# Shortest Path: Algorithm beyond Dijkstra's

Yiyi Li, Xinlei Yu

***Abstract*—The shortest path algorithm computes the shortest and minimum cost path between a start node and an end node in a connected graph. There is a overwhelming desire to obtain a more efficient and universal solution for finding the shortest path. Dijkstra's algorithm is widely known as the shortest path algorithm for decades by Edsger W. Dijkstra in 1956. This paper discusses some of the related algorithms and optimization approaches of the algorithm that have been conducted using different data structures. This paper also discusses some of other shortest path algorithm. In the result section, the paper list some of key statistics of the three shortest path algorithms.**

***Keywords:*** Dijkstra's Algorithm, Graph Theory, Greedy Algorithm, Data Structure

## I. Introduction

With the development of Internet, especially the development of network traffics, mapping services and applications, and social media, the graph theory of computing theory takes more important role. Due to the increasing interest, many seemingly rudimentary operations become complicated when graphs containing large data sets are discussed. In this way, the shortest-path problem becomes one of the most well-studied and important topics in computer science. It is a very essential algorithm in many science and engineering field such as optimization of Very Large-Scale Integration (VLSI) routing, analysis of computer networks, and mapping applications (e.g. Google Map). Dijkstra's Algorithm is one of the most popular algorithms for solving this problem, which finds the shortest path from one source node to all other nodes of a graph with non-negative edge path costs.[1]

During this whole semester, we firstly got know about the most popular shortest path algorithms, such as Dijkstra algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm. Then, we run these three codes in LLVM IR, and compare the basic properties, including the nodes and edges in the graph, the diameter of the graph, and the parallelization degree, among this three algorithms. In milestone 3, we implemented the real world applications by using the mentioned algorithms, which can help us calculating the shortest path based on the road information. Also, in milestone 3, gem5 was introduced to simulate various kinds of running environment after getting the parallel files from LLVM, which we have accomplished in milestone 2. Last but not the least, in milestone 4, we continued to run the codes using a different simulation environment, such as different CPU types, different cache size, and different numbers of cores.

The basic knowledge of shortest-path algorithms is introduced in part II, the details of research approaches will be illustrated in part III, and results of each milestones will be demonstrated in part IV in details. Finally, we will discuss the achievement and limitations of this research in part V

## II. Related Work

Madkour et al.[2] divided the majority of shortest-path algorithm into two categories: *single-source shortest-path* (SSSP) and *all-pairs shortest path* (APSP), and Dijkstra's Algorithm solves the SSSP problem. This algorithm iterates until it finds the shortest path from the source vertex to all other vertices, whose time complexity is $O(n^2)$[3], where $n$ is the number of vertices. One advantage of this algorithm is that once the algorithm identifies the shortest path from the source vertex to another vertex, it marked the corresponding vertex as one of the solved vertices, so it does not need to traverse all edges. However, this algorithm only applies to non-negative weighted paths. And it does not apply to dynamic graphs.

Unlike Dijkstra's algorithm, Bellman[4] and Ford[5] proposed another SSSP algorithm that is able to process the graph with negative weight path to get the shortest path and detect negative cycles by traversal all neighbor edges and nodes within n-1 cycles. However, because of this comprehensive traversal, the time complexity of this algorithm is $O(mn)$, where n and m are the number of vertices and edges respectively.

Based on Fredman and Tarjan's research, Driscoll and Gabow[6] presented a new data structure called the relaxed heap in 1988. While Dijkstra's algorithm using relaxed heap has same time complexity as one using the F-heap, the relaxed heap enables a processor-efficient parallel implementation of Dijkstra's shortest path algorithm. Also, the relaxed heap is a binomial queue that allows violation of heap order.

On the other hand, Floyd-Warshall algorithm[7] is one of the most widely used algorithms to compute

shortest paths between all pairs of vertices in an edge weighted directed graph, which is known as APSP. By using dynamic programming algorithm, this algorithm has a worst-case run-time of $O(n^3)$ for graphs with n vertices. The Floyd-Warshall algorithm can also calculate the graph with negative-weighted edges.

These aforementioned algorithms and data structures optimized Dijkstra's algorithm in terms of time complexity, parallel operation, and processing paths with negative weights. In this paper, we will combine the content of the above and the actual performance of Dijkstra's algorithm to find better and more novel ways to optimize it from various aspects.

## III. Approaches

In this section, we give a detail view on optimization of Dijkstra's algorithm. We first plain Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm written in C language separately. We also find a way to implement these shortest-path algorithms into real world by reading the matrix from the existing files.

Then we use LLVM to generate different files by changing the number of input nodes and input matrices. Gephi, one of the visualization applications for networks and graphs, is used to get the properties of the graph. In addition, after getting the files from LLVM, networkX and community detection scripts provided in the class are used to partition the running steps into clusters within different partition algorithms.

After getting the cluster files, gem5 is used to simulate running environment. In gem5, we change the number of cores, which is implemented by changing the number of cluster files in "parallel" folder, the cache size, and the CPU type, and tried to find the connections between these parameters and the computational time.

### A. Different shortest path algorithms

In the Dijkstra's algorithm pseudo-code above, *dist* is an array to store the current distances between *source* to other vertices. Using *EXTRACT-MIN()* function decrement heap size and remove the first element of the array. The *parent* array stores pointers to its parent/previous nodes on the shortest path between the source node and the given vertex. If *dist[v] >dist[u]+weight[e]*, then the current path is replaced with *dist[u]+weight[e]*.

The Bellman-Ford's algorithm pseudo code shows that, compared with Dijkstra's algorithm, in this algorithm. Since Bellman-Ford's algorithm can process the graph with negative weights, this algorithm checks whether negative loop exists in the graph.

