## Design *discussed with Paul, Jack

I have broken up the software system into different modules. Each module will have a **goal**, **input**, **output**, and **implementation details**. These modules will all be written as C functions, so we can reuse I/O on the two versions of our algorithm. The Test Module will be written in Python.

customer_software_{serializability} is an executable that takes in command line arguments about the name of the input file name and the number of threads to be used. Eg. a valid call would be "./customer_software_serial sample_input_1.txt 128". This will make the testing infra easier. customer_software_{serializability} will be constructed using the following modules.

| Customer Software | Test |
|---|---|
| **customer_software_serial**<br>**Modules (C Functions) used:**<br>Input Module → FW (Serial) → Output Module<br>**Executable behavior:**<br>takes in 2 command line arg as seen above, and outputs a file with the corresponding name eg. sample_output_serial_1 | For **correctness** we check sample input files against expected output files.<br><br>For **speed**, we simply run the executables with a permutation of the experiment variables. |
| **customer_software_parallel**<br>**Modules (C Functions) used:**<br>Input Module → FW (Parallel) → Output Module<br>**Executable behavior:**<br>same as serial, but outpuss sample_output_parallel_1 | |

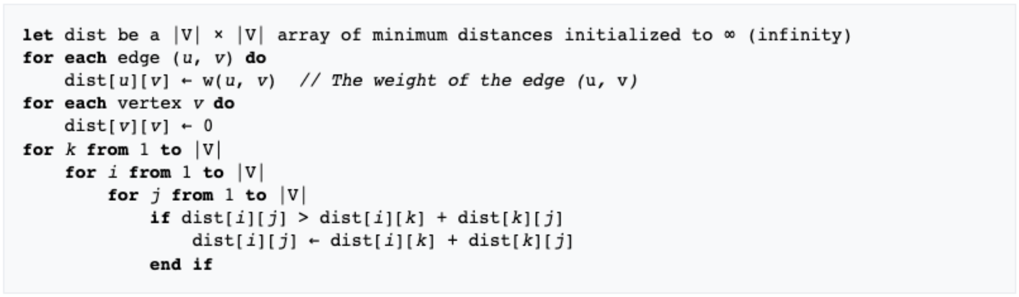
## Input Module

**Goal:** Read file to input and pass on an adjacency matrix to the FW Module
**Input:** input filename, thread count
**Output:** Memory-allocated array that is the adjacency matrix, the number of desired threads T
**Side Effects:** None
**Implementation Details:**

- Note, command line arguments of the shipped executable can only be read from main, so that's where we'll get input filename and thread count from. We will then call this input module, passing in filename and thread count as arguments.
- Input filenames will be of the form "sample_input_1.txt". This makes testing easier.
- The file input will specify that there is no edge from u to v if the corresponding edge weight is infinity, denoted by the character 'i'.
- We parse the file as follows. We scan in the first line to know what the thread count is going to be. In the case of the serial version, this number will be 0. We then scan the second line to know what N = |V| is, then we will proceed to fill out each row of a **2-dimensional int array** called adjacency matrix **whose memory we allocate here** by reading in each subsequent line, where we split the line by the space delimiter. We can then use a string to int function like atoi to convert the weight for each edge we see for that particular line, or set it to an infinity value if we see the char 'i' (a discussion on what to choose for infinity value comes later)
- We **will not handle only positive edge (non-zero, non-negative) weights**. This will simplify our project to focus on parallelization. The max edge weight is 1000.
- Let us set infinity to 1,023,000 + 1. This is because the max weight of a path is 1000 * number_of_nodes_in_a_path. We know a path from u to v does not visit the same node twice (if it did, then positive weights imply that cycle was positive, making the more straightforward path cheaper). Thus, the maximum path length is N nodes, or N - 1 edges. Thus,
- MAX_PATH_WEIGHT = 1000 * (N - 1) = 1000 * 1023, since max N given=1024
- Since 1,023,000 + 1 > 1000 * 1023, we know if we see infinity = 1,023,000 + 1 in our algorithm at the end, no path was found thus we should output that there was no path.
- this infinity is good because it doesn't risk overflow if the algorithm adds to it, which it does in the comparison step
- Sample input:

```
128
3
0 1 2
3 0 4
5 i 0
```

