in the shortest possible time.
- 2) The **heuristics** of every node were determined by picking a destination d and calculating the great circle distance between all other nodes to node d , divided

by the max speed on the map for normalization. Great circle distance was chosen instead of Euclidean distance because each node uses a latitude-longitude coordinate system, and the Earth is spherical. Note that because the heuristic depends on the destination picked, preprocessing will need to be done every time the destination changes. Changing the start node will not require another graph to be precomputed.

- 3) All **node IDs** were renumbered to be within 0 to N inclusive, N representing the amount of nodes the graph will contain. This enumeration process was done through the Morton (Z-curve) remapping process based on heuristic values calculated earlier. This was done to improve locality since nodes are accessed frequently, and having nodes u and v closer to each other in ID number enables future optimizations in DRAM access.

After finding a subgraph of Vermont that contained $N = 300,000$ nodes with a maximum of 4 neighbors (see Figure 3), the purposes of these design choices explained later, appropriate bit-widths were determined. The maximum node ID is 299,999, so 19 bits are needed to support node IDs. The maximum time weight is 98.818 seconds, which requires 7 bits if only the integer portion is accounted for. However, there are many weight amounts that are less than a second. Therefore, all weights were multiplied by 10, causing the required amount of bits to increase from 7 bits to 10 bits. The maximum heuristic amount was 5491.178 seconds, which requires 13 bits when accounting for the integer portion. The reason why heuristics was not multiplied by 10 like for the weights is because most heuristics had distinct integer portions, and heuristics does not affect the accuracy of the A* algorithm, but rather the latency. This helps save some bits.

To further determine bit-widths specifically for the A* algorithm, more metadata was collected. It was found that 13 bits were needed to store the total time needed to travel between s and d . s and d was chosen for this, as they are on opposite sides of the map and represent one of the more extreme paths possible (see Figure 4). This leads to the 14 bits needed to store $h(n)$, the heuristic cost at any given moment plus the total time traveled so far. Since a heuristic cost is 13 bits, which is only used when deciding a path and never summed over time, and the total time to travel from s to d uses 13 bits, 14 bits is needed for 13-bit addition without overflow. Another bit was added to these A* requirements for safekeeping.

A 300,000 node graph with each having a maximum of 4 neighbors was chosen out of caution with DRAM storage, as the design originally planned for storing each node with the following information, which takes up 148 bits. Since this design's DRAM only store lines of 128 bits, 2 lines must be used. This means 256 bits are used to store one node. Multiplying this by 300,000 yields 76,800,000 bits (76.8 Mb). Later redesigns of memory took advantage of such space, which reduced the amount of graph requests needed and can support more neighbors in the future. See subsection V-A for more information.

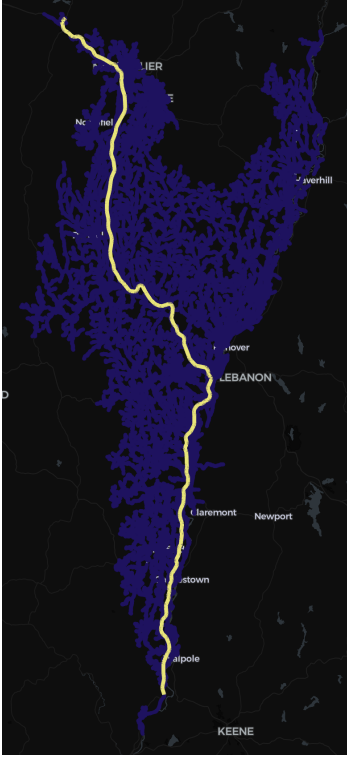


Fig. 4: The shortest path from s to d , with s located near the bottom of the subgraph and d located near the top.

Lastly, the design assumes an undirected weighted graph, as later optimizations such as the bidirectional A* will require multiple A* algorithm running in different directions.