**Algorithm 1** Dijkstra's algorithm pseudo code

```
for each vertex v ∈ V do
    dist[v] ← ∞
    parent[v] ← UNDEFINED
end for
dist[s] ← 0
Q ← V
while Q ≠ ∅ do
    u ← EXTRACT − MIN(Q)
    for each edge e=(u,v) do
        if dist[v] >dist[u]+weight[e] then
            dist[v] ← dist[u] + weight[e]
            parent[v] ← u
        end if
    end for
end while
return parent[], dist[]
```

The pseudo-code of Floyd-Warshall algorithm shows that this algorithm use dynamic programming algorithm to record the shortest-path on each pair of nodes, and returns the updated matrix with shortest paths.

After finding the applications which implement the above algorithms in C language, we executed these applications multiple times in LLVM by changing the number of points in the input and getting different outputs. In the following section, we will analyze the files generated by LLVM by different tools to understand the various properties of each shortest path algorithm. To get the fundamental properties of the graphs generated by LLVM, we will use $Gephi$ to visualize the graphs, and calculate the values, such as average weighted degree, network diameter, etc.. The $NetworkX$ library in Python also becomes an important tool for us to calculate the assortativity coefficient and the parallelization degree. Despite the $NetworkX$, the script provided in the class can also calculate the parallelization degree from another perspective. In section IV, we will show the difference results of this value.

### B. Gem5 Simulation

The gem5 simulator is a open source modular platform for computer-system architecture research, encompassing system-level architecture as well as processor micro architecture. It simulates the passage of time as a series of discrete events. Its intended use is to simulate one or more computer systems in various ways. Gem 5 is beyond a simulator; it's a simulator platform that lets you build your own simulation system using as many components as possible. It was written in C++ and Python under BSD licence.

**Algorithm 2** Bellman–Ford's algorithm pseudo code

$distance := listofsizen$
$predecessor := listofsizen$

**for** each vertex $v \in V$ **do**
  $distance[v] \leftarrow \infty$
  $predecessor[v] \leftarrow NULL$
**end for**
$dist[s] \leftarrow 0$

**for** $|V|$-1 times **do**
  **for** each edge (u,v) with weight w in edges **do**
    **if** $distance[u] + w < distance[v] then$ **then**
      $distance[v] :=< distance[u] + w$
      $predecessor[v] := u$
    **end if**
  **end for**
**end for**

**for** each edge (u,v) with weight w in edges **do**
  **if** $distance[u] + w < distance[v] then.$ **then**
    $negaloop := [v, u]$
  **end if**
  **for** $|V|$-1 times **do**
    $u := negaloop[0]$
    **for** each edge (u, v) with weight w in edges **do**
      **if** $distance[u] + w < distance[v] then$ **then**
        $negaloop := concatenate([v], negaloop)$
      **end if**
    **end for**
  **end for**
**end for**

**return** $distance, predecessor$

The figure above shows our configuration of the Gem 5 system. The system include one CPU, two L1 cache, one L2 cache, and one memory control unit.

The gem5 able to simulate a complete system with devices and an operating system in full system mode.There are numbers of levels of support for executing Alpha, ARM, MIPS, Power, SPARC, RISC-V, and 64 bit x86 binaries on CPU models including two simple single CPI models, an out of order model, and an in order pipe lined model. In this paper, we utilized 64 bit x86 to analysis the performance of proposed algorithm. We also compare different input sizes of the graph for proposed algorithms with different lambda values.

After successfully running Gem5, the system will output an output folder that includes a config.ini, which contains a list of every SimObject created for the sim-

**Algorithm 3** Floyd-Warshall's algorithm pseudo code

H

$M[i, j] := \infty \ \forall i \neq j$
$M[i, j] := 0 \ \forall i$
$M[i, j] := c((i, j)) \ \forall (i, j) \in E(G)$
**for** each i := 1 to n do **do**
  **for** each j := 1 to n do **do**
    **for** each k := 1 to n do **do**
      **if** $M[j, k] > M[j, i] + M[i, k]$ **then**
        $M[j, k] := M[j, i] + M[i, k]$
      **end if**
    **end for**
  **end for**
**end for**

**for** i:=1 to n do **do**
  **if** $M[i, i] < 0$ **then**
    **return** $Graph\ contains\ a\ negative\ cycle.$
  **end if**
**end for**

