

<https://docs.rs/plotters/latest/plotters/>
<https://github.com/plotters-rs/plotters/blob/master/plotters/examples/normal-dist.rs>
<https://www.geeksforgeeks.org/shortest-path-unweighted-graph/>
<https://doc.rust-lang.org/std/iter/struct.Zip.html>
<https://github.com/plotters-rs/plotters/blob/master/plotters/examples/histogram.rs>
<https://users.rust-lang.org/t/understanding-the-trait-std-sum-t/89161>

2. Written Report

A. Project Overview

- **Goal:** My goal for this project is to analyze the distribution of the average node distance (through closeness centrality) for a large california roadmap graph. I got my project from SNAP. There are about 2 million nodes (slightly less) and about 5 million edges on the project. The graph is a directed graph, where it is stored in a text file with the left hand side being from ID and the right being to ID. I then also make a function to calculate the shortest path length between two nodes. I then visualize my data in the form of a histogram, whilst also finding some interesting outliers and addressing them in my writeup + try to understand them. I find both the closeness centrality, which I print out some metrics of and find the average closeness centrality. I then graph the inverse closeness centrality, which is the average path length, which I find to be an more interesting metric to graph, and clearly shows the outliers near 0, which I address later on.
- **Dataset:** <https://snap.stanford.edu/data/roadNet-CA.html>, it is within my project file titled roadNet-CA.txt. It is 87.8 Mb when I process it. I create another file during my analysis (to not have to rerun a multi-hour long program). That file is 50.8 Mb and included with my project on github, there is an explanation in the code and below of why I do this and why it's better as well!

B. Data Processing

- I used knowledge of past reader functions plus the help of chatgpt (which I have cited below) to read my dataset into rust. The dataset was organized with the left side being FROM ID nodes, and the right side being TO ID nodes. After chatting with Professor Leonidas about my graph, he helped me understand that there were a few gaps in my graph, so that it was not totally dense. This would have been problematic for actually doing what I wanted to do, which was closeness centrality, distance from one node to another, and maybe even try strongly connected components.
- One important manual data cleaning decision I made was to remove the first few identifying lines of data. This included the graph size and some instructions that were denoted out. I have pasted them above for important identifying information for the graph.

- He suggested that I implement a few lines of code that would remap my graph to a new IDs and then store that remapped data in a hashmap for later reference. I do the standard reader function methods, including cleaning the first few lines out, which were documentation about what the graph was (although I removed those just for a clean text file). I then matched the from nodes, and the to nodes to the first and second part of the split line, because they were separated by a space. I then check if my HashMap has the from node or the to node already. If it does not, I insert it with my counter, which is essentially remapping it, and increment my counter by 1. Then I resize my graph, which will become an adj list of `Vec<Vec<u32>>` and push the from and the to nodes onto that. I then return a graph struct that stores all of this output data.
- I have to process some more data that I create later on, but that will be in the section below this and is more so associated with a later part of the project, so I am putting it with the rest of the project description.

C. Code Structure

1. Modules

- I have a few modules that are for ease of use and reading for my code. My reader function is one module. My closeness centrality calculations + writer is another module. This is the bulk of my code and I found it best to split it into it's own module. I then made a `closeness_centrality_translator` which takes the output of my closeness writer, which is in the form of a txt file with two lines of data separated by a space, the first being the node id (remapped) and the second being the closeness centrality calculation (I find closeness centrality a little different after talking to professor leonidas, where my calculation is basically average distance length to all the other nodes, so therefore the smallest values are going to be the most connected, or the most central in reference to other nodes), I then also do the standard `closeness_centrality` by inverting this calculation. This is a lot of text and as such I found it best for it's own module. My `plotters` function, which just does a histogram of the output of the `closeness_centrality_translator` is in another module, which I found best practice because it is doing something distinctly different, which is visualizing the output of the code. The last module that is related here is the visualization module, which uses mappers and creates a png histogram for me. The last module is a shortest path between two nodes implementation (which is just a modified BFS!)