IV. MEMORY STRUCTURES

The A* module has several key parameters that are chosen based on the size and structure of the graph. The `MAX_NEIGHBORS` parameter is set to 4, which is the maximum number of neighboring nodes each node can have. The `NODE_BITS` parameter is set to 19, which is sufficient to represent up to 300,000 nodes, since $2^{19} = 524,288$. The `COST_BITS` parameter is set to 14 to store the maximum possible path cost from the start node to any other node in the graph and the maximum possible heuristic value as determined earlier. The `F_BITS` parameter is set to 15 to store the combined cost and heuristic values (f cost). These parameters ensure that the module can efficiently store node ids, path costs, and priority values for a graph containing up to 300,000 nodes.

A. BRAM

The min-heap and visited list memory structures are built on a Xilinx true dual-port BRAM in high-performance mode. This has a two-cycle read latency while allowing independent and simultaneous reads or writes on both ports. The min-heap and visited list BRAM modules are designed around this timing. This provides a stable foundation for all BRAM data structures used by the A* module.

B. DRAM

DRAM serves as the memory for node adjacency lists, edge weights, heuristic values, and the parent and thus far cost information used by the A* algorithm. Since DRAM accesses have a significantly higher latency than BRAM, all interactions are handled through a traffic generator interface that issues read and write requests and returns data once transactions are complete. Each DRAM access reads or writes a full 128-bit cache line.

V. MODULES

A. Graph

The graph module represents the underlying graph structure and provides each node's neighbors, edge weights, and heuristics as shown in Figure 5. Nodes can have up to four neighbors, and each neighbor's information including node ID, edge weight, and heuristic is stored in DRAM. A finite-state machine handles requests and waits for data from a traffic generator interface. This module outputs the list of neighbors, their weights, heuristics, the heuristic of the current node, the number of neighbors for the current node, and signals when the output is ready for the A* module.

As each node will take up 256 bits regardless, the original design was further optimized to have the following features.

- Heuristics for each neighbor (child) are pasted inside each node to reduce access to the DRAM. This saves at most 4 DRAM calls per node.
- An indicator bit is included to indicate whether this node has more than 3 neighbors, which would result in 2 valid entries in DRAM. As a result, upon completion of the first memory access, this can save 1 DRAM call if there are less than 3 neighbors for the node.
- Storage of the node ID was removed as this information is already known beforehand when a graph read request is being made by any module.
- The A* module does not require knowing the number of children a node has. However, to save space for future changes, an unused child is filled with 1's and combinational logic is used to calculate this information.

B. Min-Heap Priority Queue

The priority queue is implemented as a binary min-heap in BRAM where each entry stores a (node id, f cost) pair as shown in Figure 6. A min-heap is used instead of other methods such as merge sort because it supports insertions and extractions in $O(\log n)$ time, which is crucial in scaling up the number of nodes. We decided to cap the size of the min heap to roughly 50,000 nodes, and this is because the A* is a very directional search resulting in most of the interactions being in the top thousands of nodes. Moreover, enforcing this limit would improve memory resource utilization as a notable portion of the graph is never visited in search. The min-heap module breaks down push and pop operations into multi-cycle states to account for the two-cycle latency of the BRAM. During heapify-up and heapify-down, the parent and

Addr		127	109 108	99 98	86 85	67 66	57 56	44 43	31 30	1	0
0	Node 0	Child 1 Node_ID	C1 Edge Weight	C1 Heuristic	Child 2 Node_ID	C2 Edge Weight	C2 Heuristic	Parent Heuristic	'0	Indicator Bit	
1		Child 3 Node_ID	C3 Edge Weight	C3 Heuristic	Child 4 Node_ID	C4 Edge Weight	C4 Heuristic	Parent Heuristic	'0		
:	:	:	:	:	:	:	:	:	:	:	
:	:	:	:	:	:	:	:	:	:	:	
:	:	:	:	:	:	:	:	:	:	:	
:	:	:	:	:	:	:	:	:	:	:	
599,998	Node 299,000	Child 1 Node_ID	C1 Edge Weight	C1 Heuristic	Child 2 Node_ID	C2 Edge Weight	C2 Heuristic	Parent Heuristic	'0	Indicator Bit	
599,999		Child 3 Node_ID	C3 Edge Weight	C3 Heuristic	Child 4 Node_ID	C4 Edge Weight	C4 Heuristic	Parent Heuristic	'0		

Fig. 5: Memory layout of the Graph module in DRAM.

child nodes are fetched through dual-port reads, comparisons occur after the data becomes valid, and writes to BRAM update the ordering. Dual-port access allows two nodes to be fetched per cycle, maintaining logarithmic-time behavior in hardware. Figure 7 shows the finite-state machine of the min-heap module.