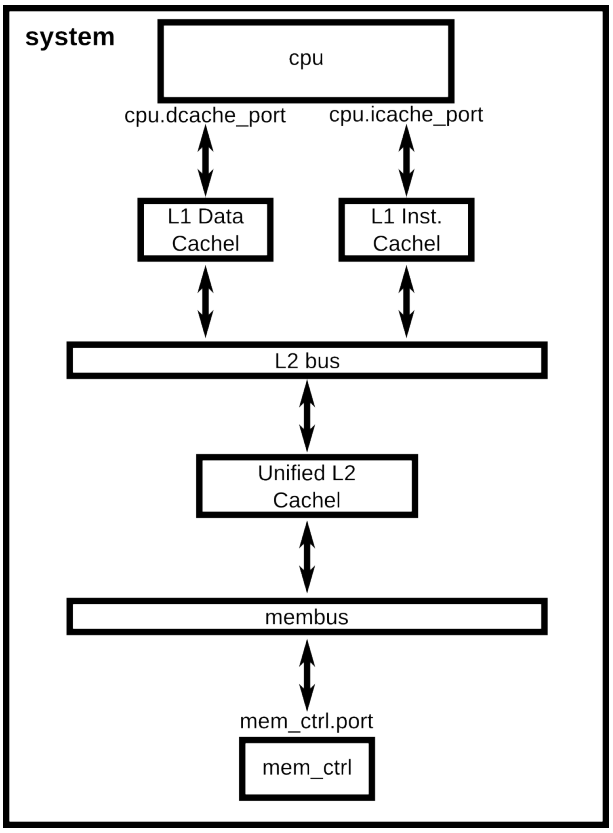



Fig. 1. Gem Se.py simulation structure

ulation and the values for its parameters, a config.json, which is the same as config.ini, but in JSON format, and a stats.txt, which is a text representation of all of the gem5 statistics registered for the simulation. Gem5 has a flexible statistics-generating system. In our paper, we focus on the stats.txt file. The file has valuable information such that the CPU statistics, which contains information on the number of syscalls, statistics for cache system, clock period in ticks, number of CPU cycles simulated and translation buffers, etc. The file also contains some memory controller statistics.

## IV. Results and Analysis

In this section, we will plot some figures and charts to show the results from LLVM, Gephi, and gem5 with the study approaches we mentioned in part III.

### A. results from Gephi & LLVM

After running the applications in LLVM getting the files generated by LLVM, we measure following statistics: nodes and edges, which measures the number of nodes and edges generated from LLVM; assortativity coefficient measures the similarity of connections in the graph with respect to the node degree; diameter is the number of links between the two nodes in the network that are the farthest apart; clustering coefficient measures how well a network decomposes into modular communities; modularity coefficient is a measure of the structure of networks or graphs which measures the strength of division of a network into modules; parallelization degree is a metric which indicates how many operations can be or are being simultaneously executed by a computer, which we provided two numbers calculated by different methods.

Figure 2 shows the growth trend of nodes and edges in the graph after increasing the input size in the applications. In this figure, we can naturally find out that as the input size increases, the numbers of nodes and edges increases as well. However, because of the ability of processing graphs with negative weighted path, Bellman-Ford algorithm needs to traverse the whole graphs with $V - 1$ times, where V is the input size, and it also needs time to check whether there is a negative loop in the graph. In this way, the time and space complexity of Bellman-Ford is much higher than those of Dijkstra algorithm; thus, the number of edges and nodes in Bellman-Ford algorithm is 10 times larger than that in Dijkstra algorithm.

Unfortunately, due to the software and hardware limitation, Gephi cannot process the graphs when input size is greater than 8. Therefore, we take the input size

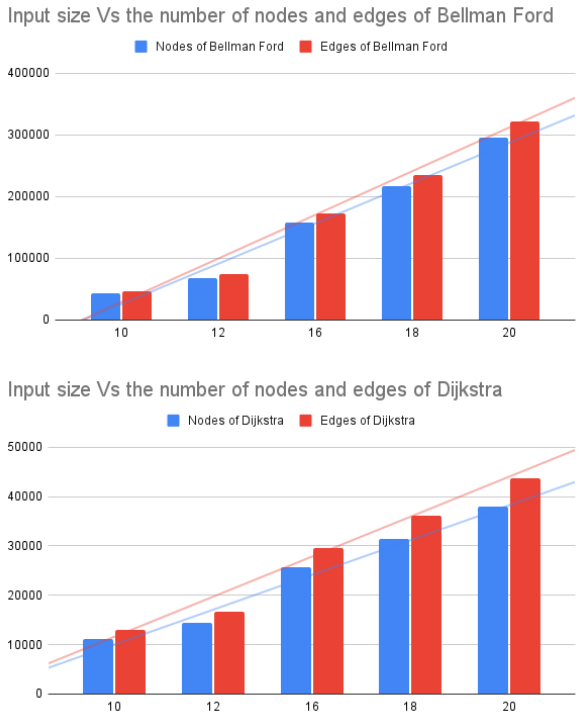



Fig. 2. Growth trend of nodes and edges while increasing the input size

TABLE I
Comparison of different properties of Dijkstra algorithm with different input nodes

| Nodes Undirected Graph Dijkstra's algorithm | | |
|---|---|---|
| Input Nodes | 4 | 8 |
| Nodes | 2093 | 7076 |
| Edges | 2412 | 8202 |
| Assorta -tivity | -0.06 | -0.052 |
| Diameter | 26 | 78 |
| Clustering coefficient | 0 | 0 |
| Modularity coefficient | 0.968 | 0.975 |
| Parallelization degree | 86(networkx) 101(script) | 247 |

of 4 and 8 as the example, processing the graph files generated by LLVM in Gephi, and compare the other parameters mentioned above.

The table above indicates the statistics from a network analysis software, Gephi. In the table I, we use Dijkstra's algorithm to calculate 4 input nodes undirected graph. As the result of LLVM, we obtain 2093 nodes and 2412 edges. From the generated python file, the output of its associativity coefficient is -0.06, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the dijkstra's is 26, also it is the maximum eccentricity. As the modularity coefficient closes to 1,

TABLE II
COMPARISON OF DIFFERENT PROPERTIES OF BELLMAN-FORD ALGORITHM WITH DIFFERENT INPUT NODES

| Directed Graph with negative weights Bellman–Ford | | |
|---|---|---|
| Input Nodes | 8 | 4 |
| Nodes | 14719 | 6322 |
| Edges | 17103 | 7430 |
| Assorta -tivity | -0.06 | -0.048 |
| Diameter | 399 | 157 |
| Clustering coefficient | 0 | 0 |
| Modularity coefficient | 0.963 | 0.96 |
| Parallelization degree | 246(networkx) 564(script) | 106(networkx) 101(script) |

TABLE III
COMPARISON OF DIFFERENT PROPERTIES OF FLOYD-WARSHALL ALGORITHM WITH DIFFERENT INPUT NODES

| Floyd-Warshall | | |
|---|---|---|
| Input Nodes | 4 | 8 |
| Nodes | 3048 | 19530 |
| Edges | 3855 | 24871 |
| Assorta -tivity | -0.076 | -0.054 |
| Diameter | 124 | 724 |
| Clustering coefficient | 0 | 0 |
| Modularity coefficient | 0.965 | 0.985 |
| Parallelization degree | 25 | 58 |

this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others.