## Algorithm (Serial) Module



```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

**Goal:** Run FW serial and output a matrix representing the shortest path
**Input:** Memory-allocated 2d N x N matrix adjacency matrix that represents the shortest path
**Output:** None
**Side Effects:** Adds an alias of the given adjacency_matrix as dist (i.e. renames). Additionally, update the matrix's value to do the algorithm above.
**Implementation Details:**

- First, note that we are OK to use the adjacency_matrix as our starting dist matrix. Doing so is equivalent to the initialization step of FW (lines 1 - 5 of the snapshot above
- We implement basically as above, starting our **timer** just before the outermost for loop and end it just after it

## Output Module

**Goal:** Writes dist to file
**Input:** Memory-allocated 2d N x N matrix dist that represents the shortest paths, name of input file, boolean indicating if dist was created using the parallel or serial version.
**Output:** text file
**Side Effects:** frees memory of the dist array

**Implementation Details:**

- We will generate the **output** by scanning through dist and writing its row values to an output file. Here's how it works. We scan one row of dist. We initialize an empty string to represent the output that should correspond to this row, i.e. the shortest path to get from the node of the current row to all other nodes. Call this string **output_row_weights**. Now, we iterate through the row of dist since it is an integer array, and convert the integer weight to a string with the itoa library function. We then append this onto the end of output_row_weights. If we ever encounter a value equal to infinity, we simply write 'i' to represent that there is no path. We then write **output_row_weights** to file and continue
- To make the testing framework easier, we'll name the output files with the following convention: "sample_input_1.txt" → "sample_output_serial_1.txt" or "sample_output_parallel_1.txt". We can grab the number from the main file where we

called the input module, and we can grab whether or not it's serial/parallel from the same main file that read in the command line argument.

- We will free memory of the dist at the end

## Algorithm (Parallel) Module

**Goal:** Run FW parallel and output a matrix representing the shortest path
**Input:** Memory-allocated 2d N x N matrix adjacency matrix that represents the shortest path, **in addition to T,** the number of threads to run.
**Output:** None
**Side Effects:** Adds an alias of the given adjacency_matrix as dist (i.e. renames). Additionally, update the matrix's value to do the algorithm above.
**Implementation Details:**

- First, note that you could've parallelized the matrix initialization in the input module, but our emphasis is parallelizing the triple for loop
- After thinking about it, I realize **you cannot parallelize on k, but you can on i and j**. I.e. for a given k, you can cut up the 2d matrix however you like and divide work among threads. For a discussion of why this doesn't lead to race conditions, see the appendix.
- I will **choose to parallelize on i** (the rows), not j. I think it makes handling indices easier when you know a worker gets a specific row, versus a sub-matrix.
    - Note that this means there will be times when T > N. Eg. N = 32, T=64, then dividing only on i would leave 32 threads idle. My implementation will not code for this case because if the thread count exceeds the row count, then the input size is so small that you'd likely not get an appreciable speed up. In other words, **threads_used = min(T, N)**
- **We will spawn the threads before the triple for loop** so that we don't constantly spawn threads for every k. We then put a barrier at the end of the k for loop (within the loop, not after it), so that all threads are done with a particular k before we move on. At the very end when we exit the k for loop, we'll call join.
- We will divide up the work statically. Since the work assigned to each thread is predictably the same (unlike the primality test), there's no need to dynamically allocate the work. Eg. Suppose N = 512, T = 16. Then each thread should get 32 jobs, and we can assign t_1 to rows 0 to 31, t_2 to rows 32 to 63, etc.

## Test Module

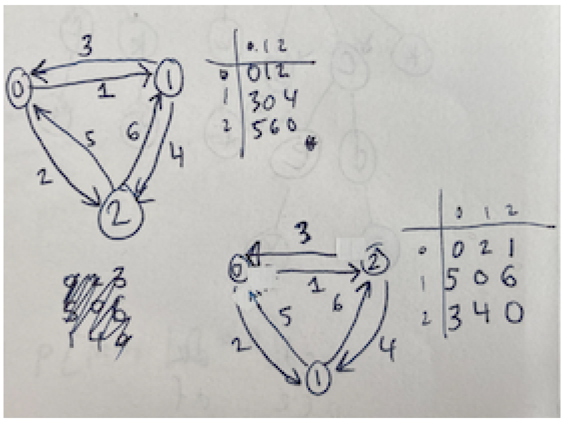
**Goal:** Test correctness of our algorithms
**Input:** the output txt files from running the serial and parallel version of the code
**Output:** An assertion for whether the experimental output matched the expected output

**Side Effects:** None

**Implementation Details:**

- I plan to (manually) come up with three N=3 test inputs that I can manually verify for my serial implementation. One of them will contain an isolated node, another will contain some infinity edges, and the last one will contain no infinity edges. If that passes, I then plan to run my serial algorithm on a decently-sized input where hand-checking wouldn't make sense, say N = 10. I would generate this graph via an RNG.
- **Swapping Technique:** In this case, I wouldn't be able to manually verify the results, but I could do a sanity check with the following trick. I will take two nodes v_1 and v_2, and swap their labels. This corresponds to swapping rows v_1 and v_2, then swapping columns v_1 and v_2.

  For example, if I were to re-label nodes 1 and 2 in the following (0-indexed) graph:

  You can see that the structure of the graph does not change, so our output should not meaningfully differ except that the rows and columns of 1 and 2 be swapped.