○

2. Key Functions & Types (Structs, Enums, Traits, etc)

- For each non-trivial item, restate:

- **Purpose**
- **Inputs/Outputs**
- **Core logic and key components**

Functions:

- **reader::prep**
 - The purpose is to read in the data from a text file, where there are two columns of data split by a space, and output a struct with an adjacency list and more. The left column is the FromID and the right is ToID. It is a directed graph. I want to get the reader into a `Vec<Vec<u32>>` and then output that in the form of a struct.
 - We take in a path that is a `str` of the actual title of the file, and we output a struct where we have the graph, the nodes count, and the two hashmaps, one with old to new, and one with new to old.
 - The core logic is that we read in each line, then split the line by the space in the middle, then we make the left part the from, and the right side too. We then process them and check if they're in our hashmaps, if they're not we add them, with a counter. The counter is so important because it is the new remapping of the hashmaps. We remap because the graph had a few gaps that would have been bad, so after chatting with professor leonidas we changed it. We then return this in the form of a struct for ease of use and reference
 - The struct: graph, nodes, new_to_old, old_to_new
- **Closeness - BFS implementation**
 - The BFS implementation is very similar to that in class and the in notes do much of it justice, but the purpose of the bfs is to calculate the Breadth and return the vector for a single value, this is then scaled up with later functions, but for right now that is all that it does.
 - In / out: it takes in a graph reference which is a `&Vec<Vec<u32>>`, and a start value which is a `u32`. The output is a `Vec<Option<u32>>` which stores the distance to the other paths in the graph.
 - Basically for BFS we make a dist tracker which is what we're appending the nodes to, and a deque, which tracks our progress. We make the start value our start, and push it onto our queue. We then while match and make our current distance, and iterate through the neighbors of our current node. We then update and pushback, and then eventually return a vec that stores the path length to each node in the graph.
- **Closeness - closeness**
 - The purpose of this is to take our BFS output from a specific point, and then unwrap and sum those distances. Then we divide by the graph length minus 1 to get an inverse closeness centrality, or average path length for that specific node.

- Takes a graph and a node and returns a result that stores the average path length for a specific node.
- This is one of the most important functions, along with the one below it. We first make our calc, which is just a bfs run on our input node. Then we make a total distance tracker. Then we iterate through the BFS output variable and match it to unwrap it, making sure that we don't add it to itself. Then we basically add it to the total tracker. We then just return the total distance divided by the graph length - 1 to get average path length.
- **Closeness - parallel**
 - This is where it all comes together. I had to use the help of chatgpt to transform my non-paralleized function into a paralleized function and that is cited and explained below. The purpose of parallel is to run the closeness function on each node in the graph, and then store it and output it. That means that it is calculating the average path length to other nodes the amount of times the graph is long. This means that it did BFS + closeness ~ 2 million times. I ran this overnight and it took more than four hours, and probably closer to six ish hours.
 - Parallel takes in a graph and outputs a result type of a vector of tuples pairs, where the u32 is the node, and the f64 is the calculated average path length.
 - I used rayon for this parallelization and specifically into_par_iter, which takes a range of values. We can just give it the values of the graph, because thanks to our remapping of the graph, we don't have to worry about (totally) disconnected nodes. We assign that to a closure where we map the node to the specific index that we're at. We then call closeness centrality, unwrap it, and just assign that node as the u32 node and the value as the value, then collect it. After talking with Prof Leonidas, my computer has 8 performance cores (+4 other ones that he said wouldn't help with this) so my paralleization helped speed my code to about 8x. We then sort using a sort function that I've used in the past and partial compare.
- **Closeness - writer**
 - Purpose: to take the output of parallel and write it to a file so that parallel never has to be run again.
 - In and out: takes a string path and the output of parallel in the form of a vec<(u32, f64). Returns a io result that is empty because all we want to do is write to a file.
 - Core logic and key components: we create a file path, make a buf reader, then iterate through our content, and write a newline each time, using buf reader to help the write, with a ? to handle exceptions.
 - The file that it wrote to is called closeness_centrality_storage
- **Closeness_centrality_translator - reader**