In terms of 8 nodes, we obtain 7076 nodes and 8202 edges. From the generated python file, the output of its associativity coefficient is -0.052, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the dijkstra's is 78, also it is the maximum eccentricity. As the modularity coefficient closes to 1, this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others.

As table II shows, we use Bellman-Ford algorithm to calculate 5 input nodes and 8 input nodes with negative weights directed graph. We first discuss the 5 input nodes one. As the result of LLVM, we obtain 6822 nodes and 7430 edges. From the generated python file, the output of its associativity coefficient is -0.048, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the Dijkstra's is 157, also it is the maximum eccentricity. As the modularity coefficient closes to 0.96, this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others. The degree of parallelization is 106 by networkX and 101 by greedy community detection script.

In terms of the 8 input nodes with negative weights directed graph, we obtain 14719 nodes and 17103 edges, which is significant larger than the 5 input nodes one. From the generated python file, the output of its associativity coefficient is -0.06, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the Dijkstra's is 157, also it is the maximum eccentricity. As the modularity coefficient closes to 0.96, this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others. The degree of parallelization is 106 by networkX and 101 by greedy community detection script.

In the table III, we use Floyd-Warshall algorithm to calculate 4 input nodes and 8 input nodes. We first discuss the 4 in put nodes one. As the result of LLVM, we obtain 3048 nodes and 3855 edges. From the generated python file, the output of its associativity coefficient is -0.076, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the Dijkstra's is 124, also it is the maximum eccentricity. As the modularity coefficient closes to one, this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others. The degree of parallelization is 25.

In terms of the 8 input nodes Floyd-Warshall, we obtain 19530 nodes and 24871 edges, which is significant larger than the 4 input nodes one. From the generated python file, the output of its associativity coefficient is -0.076, which large-degree nodes slightly tend to attach to low-degree nodes. The Diameter of the Dijkstra's is 724, also it is the maximum eccentricity. As the modularity coefficient closes to 0.985, this indicates strong community structure. The clustering coefficient is zero; there are hardly any connections in the neighborhood. Many of the load and store instructions is not related to others. The degree of parallelization is 58.

In conclusion, for these three algorithms, as the number of input nodes increases, the numbers nodes and edges of the graphs generated by LLVM increase dramatically, and the network diameters of these graphs

also increases obviously. For example, when the number of input nodes increases from 4 to 8, which is the twice the size of the previous number, the nodes and edges increases to 3 times, or even 5 times of them. On the contrary, the values of clustering and modularity coefficient have few relations with the number of input nodes, which remain in a relatively stable range.

### *B. partition results*

In this paper, we also analysis the shortest path algorithms file using Gem5 with LLVM cluster .ll files, which stands for list and label preview file. We have introduced Gem5's basic information and functionalities in the approach and methods section. In this result section, we will first plot tables to see how the changes of regularization parameters $\lambda$ affect the numbers of the simulated clock cycle as the output of the Gem5.

The figure 2 shows the cluster size vs input size of Bellman-Ford algorithm with different regularization parameters. The input size is 10, 12, 16, 18, and 20. We can observe a pattern of increasing cluster size while input sizes get larger. The top table showing $\lambda 2$ has no effect on with cluster size for different input size while $\lambda_1$ stay the same. And this result is the same as the conclusion proposed by Xiao et. al in 2017.[8] The second table shows both $\lambda_1$ set to 0.5 and another one of the $\lambda_1$ is set to 0 and both $\lambda_2$ set to 0.5. The program with $\lambda_1$ set to 0.5 has significant smaller cluster size than the one with $\lambda_1$ set to 0. However, the third table shows both $\lambda_1$ set to 0.5 and one of the $\lambda_2$ is set to 0.5. The program with $\lambda_1$ set to 0 has slightly smaller cluster size than the one with $\lambda_1$ set to 0. In addition, the fourth table shows both $\lambda_2$ set to 0 and one of the $\lambda_1$ is set to 0.5. The program with $\lambda_1$ set to 0.5 has smaller cluster size than the one with $\lambda_1$ set to 0.

After getting the result from Table IV, which says the $\lambda_2$ has few influences on the cluster size, in the following experiment, we just changed the value of $\lambda_1$ ranging from 0 to 0.9, to observe the changes of cluster size.

Taking $n = 12$ as the input size, Figure 3 shows the correlation between the cluster size and $\lambda_1$, which got from python function, where $R^2 = 0.0.8863$ and $P < 0.0001$, which demonstrates a strong negative correlation between this two parameters.

Besides the community detection scripts provided in the class, we also used modularity-based partition algorithms to get communities with different input sizes.

As Figure 5 shows, like community detection script provided in the cluster, by using the modularity-based partition algorithm, the cluster size increases as the input size increases.