Fig. 6: Memory layout of the Min-Heap module in BRAM.

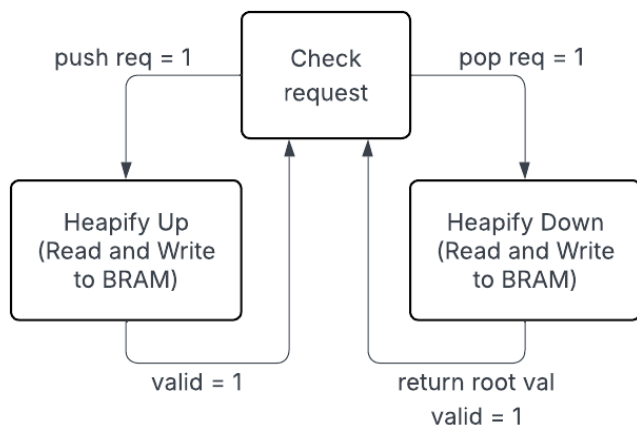


Fig. 7: Finite-state machine of the min-heap module.

The min-heap module was validated using a Python-based cocotb testbench designed to verify correctness under randomized conditions. A clock generator drives the simulation, and the min-heap is reset before each test. The testbench defines asynchronous helper functions for pushing and popping values, each waiting on the module’s handshake signals (push done, pop valid) to ensure correct synchronization with the hardware finite-state machine. Multiple test cases are executed such as random push sequences, sorted and reverse-sorted inputs, duplicate values, and interleaved push-pop operations. After each pop, the returned element is compared against the expected minimum value computed in software, ensuring that the min-heap property is maintained at every step. The final integration

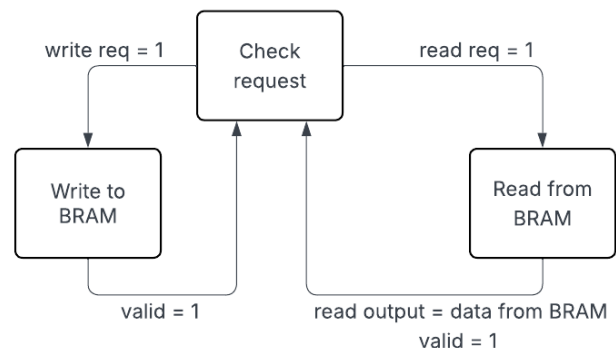


Fig. 8: Finite-state machine of the visited list module.

The visited list module was validated using a Python-based cocotb testbench designed to verify correctness under randomized conditions. A clock generator drives the simulation, and the visited list is reset before each test. The testbench defines asynchronous helper functions for reading and writing values, each waiting on the module's handshake signal (data out valid) to ensure correct synchronization with the hardware finite-state machine. The first test validated basic functionality by confirming that uninitialized nodes read as unvisited, and that a single write correctly updated the corresponding entry. The second test performed randomized testing where a shuffled set of nodes was written to BRAM, and the testbench verified that all nodes returned the expected visited status. The final test interleaved writes and reads in randomized order to mimic dynamic A* exploration behavior and confirm that accesses did not introduce timing hazards or incorrect read-outs. This

comprehensive cocotb-based testing framework verifies both correctness and timing behavior and provides high confidence in the visited list module implementation.

