Technical Report: Final Project
EECE 2560: Fundamentals of Engineering Algorithms

Tyler Myers & Alex Hunt
Department of Electrical and Computer Engineering
Northeastern University
myers.ty@northeastern.edu & hunt.al@northeastern.edu

December 5, 2024

Contents

1 Project Scope

The purpose of this project is to develop a program to dispatch emergency services to various locations on Northeastern University's campus based on the urgency of the situation, route, terrain, and distance. The project's main objectives are:

- To find optimal paths for emergency services based on their location on campus and the incident location.
- To respond to incidents as quickly as possible by dispatching the nearest available emergency services based on the incident type.
- To dynamically deploy emergency services to locations on campus that are incident hot spots.

The deliverables for this project were a terminal-based program, documentation for the program, and a project report.

2 Project Plan

2.1 Timeline

The overall timeline for the project is divided into phases:

- **Week 1 (October 7 - October 13)**: Define project scope, establish roles, determine necessary tools and skills, conduct research, and set up project repository.
- **Week 2 (October 14 - October 20)**: Begin project development and start coding basic functionalities.
- **Week 3 (October 21 - October 27)**: Continue programming, ensure basic functionalities are compatible, begin writing documentation.
- **Week 4 (October 28 - November 3)**: Further work on the program. Set up the slideshow. Further work on the documentation.
- **Week 5 (November 4 - November 10)**: Finalize the programming. Conduct testing. Wrap up the documentation.
- **Week 6 (November 11 - November 17)**: Polish and finalize the technical report and slideshow.
- **Week 7 (November 18 - November 28)**: Presentation, report submission, and project end.

2.2 Milestones

Key milestones include:

- Project scope, plan, GitHub repository setup. (October 7).
- Begin project development. (October 9).
- Finalize programming (November 4).
- Final program testing (November 10).
- Report ready for submission and slideshow ready for presenting (November 17).
- Final presentation and report submission (November 28).

2.3 Team Roles

- **Tyler Myers:** Route Traversal, Divide & Conquer
- **Alex Hunt:** Greedy Algorithm, Dynamic Programming

3 Methodology

3.1 Pseudocode and Complexity Analysis

The first algorithm is allocating officers across campus so they are already in place beforehand to respond to emergencies. This approach is suitable for the project requirements because it ensures over the long term every emergency will always have equal access to officers based on their historical emergency presence.

Algorithm 1: DynamicOfficerAllocation

Input: numOfficers
Output: None
numOfficers ← numOfficers - 4
while numOfficers > 0 **then**
    i ← 1
    worstZone ← 0
    while pastEmergencies[worstZone] == 0 **then**
        worstZone++
    end
    while i < 4 **then**
        if PastEmergencies[i] != 0 **then**
            if OfficersAllocated[i]/PastEmergencies[i] < OfficersAllocated[worstZone]/PastEmergencies[worstZone] **then**
                worstZone ← i
            end
        end
        i++
    end
    OfficersAllocated[worstZone]++
    numOfficers–
end
End Algorithm: DynamicOfficerAllocation

This algorithm takes an input of the number of officers currently on staff and allocates them among four zones on campus. The algorithm starts by allocating one officer to each zone which is safe because Northeastern is a very large university and will always have several dozen officers available. It is also best to ensure every zone has access to at least 1 officer. The first while loop iterates every time until no officers are unallocated. Start by assuming the worst ratio of officers to past emergencies is in zone 0 then compare it to every zone with the inner loop. The officer is allocated to the zone with the worst ratio.

Frequency Count:
(n = numOfficers)
Decrementing numOfficers: 1 operation
while loop: n operations
i initialization n operations
worstZone initialization n operations
while loop 4n operations, worst case it iterates through all 4 zones because 3 had no past emergencies
worstZone increment 4n operations
while loop 3n operations, starts at index 1 and runs until index 3
if 3n operations
worstZone assignment 3n operations
i++ 3n operations
OfficersAllocated[worstZone]++ n operations

numOfficers- n operations
Total: 25n + 1
Time Complexity: O(n)