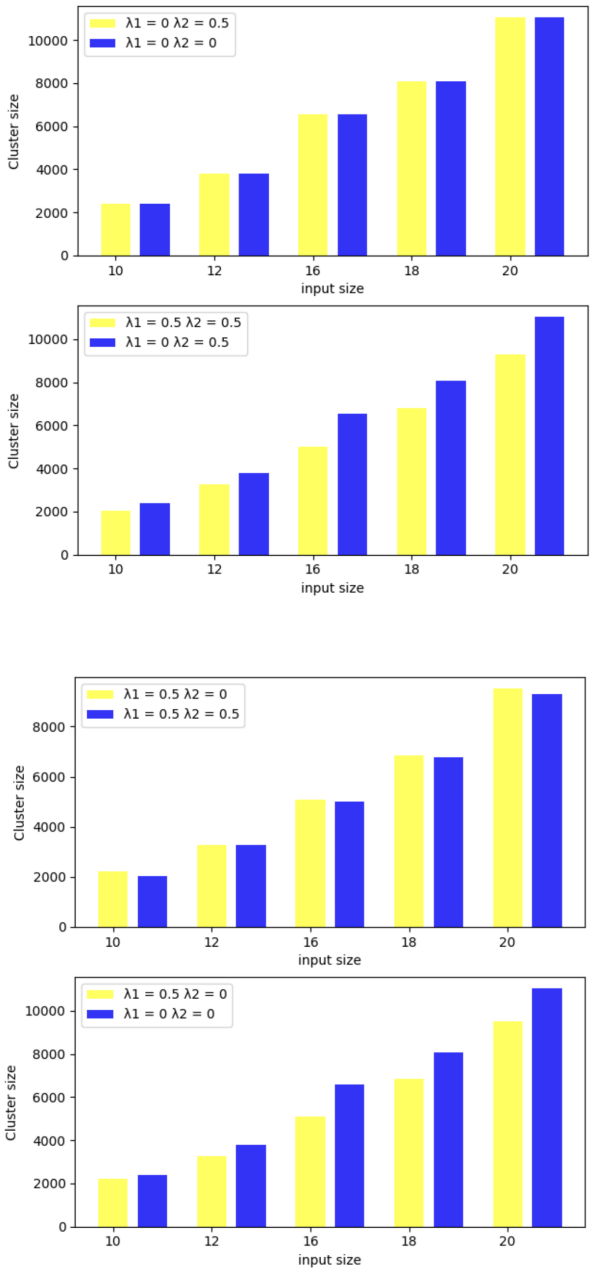
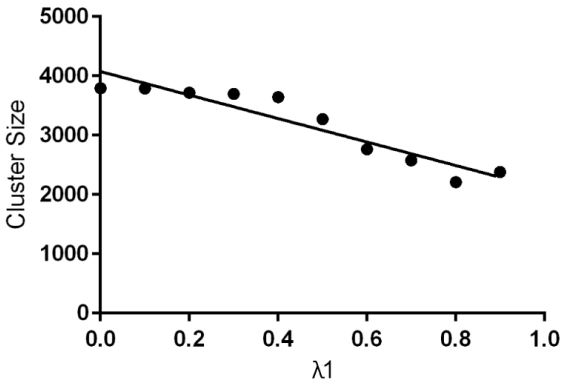



Fig. 3. Comparison of different regularization parameters of Bellman-Ford Algorithm

To further understand how this modularity-based partition algorithm works and the partitioning results, we also compared the number of cluster file generated by networkX with that generated by community detection script. The result is shown in Figure 6. In Figure 6, by taking the Bellman-Ford shortest path algorithm as the example, we set the input size as 12, and $\lambda_2$ as 0.5 and change the value of $\lambda_1$. As the value of $\lambda_1$ increases, the cluster size decreases generally. However, it is notable

Fig. 4. Cluster size vs $\lambda_1$ of Bellman-Ford Algorithm



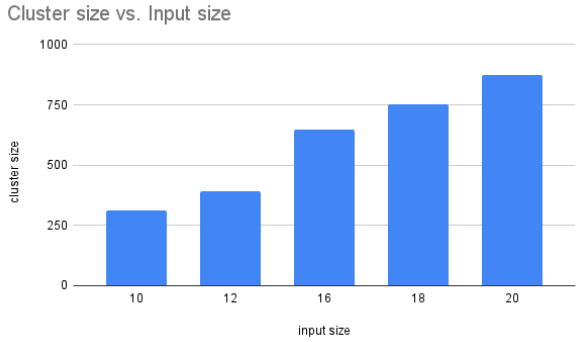




Fig. 5. Cluster size vs. Input size based on modularity partition algorithm

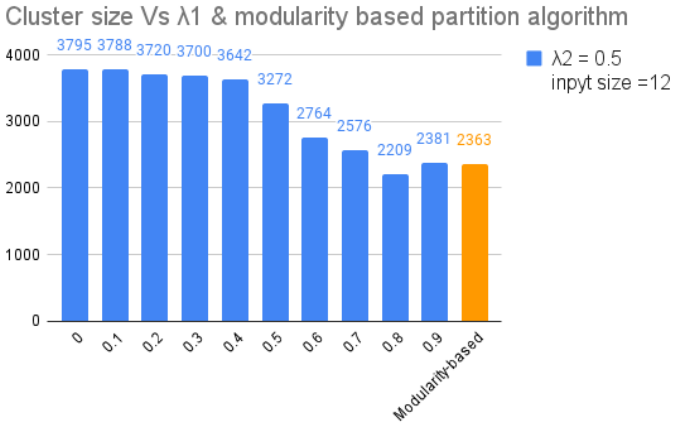



Fig. 6. The comparison of cluster size generated by community detection and modularity based partition algorithm