- Purpose: to read the written file of the closeness - writer. It is very similar to our last reader but differs in output and some logic, so I made it a separate function. Will be brief because it's a similar reader.
- In / out: takes file name and outputs an io result with two vectors, one closeness_centrality and the other the inverse, average path length. Both metrics are interesting and I did different things with them.
- Core logic and key components: init two vecs of u32, f64, do a similar split, just make a parse::<f64>. I then inverse the value when adding it to closeness centrality.
- **Visualizations - plotter**
 - Purpose: to plot and create a histogram of our average path length, and adjust for the small outliers, because I find them interesting and they should be on the graph as such. It creates a png histogram.
 - In / out: takes a vec of tups with node and average path length values, a minimum, a bin count, and a title for the filename. Outputs an empty result.
 - Core logic and key components: I used the scaffolding from the rust website linked above and chatgpt help to debug and understand what I should be doing. That is cited below as well. We make a drawing area, fill it with whitespace, and assign our max, mins, and bin count, we then make our bin_width equal to our max - min / bin count which is really just max / bin count because our min is so small. We make nodes and values and push onto them. Then we iterate through our values and scale our bin index, and increment our bins [bin index]. Creating a chart, and configuring that chart too.
 - build_cartesian_2d is the important part and how to get it to work.
 - Then we draw the chart, with .data and iterating over our bins and i and count
 - Then present
- **Shortest path - shortest path**
 - Purpose: find the shortest path between two nodes
 - In / out: takes a graph and a start node and an end node
 - Core logic and key components: runs BFS on the start node, then returns the dist[end] where dist is the output of BFS and end is the end node.

3. Main Workflow

- My reader module has a reader which reads the input graph file, then my closeness module takes that output, which is in the form of a graph struct adj list, and runs a BFS on each node, and then does the closeness centrality calculation, which is then given to my writer function in closeness, to write a new file with the nodes and avg path length (I will calculate the inverse which is the standard closeness centrality later). Importantly this is not currently running in my main, but above main there is a function that can be called to run it, along with a commented out piece of code that will run it, but it takes approx. 4-6 hours. This

is why I have the writer. I then made a module called `closeness centrality translator`, which reads the newly created file each time, which saves us having to run the total calculation of 4-6 hours each time, which is ~ 2 million BFS. It creates two files, the closeness centrality, and the `avg_distance` length (one is the inverse of the other). This is given to main, and then to my visualization function which creates a histogram.

- Alongside this, my `shortest_path` module just calculates the shortest path, which is a modified BFS but returns the `dist[index]` of the end value, after running BFS on the start.
- Reader -> closeness (creates `closeness centrality` storage) -> `closeness centrality translator` -> visualization
 - Plus some calculations of mean, median and running shortest path in main.

D. Tests

- **cargo test output** (paste logs or provide screenshots).

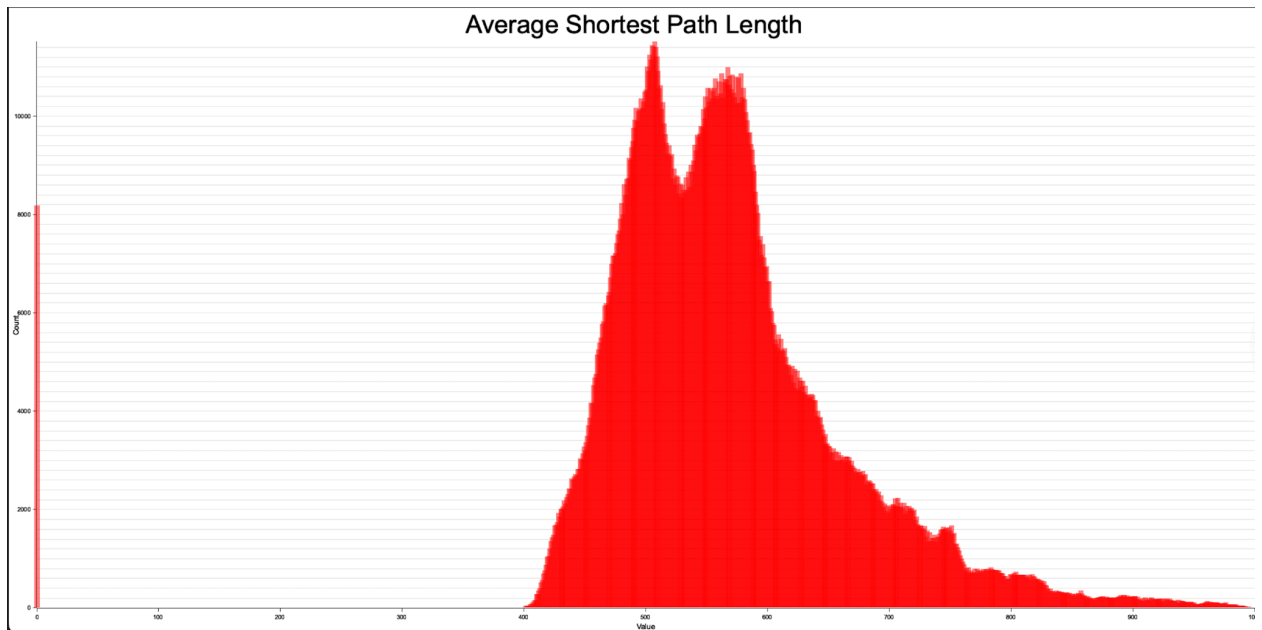
```
running 3 tests
test shortest_path::tests::tester ... ok
test closeness::tests::tester ... ok
test closeness centrality translator::tests::tester ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 2.64s
```