  ```
  output
  0 1 2
  3 0 4
  5 6 0
  ```

  ```
  output (swapped)
  0 2 1
  5 0 6
  3 4 0
  ```
- I will do this row swapping technique by creating my 10 x 10 input file, called sample_input_4.txt, then comparing it to sample_input_5.txt and sample_input_6.txt where I will have swapped two rows/columns

## Performance Hypotheses

**Parallel overhead. For T = 1, N = 16, ..., 1024.**

The ratio of serial to parallel will start above 1 and converge to 1. This is because you are essentially running the same program in both cases, just that in the parallel case you have the overhead of spawning a thread which makes it take longer. However, this overhead is fixed since you are only spawning one thread. Thus, the proportion of this spawn time compared to the work running the algorithm gets smaller as the work of the algorithm gets larger with N. Thus, we expect parallel runtime to get closer to serial run time.

**Parallel Speedup. For T = 2, ..., 64. For N = 16, ..., 1024.**

Fix N. I predict the program will run faster as T increases, up to the point that increasing T doesn't put more threads into use (N = T), at which point the time should stay constant because my implementation doesn't spawn unused threads.

The time decrease should be proportional to the number of workers. Eg. I have double the workers, so my time should decrease by half since an additional worker evenly divides up the work more. This is because I don't anticipate any one worker having a much harder time than the others (unlike the primality testing exercise) because the worker's operations are all the same, no matter what slice of the (k, i, j) problem space they get. Thus, your speed up is directly proportional to the number of workers you have. Maybe it is slightly less than proportional because of the overhead of spawning a thread, but the trend should then be more evident on a larger input size where thread spawn overhead counts for less.

I do wonder if the overhead cost may outweigh spawning additional threads. Eg. let T = 32 or 64 and N = 128. Then even though each thread can get assigned one row of work, it is not clear that taking the time to spawn a new thread + managing it with joins will be faster than just letting another thread chip at the work. I do not know honestly, so we'll see.

## Appendix

**Here’s why you cannot parallelize on k, but you can on i and j**

- We have the following recurrence

$$\mathrm{shortestPath}(i, j, 0) = w(i, j)$$

and the recursive case is

$$\mathrm{shortestPath}(i, j, k) = \min\Big(\mathrm{shortestPath}(i, j, k-1), \mathrm{shortestPath}(i, k, k-1) + \mathrm{shortestPath}(k, j, k-1)\Big).$$

-

**We can’t parallelize on k** due to the nature of the algorithm. The way that this dynamic programming problem works is to say “for a given k, what is the shortest path from node i to j using nodes {0, 1, ... k}?”. The answer is then based on the answer to a smaller k, as seen in the recurrence relation. What this means is that if you were to parallelize computation on different k’s, then the thread t_2 with k = 2 would have to wait on the work of the thread t_1 with k = 1.

Concretely in code, parallelizing on k would be dangerous. Suppose we have the same case as above. Then if the thread t2 were executing an instruction and doing the comparison