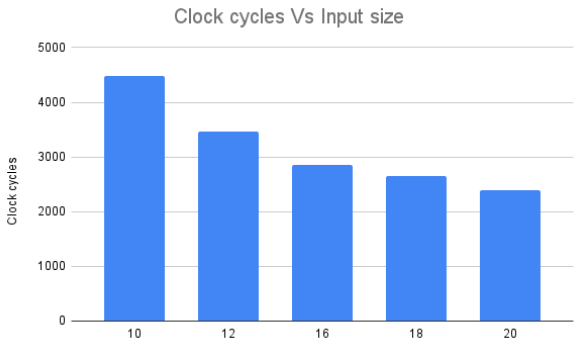



Fig. 7. Clock cycles vs input size of Dijkstra algorithm partitioned by of networkX

that the value of cluster size is a little bit higher for $\lambda_1$ becomes 0.9 than for $\lambda_1$ equals to 0.8, and the value of cluster size calculated by networkX is similar to the value at $\lambda_1$ is 0.9. This remarkable observation will also be discussed in the next subsection on the relationship between computation time and cluster size

### *C. virtualization results from Gem5*

In this subsection, we will illustrate the research results running from gem5, in which we changed the parameters, such as importing different numbers of cluster files as the number of cores during the running process, and changing the value of L1 data cache, and the CPU type, in gem5 to simulate different running environment for our applications.

Firstly, we tried to find out the importance of parallel files while running gem5. Normally, as the input size increases, the computation time will increases as well. However, as we mentioned in subsection IV-B, the cluster size will also increases when the input size increases, in this way, more parallel files will be generated. This redundant parallel files causes the number of cores while running the application increases, and this increasing value makes the clock cycle, the representative of computation time, decreases. Figure 7 shows the decreasing trend of clock cycle while we increase input size but also increase the number of cores at the same time. The number of cores is generated by modularity-based partition algorithm, implemented by networkX.

By putting different clustering files in the parallel folder, which we obtained from the LLVM by changing the value of $\lambda_1$, we also found that the value of clock cycles, which can be the representatives of execution time, is also changed while the size of cluster changes.

Figure 8 shows the comparison of clock cycles with different sizes of parallel files generated with different value of $\lambda_1$ in community detection script and the parallel files generated in modularity-based partition algorithm. The increasing trend of the clock cycles are the same as the decreasing trend of the number of parallelization degree, and there is also a peak in Figure 8 at $\lambda_1 = 0.8$, which matches the valley in Figure 6. The

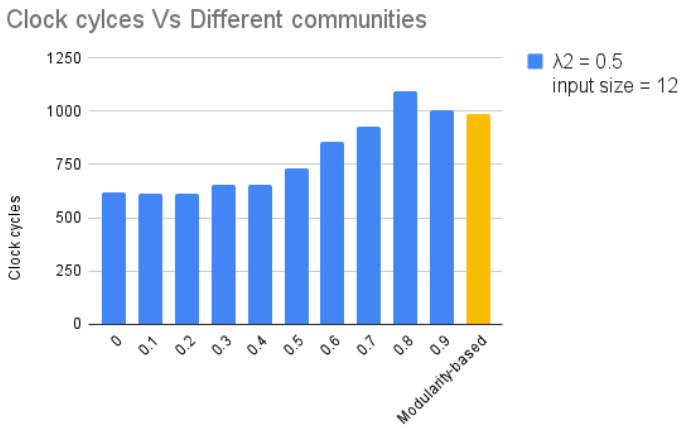



Fig. 8. The comparison of clock cycles with parallel files generated by community detection and modularity-based partition algorithm

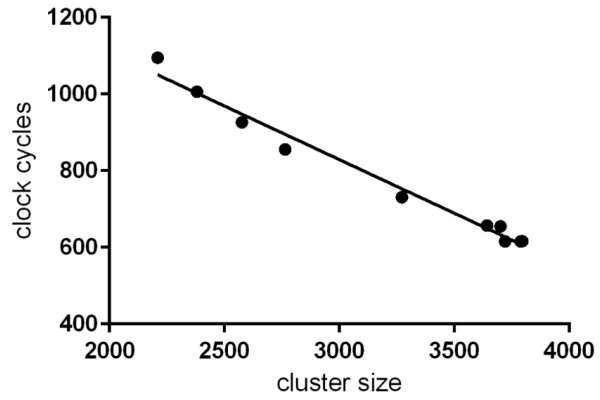



Fig. 9. Clock cycles vs in size of Bellman-Ford Algorithm

computation times are about the same if we set the $\lambda_1$ as 0.9 or use the modularity-based partition algorithm. Through this two mentioned figures, we could conclude that the value of cluster size influences the computation time, regardless of the partition algorithms we used. The larger the cluster size is, the faster the application runs in gem5.

Figure 9 shows the correlation between the value of clock cycles and the size of the cluster, where $R^2 = 0.9818$ and $P < 0.0001$, which shows that there is a strong negative correlation between the cluster size and the execution time. And this result implies that the higher the parallelization degree is, the faster the application runs. This result is consistent with the previously mentioned results.

In the project, to test the performance of the shortest path algorithms on the Gem 5 simulation, we experiment with three main comparisons and obtained some interesting results from them. They are computation time vs cores, computation time vs L1 Cache size, and computation time vs CPU Type. These results and figures are shown in the following section.

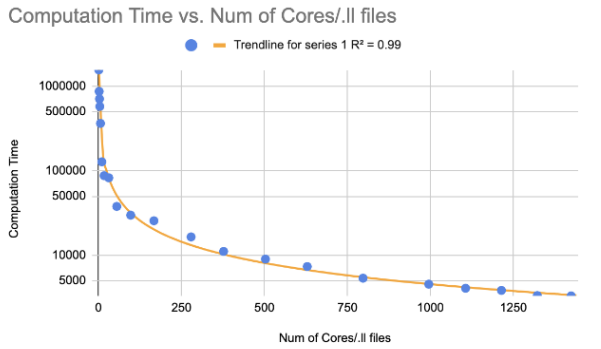
We test three groups of the number of cores with Dijkstra's algorithm with an input size of 16 nodes. For each group, we changed the $\lambda$ value to expect some performance difference between them. For the first group, Dijkstra's algorithm with $\lambda_1 = 0$ and $\lambda_2 = 0$ with an input size of 16. The result shows an exponential decrease in computation time while the numbers of cores increase. The plot diagram is log scaled at the Y-axis.