- Shortest_path test
 - I create a small graph by hand, calculate myself what the shortest path length between the nodes that I choose are, and then use the `assert_eq` macro to check that the output of my path length function gets the correct answer. This ensures that for a graph that I know what the answer is, it is correct.
- Closeness test
 - I make another small graph and run my parallel function on it to ensure it gives me the output that I am expecting. I used chatgpt to help me calculate the avg path length. I check that the output length is five, and check the actual output too.
- Closeness centrality translator test
 - I basically make sure that my calculations of closeness centrality are right. I inverse my average path length (apl) for each iteration and check that it is within the margin of error that I would assume is inherent in float point addition / subtraction within computers. This is significant because if this didn't work, my calculations would be totally messed up.

E. Results

- All program outputs (screenshots or pasted).



- I calculate two things, avg path length, and the inverse of that, which is the standard closeness centrality formula where $n-1 / \text{sum of distance}$ is closeness centrality.
- The most important part of this project is understanding the outliers and how I calculate my average path length. I calculate it by dividing the sum of the path lengths that are output by the BFS and dividing by the total number of nodes. While this is an interesting metric and after discussing with a professor who said it was a decent approach, I found that it let some of the data become outliers. With respect to the almost totally disconnected nodes, where perhaps the sum of the BFS vector would only be a few digit integer or less, when dividing by the length of the nodes minus 1, you get a incredibly small number, usually less than $5e-4$. When we remapped the data when processing it, it ensured that there were no gaps, but there are still pieces of data that have only one connection. This would be prevalent in closeness centrality or average path length, but I found it interesting in average path length.
- After I ran some calculations, the outliers represent about .4% of my graph. They are outliers because a lower average path length would seem to indicate a higher connectivity but as I explain earlier they do not. There are about 8000 outliers that only connected to a few, if not 1, other nodes.

F. Usage Instructions

- I have left out the actual calculations of closeness centrality from main. It took about 4-6 hours to run with the rayon parallelization (I have 8 performance cores, so 8x faster than what it would have been). I used cargo run - - release (google docs combines - - when put together so I left them one space apart). The rest of my calculations are on the file that I generate from running closeness centrality, and as such that is what I will be working with, as I only wanted to run my code once. I still found it useful to run a whole closeness centrality instead of randomly sampling a portion of my dataset to run it on,

but I decided to just write the output to a file, and only run it once. I am running my code from outside the src folder, and as such I have stored my files there. I have also duplicated my files from inside the src folder so they can be run from there, but that would require changing my path in the main, which I have kept as such.

- There is nothing to do except run `cargo run -- release`, which will output some metrics, including outliers to my calculation, and create a PNG file.
- Cargo test runs 3 tests.
- If you want to run the closeness centrality, it will take about 4-6 hours. My code with release (without closeness calculator) runs in about ten seconds.

G. AI-Assistance Disclosure and Other Citations

- Cite any **substantive** ChatGPT/GenAI you used (e.g. screenshot or description).
 - You can skip this for common knowledge/debugging use cases
- For each cited snippet, include your own explanation to show understanding.
- You can also provide links to other sources you found useful that are not "common knowledge"