Algorithm 2: solveKnapsack

Input: *items* (list of Equipment): Each item has *name*, *weight*, *importance*, and *quantity*. *maxWeight*, the maximum weight of the knapsack.
Output: A list of strings representing the names of the chosen items.
Begin Algorithm: solveKnapsack
$n \leftarrow$ Size of *items*
$dp \leftarrow$ 2D array of size $(n+1) \times (maxWeight + 1)$ initialized to 0
for $i \leftarrow 1$ ***to*** n **do**
  for $w \leftarrow 1$ ***to*** $maxWeight$ **do**
    if *items[i-1].weight* $\leq w$ ***and*** *items[i-1].quantity* > 0 **then**
      $dp[i][w] \leftarrow \max(dp[i-1][w], dp[i-1][w - \text{items}[i-1].\text{weight}] + \text{items}[i-1].\text{importance})$
    end
    else
      $dp[i][w] \leftarrow dp[i-1][w]$
    end
  end
end
chosenItems $\leftarrow$ empty list
$w \leftarrow maxWeight$
for $i \leftarrow n$ ***down to*** 1 ***and*** $w > 0$ **do**
  if $dp[i][w] \neq dp[i-1][w]$ **then**
    Add items$[i-1]$.name to *chosenItems*
    $w \leftarrow w - \text{items}[i-1].\text{weight}$
  end
end
Return *chosenItems*
End Algorithm: solveKnapsack

The solveKnapsack algorithm is designed to solve a 0-1 Knapsack Problem using a dynamic programming approach. The algorithm takes as its input a list of item objects, each having a weight and importance attribute, and the maximum weight of the knapsack. It starts by initializing a 2D array, dp, to store the maximum possible importance for every weight and item combination. The first two nested loops fill out the table: the outer loop iterates over every item and the inner loop iterates over all the possible weights. For every item, the algorithm checks whether if it is possible to fit the item in the current knapsack capacity given that the item is available. If it can, the algorithm decides to include the item based on whatever choice has the higher importance. Once the table is filled, the algorithm backtracks to find the items included in the best solution.

Time Complexity Analysis

The algorithm solveKnapsack has a time complexity of O(n$\times$w) where n is the number of items and w is the knapsack's weight capacity. This is because of the two nested loops: one that iterates over the items and one iterating over the possible weights. For every subsequent item-weight combination, the algorithm performs a constant amount of work and the number of operations grows with each factor. Therefore, the time complexity is proportional to the number of pairs in the dp table, that is, n $\times$ w.

Algorithm 3: DeployOfficerToIncident

Input: *officerZones* (list of zones, each containing officers).
emergencyLocation (string): The location of the emergency.
zoneNames (list of strings): The names of the zones.
Output: A pair determining deploying the nearest available officer to the emergency location (1), or indicate failure (0) as the first value; and the officer's location as the second.

```
Begin Algorithm: DeployOfficerToIncident
shortestPath ← ∞
selectedZone ← −1
selectedIndex ← −1
deploymentZones ← empty list
for i ← 0 to |zoneNames| − 1 do
    distance ← getPolyLineDistance(zoneNames[i], emergencyLocation)
    Add (i, zoneNames[i], distance) to deploymentZones
end
Sort deploymentZones in ascending order of distances.
for (zoneIndex, zoneName, distance) ∈ deploymentZones do
    for officerIndex ← 0 to |officerZones[zoneIndex]| − 1 do
        officer ← officerZones[zoneIndex][officerIndex]
        if officer.isAvailable then
            selectedZone ← zoneIndex
            selectedIndex ← officerIndex
            shortestPath ← distance
            Break inner loop
        end
    end
    if selectedZone ≠ −1 then
        Break outer loop
    end
end
if selectedZone ≠ −1 and selectedIndex ≠ −1 then
    selectedOfficer ← officerZones[selectedZone][selectedIndex]
    selectedOfficer.isAvailable ← false
    Print "Deployed officer with badge ID selectedOfficer.ID from
      zoneNames[selectedZone] to emergencyLocation with distance
      shortestPath meters."
    Return 1
end
else
    Print "Critical campus-wide emergency present! No available
      officers to deploy!"
    Return 0
end
End Algorithm: DeployOfficerToIncident
```

The DeployOfficerToIncident algorithm picks the nearest available officer to respond to an emergency based on the proximity of the location that they are stationed at to the emergency location. The algorithm begins by computing the distance between the emergency location given by the user and the predefined deployment zones. Then, the algorithm stores the distance for each zone and sorts the zones by distance in ascending order using the introsort algorithm. The first available officer is selected and their availability attribute is changed to false. If the algorithm is unable to find any available officers, the algorithm returns a failure message. If an available officer is found, the algorithm prints a message indicating their deployment. Finally, it returns a pair where the first value being 1 indicates success, and 0 indicating failure; and the second value is a string of the location further use in the program.