For the second group, we changed the $\lambda$ value to expect some performance difference between them. Dijkstra's algorithm with $\lambda_1 = 0$ and $\lambda_2 = 0.5$ with an input size of 16. The result shows an exponential decrease in computation time while the numbers of cores increase. To compare the first group and the second group, the result shows there is no significant difference between different $\lambda_2$. That is $\lambda_2$ does not affect the computation time of the shortest path algorithm. The plot diagram is log scaled at the Y-axis.

For the third group, we changed the $\lambda$ value to expect some performance difference between them. Dijkstra's algorithm with $\lambda_1 = 0.5$ and $\lambda_2 = 0$ with an input size of 16. The result shows an exponential decrease in computation time while the numbers of cores increase. To compare the second group and the third group, the result shows there is no significant difference between different $\lambda_1$. That is $\lambda_1$ does not affect the computation time of the shortest path algorithm. The plot diagram is log scaled at the Y-axis.

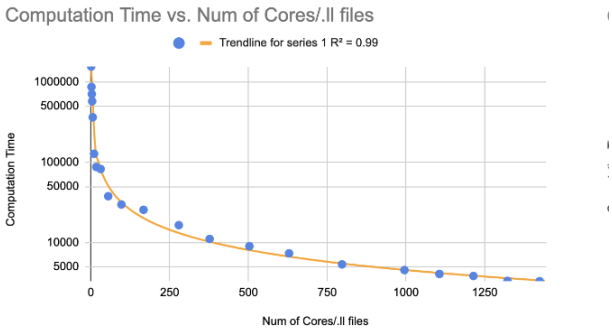
To conclude the number of cores and computation time, we observed an exponential decrease in computation time while the numbers of cores increase. This means a larger number of cores can reduce the computation time significantly; however, the computation time will converge at a line of a point. In the real world, we might have a trade-off between the number of cores and the reduction of the computation time. Furthermore, three groups with different $\lambda$ values have no different performance outcomes overall. And we observed there is not a significant effect on the computation time with $\lambda_1$ and $\lambda_2$ values.

In the project, we obtained a result of computation time vs L1 Cache size. The L1 Cache size affects the computation time of the Gem5 results running the shortest path algorithm. To test the result of the computation time, we run both the L1 data cache size and L1 input cache size from 1 KB to 256 KB. The result is shown in figure 13 below, the result shows an exponential decrease in the cache size increases. The exponential decrease converges around 2000 in computation time. That is,



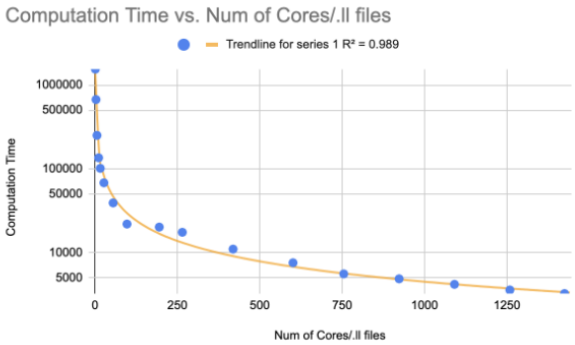

Dijkstra algorithm with λ1= 0 and λ2 = 0 with input size 16

Fig. 10. Number of the cores and Computation time




. Dijkstra algorithm with λ1= 0 and λ2 = 0.5 with input size 16

Fig. 11. Number of the cores and Computation time




Dijkstra algorithm with λ1= 0.5 and λ2 = 0
with input size 16.

Fig. 12. Number of the cores and Computation time

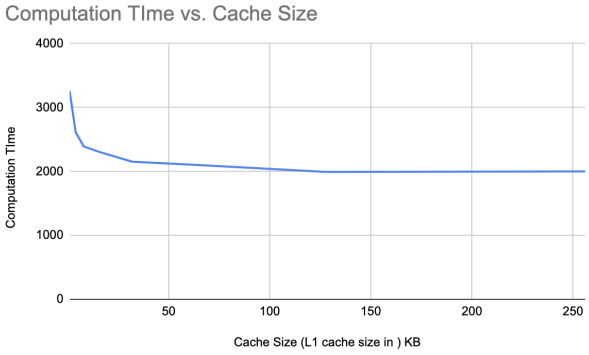



Fig. 13. L1 Cache Size and Computation Time

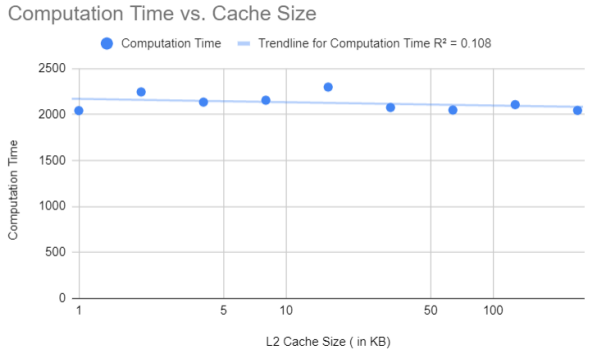



Fig. 14. L1 Cache Size and Computation Time

when the cache size is increased from 128KB to 256KB, there is a significant difference in the computation time between the two cache sizes.