With regard for AI use in this project, it was massively helpful and made the project overall much easier. The parts where I substantively used AI were one line in my reader function (although it was already in decent shape I was just struggling on syntax to get it working, because I had past reader functions that had a similar use case from previous homeworks), the closeness centrality calculations, and modifying some code that I was struggling with in respect to the histogram graph. This modification was based off the lecture notes and rust's histogram helper website as well. I struggled mostly to create the bins and understand why my code wasn't working. I would say AI was most helpful in suggesting using other things for my histogram because I was really struggling to get it working.


```

use std::fs::File;
use std::io::{self, BufRead, Write};

// type Vertex = usize;
pub fn prep(path: &str) -> Result<Vec<Vec<usize>>, io::Error> {
    let file = File::open(path)?;
    let reader = io::BufReader::new(file);
    let mut lines = reader.lines();
    let size: usize = lines.next().unwrap()?.trim().parse().unwrap();
    let mut graph: Vec<Vec<usize>> = vec![Vec::new(); size];

    for line in lines {
        let line = line?;
        let mut parts = line.split_whitespace();
        let from: usize = parts.next().unwrap().parse().unwrap();
        let to: usize = parts.next().unwrap().parse().unwrap();
        graph[from].push(to);
    }

    Ok(graph)
}

// these are the first four lines of the file that I have removed
// # Directed graph (each unordered pair of nodes is saved once):
// roadNet-CA.txt
// # California road network
// # Nodes: 1965206 Edges: 5533214
// # FromNodeId ToNodeId

// strongly connected components

I have removed the top line files

```

Thanks — super helpful context. So you've already removed the first four comment lines from the file, but you're still getting this:

```

css
called `Result::unwrap()` on an `Err` value: ParseIntError { kind: InvalidDigit }

```

That means some remaining line still has a non-numeric value, probably due to:

- a stray header ("FromNodeId ToNodeId") that wasn't removed
- blank or malformed lines
- or maybe the file wasn't fully cleaned

✓ Minimal fix to prevent panics:

You can *make your parser robust* without assuming the file is perfect. Replace your core loop with this version:

```

rust
use std::fs::File;
use std::io::{self, BufRead};

pub fn prep(path: &str) -> Result<Vec<Vec<usize>>, io::Error> {
    let file = File::open(path)?;
    let reader = io::BufReader::new(file);
    let mut lines = reader.lines();

```

```

rust                                                                    Copy Edit

use std::fs::File;
use std::io::{self, BufRead};

pub fn prep(path: &str) -> Result<Vec<Vec<usize>>, io::Error> {
    let file = File::open(path)?;
    let reader = io::BufReader::new(file);
    let mut lines = reader.lines();

    // Safer unwrap: return an error instead of panicking
    let size_line = lines.next().ok_or_else(|| io::Error::new(io::ErrorKind::InvalidData,
    let size: usize = size_line.trim().parse().map_err(|_| io::Error::new(io::ErrorKind::

    let mut graph: Vec<Vec<usize>> = vec![Vec::new(); size];

    for line in lines {
        let line = line?;
        if line.trim().is_empty() || line.starts_with('#') {
            continue;
        }

        let parts: Vec<&str> = line.split_whitespace().collect();
        if parts.len() < 2 {
            continue; // Skip malformed lines
        }

        let from = match parts[0].parse::<usize>() {
            Ok(val) => val,
            Err(_) => continue,
        };

        let to = match parts[1].parse::<usize>() {
            Ok(val) => val,
            Err(_) => continue,
        };

        if from < size && to < size {
            graph[from].push(to);
        }
    }

    Ok(graph)
}

```

The strong majority of the reader function that I use for both readers that I implement comes from HMWRK7 reader function, which basically does the exact same thing as this one. I was struggling to get my reader to work and took some small suggestions from chatgpt. I found it easier to remove the # marks in front of each file though. A walkthrough of the functions that I use are above, but I would say the most important thing that I added is the parse part, where I actually assign what the types are going to be, along with a check to make sure that the size of the collected graph is only 2. That was very helpful. The other part that was helpful was the

parse::usize which I modified to be parse::u32 which indicates to rust that whatever you are parsing is a u32.

<https://www.geeksforgeeks.org/shortest-path-unweighted-graph/>

I used this link to understand how to do shortest path with BFS, and although it didn't have rust, the implementation was very similar so when it said that you basically just use dist. It was helpful in understand that all we really need to do it use the dist file and just return the end target so it was just dist[end] after we were finished running BFS on the start node.