D. Cost Table

The cost table is implemented in DRAM and accessed through a traffic-generator interface that provides read and write transactions. Each entry stores a (parent node id, $g(n)$) pair as shown in Figure 9. The cost table module accepts read and write requests from the A* module and forwards them to the traffic generator when its request ready signal is asserted. For writes, the module constructs a 128-bit DRAM line which has the parent pointer and cost fields. For reads, the module returns a full DRAM line that is unpacked into the corresponding fields once the traffic generator asserts that the memory request has been completed.

	32	14	13	0
Node 0	Parent Pointer (Node_ID)		Thus Far Cost	
:	:	:	:	:
:	:	:	:	:
:	:	:	:	:
Node 299,999	Parent Pointer (Node_ID)		Thus Far Cost	

Fig. 9: Memory layout of the Cost Table module in BRAM.

E. A*

The A* module searches from the start node expands nodes in order based on the min-heap until it reaches the goal node. For every expanded node, the module retrieves its neighbors from the graph module, checks if the neighbor is visited through the visited list module, and issues DRAM reads to the cost table to load the stored g-cost and parent pointer. If a neighbor has an improved path cost, the module issues a DRAM write to update the cost table and pushes the node into the min-heap in BRAM. The visited list is updated only after a node is fully processed and all of its neighbors are pushed to the min-heap in BRAM. Then, the next node to be explored is popped from the min-heap in BRAM. Figure 10 shows the finite-state machine of the A* module.

The A* module was validated using a Python-based cocotb testbench designed to verify correctness under randomized conditions and simulate its interactions with both the graph and cost table modules. A clock generator drives the simulation, and the A* module is reset at the start of each test. The testbench fully models the graph module by returning neighbor lists, edge weights, and heuristic values for each node whenever graph request is high. The cost table is modeled as a Python dictionary where read requests return stored (g cost, parent) values, and write requests update the dictionary while setting an acknowledge signal high to synchronize with the hardware finite-state machine. The first test confirmed basic functionality by checking that the module correctly processes the start node and expands its neighbors. The second test was on a 50-node graph to mimic dynamic A* exploration behavior and verify that heap operations, visited-node tracking,

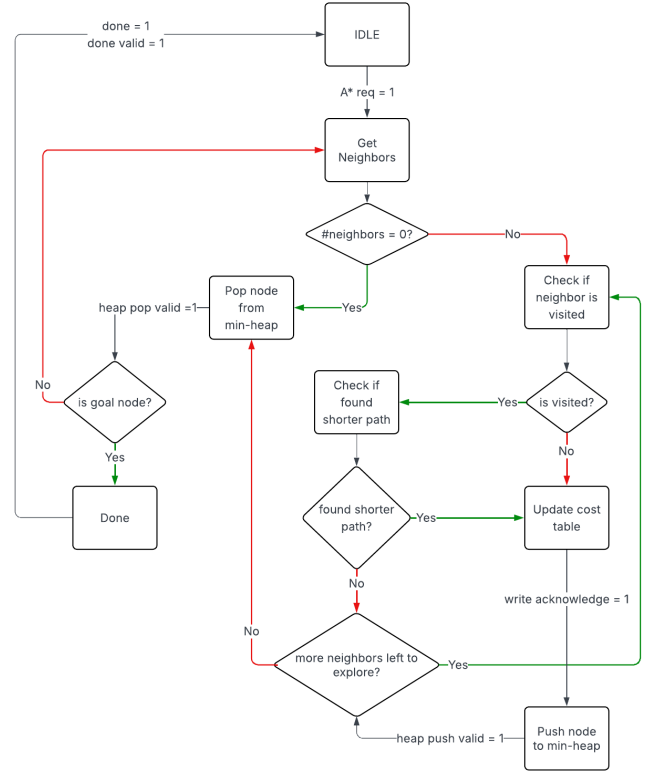


Fig. 10: Finite-state machine of the visited list module.

and cost-table updates correctly guided the search toward the goal node. The final path reconstructed from parent pointers was compared against the known correct path, confirming that the module computes the optimal route. This testbench provides end-to-end verification of the A* module, ensuring correct coordination between the min-heap, visited list, and interactions with the simulated graph and cost table modules.