In the project, we also obtained a result of computation time vs L2 Cache size. The L2 Cache size affects the computation time of the Gem5 results running the shortest path algorithm. To test the result of the computation time, we run both the L2 cache from 1 KB to 256 KB. The result is shown in figure 14 below, the result shows no difference in the cache size increases. That is, when the cache size is increased from 1KB to 256KB, there is a no significant difference in the computation time between the cache sizes.

In the experiment, we compare three different CPU types: DerivO3CPU, TimingSimpleCPU, and O3CPU with Dijkstra's algorithm. TimingSimpleCPU is a CPU type that memory requests actually take time to go through to the memory system and return. DerivO3CPU inheritance from O3CPU in Gem5. In figure 15 below, we have compared the three CPUs (DerivO3CPU, TimingSimpleCPU, and O3CPU) computation time with the number of cores, which is the number of parallel files on the side of the parallel fold in this case. The plot diagram is log scaled at both the X and Y axis. From the

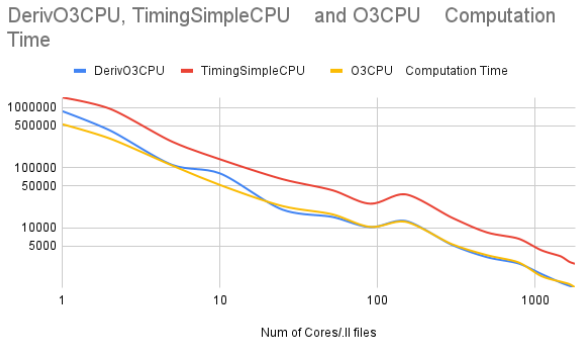



Fig. 15. Three CPUs with Dijkstra Algorithm

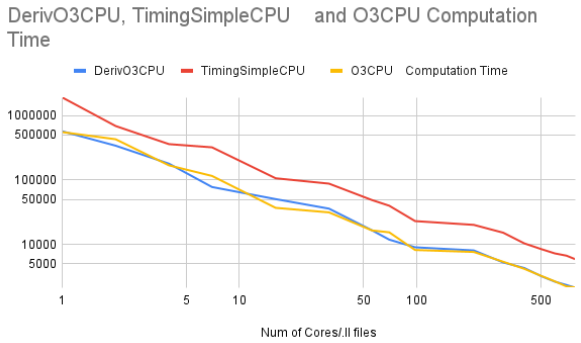



Fig. 16. Three CPUs with Bellman-Ford Algorithm

result, the TimingSimpleCPU is the slowest one among the three CPUs. DerivO3CPU performance is slightly better than O3CPU when the number of cores is less than 20 however, there is no significant difference between the two CPUs when the number of cores is more than 20. For all three CPUs, the computation time is decreased while the number of cores or the number of parallel files increases.

In The following experiment, we compare three different CPU types: DerivO3CPU, TimingSimpleCPU, and O3CPU again with the Bellman-Ford algorithm as well. TimingSimpleCPU is a CPU type that memory requests actually take time to go through to the memory system and return. DerivO3CPU inheritance from O3CPU in Gem5. In figure 16 below, we have compared the three CPUs (DerivO3CPU, TimingSimpleCPU, and O3CPU) computation time with the number of cores, which is the number of parallel files on the side of the parallel fold in this case. The plot diagram is log scaled at both the X and Y axis. From the result, the TimingSimpleCPU is the slowest one among the three CPUs. O3CPU performance is slightly better than DerivO3CPU when the number of cores is less than 20; however, DerivO3CPU performance is slightly better than O3CPU when the number of cores is from 12 to 50. There is no significant difference between the two CPUs when the number of cores is more than 20. The result we observed from both Dijkstra's algorithm and the Bellman-Ford algorithm is very similar in that the TimingSimpleCPU has the worst computation time while O3CPU and DerivO3CPU perform similarly. For all three CPUs, the computation time is decreased while the number of cores or the number of parallel files increases.

## V. Conclusion

In Milestone 4, We tried several different partition algorithms provided in the class or in networkX to get different parallel files and parallelization degree. After getting these parallel files, we have compared the resulting number of cores with computation time. We observed an exponential decay in computation time with the increasing number of cores. For the simulation environment, we tried to change the CPU type and the size of cache to find the variety in performance. We have compared three different CPU types for both Dijkstra's algorithm and the Bellman-Ford algorithm. Both O3CPU and DerivO3CPU have better performance than SimpleTimeCPU. We tested the shortest path algorithm with different partition algorithms. For future work, we could find more real-world application input sets for the shortest path to test on the LLVM and Gem5 with a variety of parameters.

Potential plan for future improvement:

- Although Dijkstra's algorithm is efficient and universal for most fields to calculate the shortest path problem, an optimized solution such as using different data structures to reduce the time complexity or space complexity.
- Find an optimal solution for calculating the shortest path in a real-world application such as the Anaheim Road Network Problem we addressed in the milestone 4 presentations.
- Test the application on the network-like data set and obtain testing results using the Gem 5 simulation.

## References

[1] B. Popa, D. Selisteanu, A. E. Lorincz, and T. Robert, "Optimization possibilities for the shortest-path algorithms in the context of large volumes of information," in *2022 8th International Conference on Control, Decision and Information Technologies (CoDIT)*, vol. 1, 2022, pp. 361–366. DOI: 10.1109/CoDIT55151.2022.9804024.

[2] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah, "A survey of shortest-path algorithms," *arXiv preprint arXiv:1705.02044*, 2017.

[3] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[5] L. R. Ford Jr, "Network flow theory," Rand Corp Santa Monica Ca, Tech. Rep., 1956.

[6] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation," *Communications of the ACM*, vol. 31, no. 11, pp. 1343–1354, 1988.

[7] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[8] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan, "A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2017, pp. 217–224.