<https://github.com/plotters-rs/plotters/blob/master/plotters/examples/histogram.rs>

I used the rust docs code for a histogram as the base for my understanding. I felt it was best to start off it that and the lecture as an understanding as the base for my code. I then tried to implement what I could but I had to ask chatgpt for some help in the preprocessing of my data within the plotters function.

```
pub fn many(graph: Vec<Vec<u32>>) -> Result<Vec<(u32, f64)>, Box<dyn
std::error::Error>> {

    let mut storage: Vec<(u32, f64)> = Vec::new();

    let length = graph.len();

    println!("Length of graph is {}", length);

    for iter in 0..length {

        if iter % 100 == 0 {

            println!("Working on node {}", iter);

        }

        let current_closeness = closeness(&graph, iter);

        storage.push((iter as u32, current_closeness.unwrap()));

    }

}
```

```
storage

.sort_by(|x,y| x.1.partial_cmp(&y.1).unwrap());

Ok(storage)
}
```

Wrote this closeness centrality function but this wasn't parallelized. I struggled to get it working to parallelize it, so I asked for chatgpt with some help with this current understanding. I knew that I wanted to parallelize it but I couldn't figure out how to implement it.

```
pub fn many(graph: Vec<Vec<u32>>) -> Result<Vec<u32, f64>,>
Box<dyn std::error::Error>> {
    let mut storage: Vec<Vec<u32, f64>> = Vec::new();
    let length = graph.len();
    println!("Length of graph is {}", length);
    for iter in 0..length {
        if iter % 100 == 0 {
            println!("Working on node {}", iter);
        }
        let current_closeness = closeness(&graph, iter);
        storage.push((iter as u32, current_closeness.unwrap()));
    }

    storage
        .sort_by(|[x, y], [x2, y2]| x.1.partial_cmp(&y.1).unwrap());

    Ok(storage)
}
```

I am really struggling to implement rayon to parallelize this graph, could you help me do that? I know that I want to use rayon to call my bfs and run it simultaneously?

Absolutely — you're very close, and your instinct is right: you want to use `rayon` to run `closeness(&graph, i)` for each node `i` in parallel, just like your current `many` function but faster. Here's exactly how to do that with minimal changes to your code.

✓ Parallel Version Using Rayon

Make sure this is at the top of your file:

```
rust
use rayon::prelude::*;
```

Then modify your `many` function like this:

```
rust
pub fn many(graph: Vec<Vec<u32>>) -> Result<Vec<u32, f64>,> Box<dyn std::error::Error>> {
    println!("Length of graph is {}", graph.len());

    // Process each node index in parallel
    let mut storage: Vec<Vec<u32, f64>> = (0..graph.len()).into_par_iter()
        .map(|i| {
            if i % 100 == 0 {
                println!("Working on node {}", i);
            }

            // SAFETY: graph is shared immutably and closeness doesn't mutate it
            let closeness_val = closeness(&graph, i).unwrap_or(0.0);
            (i as u32, closeness_val)
        })
        .collect();

    // Sort the result after collection
    storage.sort_by(|[x, y], [x2, y2]| x.1.partial_cmp(&y.1).unwrap());
    Ok(storage)
}
```

✓ Notes

- Rayon's `into_par_iter()` handles multithreading under the hood.
- Your `graph` is passed by value, but we use it immutably (via `&graph` inside the closure), so Rayon can safely clone the reference internally for parallel reads.
- The sort happens afterward, on the full collected vector.

📦 Cargo.toml (in case you haven't added it yet)

```
toml
[dependencies]
rayon = "1.7"
```

Let me know if you also want to parallelize BFS internally for each node — but for now, this setup will give you parallel node processing using your existing BFS. Do you want to measure the speedup or print a runtime comparison too?

👍 🗨️ 🔄 📄 🏠

Do you like this personality? 👍 🗨️ ✕



Ask anything

+ 🔍 Search 🔍 Deep research 🖼️ Create image ⋮



ChatGPT can make mistakes. Check important info.

This is the help that chatgpt gave me, as I explain earlier to, what we do is create a range that we iterate through using the `into_par_iter`, and then map associates that node with the current

index in range, then we call closeness that I had already written, and assign the values. The sorting functionality is still based off my earlier code, which was based off earlier completed homeworks.

Tips:

- Don't write filler content—just hit each bullet.
- Write in concisely and to the point.
- Treat this as “documentation,” not an essay/paper.
- Include headings to label important sections.