VI. RESULTS

To validate the correctness of our hardware A* implementation, we conducted a series of experiments using multiple starting nodes distributed across the full map. For each start node, the A* algorithm successfully computed the shortest path to the target, demonstrating that the FPGA design consistently produces correct solutions regardless of initial search location. These tests also confirm that the cost table, visited list, and min-heap remain fully synchronized under varying graph-local conditions.

Figure 11 illustrates the complete set of paths generated by A*, with each trajectory corresponding to a different start node as mentioned earlier. All paths converge to the correct goal location and follow the expected optimal route through the traversable regions of the map. Table ?? summarizes repeated executions for the multiple start nodes and End Node 147182. The Node IDs are assigned in such a way that nodes that are closer together by number of edges have Node IDs that are closer together in number. Across the multiple runs on

each start/end node combination, the FPGA implementation produced identical shortest-path results, indicating little variability in performance.

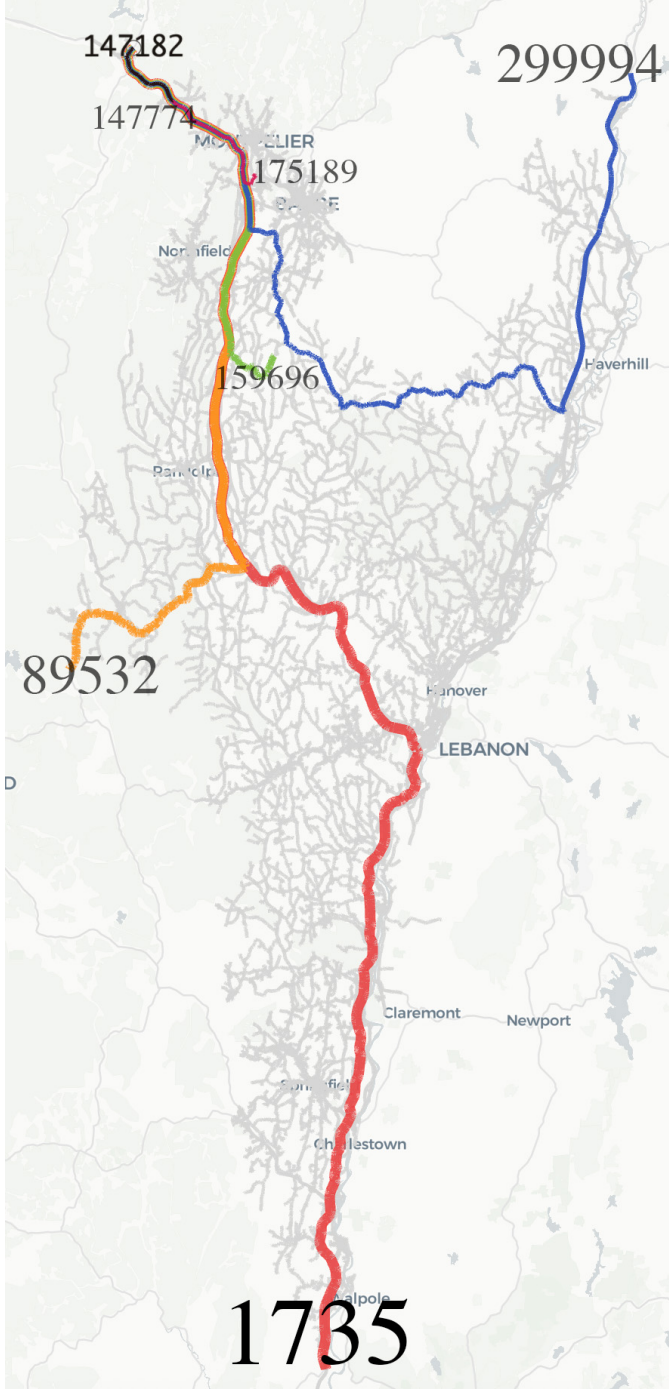


Fig. 11: Complete set of paths generated by A* across 6 varying start nodes. The start node IDs are 1735, 299994, 89532, 159696, 175189, 14774. The end node ID is 147182.