```
if dist[i][j] > dist[i][k] + dist[k][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
```

Then this presupposes that dist[i][k] on the RHS indeed represents shortestPath(i, k, k - 1) as written in the recurrence relation. But if t1 has not gotten there, then clearly the cell dist[i][k] does not do that, thus we fail the recurrence relation.

**We can parallelize on i.** We know we must have all threads operate on the same fixed k=K, so how else can we parallelize the work? I claim you can parallelize on i. Suppose you have two threads t_1 with i = 1, t_2 with i = 2.
Now, what subproblems does each thread tackle and do they overlap?

t_1 tackles k=K, i = 1 for j = 1 to N.
t_2 tackles k=K, i = 2 for j = 1 to N.

Looking at the lines where we may have a read/write overlap,

```
if dist[i][j] > dist[i][k] + dist[k][j]:
```

dist[i][j] = dist[i][k] + dist[k][j]
It seems that we may be safe. After all, here's what these two lines look like for each thread

| t_1 | t_2 |
|---|---|
| if dist[1][j] > dist[1][K] + dist[K][j]:<br>        dist[1][j] = dist[1][K] + dist[K][j] | if dist[2][j] > dist[2][K] + dist[K][j]:<br>        dist[2][j] = dist[2][K] + dist[K][j] |

Note how even though both threads access dist[K][j], this is allowed because they are both reading the value, not writing it.

The potentially problematic part comes up for thread k, i.e.
t_k tackles k=K, i = k, j = 1 to N.

| t_k |
|---|
| if dist[K][j] > dist[K][K] + dist[K][j]:<br>        dist[K][j] = dist[K][K] + dist[K][j] |

Here, you can see that t_k could potentially be writing to dist[K][j] **while other threads are reading from it!** This would be bad because it means that the other threads would be looking up a value that they don't want. The other threads want dist[K][j], which mathematically is shortestPath(K, j, K - 1). But by writing to the cell, you update that cell to represent shortestPath(K, j, K), which is not the recurrence.

However, if you look closely, you will see that t_k will never overwrite dist[K][j]. This is because the conditional that triggers the write only happens when dist[K][K] **is negative**. Since we assume no negative edge weight cycles, this never happens! Thus, we are safe to parallelize on i.

**By a similar argument, we are safe to parallelize on j.** This is basically dividing up the work of going through the ith row. Suppose we have the following for some fixed K, I.

t_1 tackles k=K, i = I for j = 1 to N / 2.
t_2 tackles k=K, i = I for j = N / 2 + 1 to N.

| t_1 | t_2 |
|---|---|
| if dist[I][j] > dist[I][K] + dist[K][j]:<br>        dist[I][j] = dist[I][K] + dist[K][j] | if dist[I][j] > dist[I][K] + dist[K][j]:<br>        dist[I][j] = dist[I][K] + dist[K][j] |

| | |
|---|---|
| for j in [1, N/2] | for j in [N/2 + 1, N] |

Again, both threads read dist[K][j] but never write to it, sowe're safe. The program variable dist[I][j] they write to is neatly separate for these two threads.

Again, the potentially problematic part comes up for the thread that writes to j = K, call it t_k.

| t_k |
|---|
| at one point, the conditional will look like<br>if dist[I][K] > dist[I][K] + dist[K][K]:<br>   dist[I][K] = dist[I][K] + dist[K][K] |

You can see that dist[I][K] may be written to in t_k, even though it is read from in the other threads (see t_1, t_2). Again, this is a false alarm because this write only happens if dist[K][K] is negative, which is never true.