These results demonstrate that the FPGA A* module is both functionally correct and stable across diverse initial conditions, validating its use as a reliable single-source path finding engine.

Start Id	End Id	Run	Total Time (sec)	Cycles	DRAM Cycles	DRAM Waiting
1375	147182	1	0.36962886	36962886	26454051	71.57%
1375	147182	2	0.36963137	36963137	26454222	71.57%
1375	147182	3	0.36964198	36964198	26454899	71.57%
1375	147182	4	0.36964003	36964003	26454574	71.57%
1375	147182	5	0.36963992	36963992	26454563	71.57%
299994	147182	1	0.36581997	36581997	25821634	70.59%
299994	147182	2	0.36581784	36581784	25821343	70.59%
299994	147182	3	0.36582307	36582307	25821964	70.59%
299994	147182	4	0.36582466	36582466	25822117	70.59%
299994	147182	5	0.36582363	36582363	25822000	70.59%
89532	147182	1	0.24934333	24934333	17156087	68.81%
89532	147182	2	0.24934630	24934630	17156383	68.81%
89532	147182	3	0.24934245	24934245	17155993	68.80%
89532	147182	4	0.24933838	24933838	17155576	68.80%
89532	147182	5	0.24935210	24935210	17156795	68.81%
159696	147182	1	0.10745642	10745642	7641659	71.11%
159696	147182	2	0.10745703	10745703	7641720	71.11%
159696	147182	3	0.10744725	10744725	7640766	71.11%
159696	147182	4	0.10745183	10745183	7641284	71.11%
159696	147182	5	0.10745090	10745090	7641182	71.11%
175189	147182	1	0.04658535	4658535	3394661	72.87%
175189	147182	2	0.04658321	4658321	3394369	72.87%
175189	147182	3	0.04658321	4658321	3394369	72.87%
175189	147182	4	0.04658156	4658156	3394213	72.87%
175189	147182	5	0.04657743	4657743	3393780	72.86%
147774	147182	1	0.00348638	348638	265129	76.05%
147774	147182	2	0.00348646	348646	265137	76.05%
147774	147182	3	0.00349153	349153	265582	76.06%
147774	147182	4	0.00348798	348798	265243	76.04%
147774	147182	5	0.00348874	348874	265407	76.08%

TABLE I: Execution statistics for runs with varying Start IDs.

VII. PYTHON CPU COMPARISON

To compare our FPGA implementation with a Python implementation running on a CPU, we implemented the A* algorithm in Python. We primarily focused on the longest-shortest path emphasized throughout the paper as a primary mode of comparison due to its both depth and breadth of search. This path started at Node Start Id = 1375 and ended at Node End Id = 147182, and the Python CPU took [0.25, 0.197, 0.3728] seconds while the FPGA A* implementation took 0.3696 sec. We ran the same Python file on 3 computers with the first result (0.25) running on a M2 Max processor, the second result (0.197) running on a M3 Pro Processor, and the third result (0.3728) running on an Intel i7 Processor. While upon first glance it appears that the FPGA performed much slower, we noted that the FPGA was running 100MHz while the CPUs were in the GHz range. However, because the CPU time measurement is dependent not only on the Processor speed but also other modern hardware optimizations such as boost frequency scaling to improve performance upon higher workload, the Python CPU results, while a good reference to have, render our FPGA vs. CPU implementation test inconclusive. We were not able to find an easy way to directly measure CPU cycle counts which was the number we were hoping to have for comparison.

VIII. DISCUSSION

Our FPGA implementation of single- and bidirectional A* pathfinding is resource-efficient, using 25.5 Block RAM tiles (34% of capacity), mostly RAMB36 blocks, and minimal LUT-based memory. Slice logic utilization is 17% for LUTs and 11% for registers. No DSPs are currently used. The design meets timing requirements with a WNS of 0.136 ns, with critical paths primarily in the min-heap and neighbor expansion logic. We have achieved our minimum and ideal project goals by correctly computing shortest paths and supporting

bidirectional search. With minor adjustments, the design could handle dynamic edge weights, batch searches, or real-time debug output. For future improvements, processing neighbor expansion in parallel could have better scalability. In our experiments, we processed up to 300,000 nodes with a maximum of 4 neighbors per node, although in the future, more memory could allow for larger graphs and more extensive parallelism.

A stronger heuristic could have been employed in future runs. Although our individual maximum edge weight was 98 seconds, the maximum heuristic value from pre-processing was 5,491 seconds. However, the total cost to travel from the start node s to the destination node d was 61,189 seconds, a factor we had not fully accounted for. As a result, as the A* algorithm progresses farther from the start node, the thus far cost computations overpowers the heuristics in deciding which neighbor to explore. This resulted in the A* algorithm exploring a larger portion of the graph than anticipated. In future implementations, scaling the heuristic values in relation to the expected order of magnitude of the thus far cost would provide the algorithm with a stronger sense of direction, allowing it to more efficiently prioritize promising paths.

IX. OPTIMIZATION: BI-DIRECTIONAL A*

We implemented parallel A* searches to leverage the hardware's ability to process multiple nodes simultaneously. Our focus was on bi-directional search, where one A* search begins at the source node and another at the destination node. The two searches meet in the middle, combining to produce the shortest path.

We aimed to explore whether running parallel A* searches - one from the start node and the other from the destination - could outperform the standard A* in practice. However implementing this in practice proved to be challenging due to the memory requirements and the necessary routing, as each A* search would require its own pre-computed heuristics stored in DRAM. One experimental approach involved removing heuristics entirely from one direction while the other maintained the A* search, running both searches simultaneously. We expected the meeting point to be closer to the end node which appears as though the Dijkstra algorithm is running from one end and the A* algorithm is running from the other end. We found that depending on the graph structure, this method can yield different performance outcomes. Additionally, memory constraints can influence the effectiveness of this optimization, limiting the number of nodes that can be processed in parallel. We found that requests to memory were not evenly split between the two parallel A* searches, leading to the meeting node being closer to the end node. Further tests would need to be performed for memory optimization.

X. CONCLUSION

We present the FPGA-Accelerated Pathfinding System (FAPS), a hardware-based solution for low-latency shortest-path computation on large-scale road networks. By implementing the A* algorithm with memory structures including a min-heap priority queue, visited list, and cost table, and

leveraging DRAM and BRAM efficiently, FAPS demonstrates the potential of FPGAs to accelerate graph traversal tasks. FAPS provides a scalable and efficient platform for real-time navigation.

CONTRIBUTIONS

N.A. A* module (and testbenching), Bidirectional A* module (and testbenching), Min-Heap module with BRAM interface (and testbenching), Visited List module with BRAM interface (and testbenching), Python A* implementation.

S.P. System architecture and problem breakdown into module definitions, memory layout, traffic generator, bi-directional Traffic Generator and DDR3 interfaces, cost table. Debugged A* and helped create helpers for bidirectional A*. Created multiple versions of traffic generator.

K.C. Preprocessing inputs in Python, SD card controller-graph loading module, graph reading module, graph debugging/unloader via UART module, connections from graph modules to DRAM. Created console and measurement tools. Debugged A* using PMOD signals and other DRAM dumps.

REFERENCES

- [1] R. Rodríguez-Puente and M. S. Lazo-Cortés. Algorithm for shortest path search in geographic information systems by using reduced graphs. Computational Intelligence and Neuroscience, 2013.
- [2] R. Chen and C. Gotsman. Efficient fastest-path computations for road maps. Computational Visual Media, 2021.
- [3] Y. Zhou, T. Wang, and X. Jin. FPGA implementation of A algorithm for real-time path planning. Computational Intelligence and Neuroscience, 2020.
- [4] PhysicsFX. (2025). Bidirectional A* vs A* vs Dijkstra's Algorithm: Tokyo's Cyber Labyrinth. YouTube. <https://www.youtube.com/watch?v=WSVOYXY-iFU>