Time Complexity Analysis
The time complexity of the algorithm can be broken down into two parts.
The first part, the distance calculation, the algorithm computes the distance to the emergency location using the getPolyLineDistance method. This is performed for each of the four zones, so the time complexity of this section is O(4). The second part where the algorithm sorts the list of zones by their distance has a time complexity of O(nlogn) where n is the number of zones. Since there are only four zones, the time complexity can be simplified to O(4log4) = O(1). Once the algorithm has sorted the zones by distance, it iterates through each zone to find an available officer. For each zone, every officer is checked until an officer is found. This section has a time complexity of O(n), where n is the number of all officers.
Therefore, the time complexity of this section is O(n+4log4+4) = O(n) time complexity.

3.2 Data Collection and Preprocessing

All data collected and used throughout this project had to be created by our team as it is sensitive information we would not be permitted to access from NUPD's records. With our data it is stored and parsed past from two CSV files. The first file contains two main forms of data regarding past emergencies including what zone they occurred in and what the emergency was. The emergency type currently is unused by our program, yet is included to help setup for future expansion on the project allowing distribution of officer personnel based upon their skillset. Yet, the used data is the zones of the past emergencies this data is parsed into an array of size four with each index representing the number of emergencies that have occurred in that zone.
The second file contains all officers currently on duty which are referenced by their id, along with what zone they are currently allocated to with a boolean dictating if they are currently available to be dispatched.

The next aspect of data preprocessing occurs whenever personnel input an emergency into our project. We pull routing information from Google API, and the information we acquire from them cannot simply be parsed and understood by

our code. To extract what we need from it we utilize cURL to reformat the information into a parsable JSON format.
In this step we also utilize data hardcoded into our project regarding resources available. This data is processed through a 0-1 Knapsack algorithm to tell officers what equipment suits their emergency.

4 Results

The result of this program is a command line interface allowing easy intractability for any officer personnel. When first loading the program personnel will see the following:

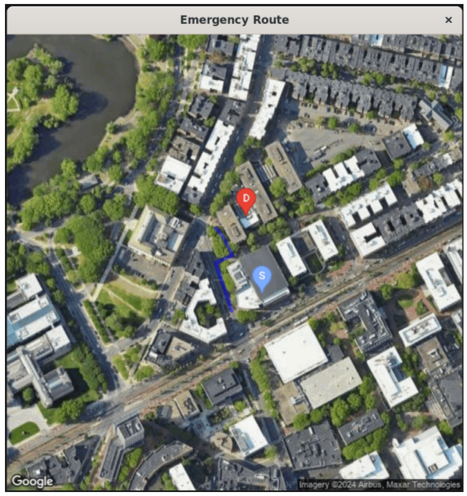
```
Enter the destination of the emergency:
```

This prompts the user to enter the name of any building regarding Northeastern University's Boston Campus. Due to utilizing tools including Google API autocomplete and searching within a 3km area of NU's Boston campus the code ensures we input the best guess for what building the emergency occured at. Additionally due to the autocomplete feature, if the building is listed on Google Maps then it can be found allowing coverage of edge cases especially because Google Maps will update as buildings change names, are destroyed, and are built.
Once the user enters a location they will be greeted with the following screen:

```
Enter the type of emergency:
1. fire alarm
2. fighting
3. theft
4. alcohol overdose
5. drug overdose
6. acute non lethal injury
7. potentially lethal injury
8. exit
```

Following this, the user can input any of the following options and if an incorrect input is entered then the same prompt will print followed by an error saying to enter a correct input. Upon sufficent entry of both statements the following will be displayed to the user:

This figure will be shown to the user. The S marker represents the officer's current location and starting point. The D represents the location of the emergency and the officer's destination. The blue line represents the path they should take.



Alongside the map the terminal will output the following message which provides more details to the officer.

```
Emergency at: Stetson West
Deployed Officer ID: 3 from Marino to Stetson West with distance 122 meters
Optimal equipment for fire alarm
fire axe
fire extinguisher
```

It starts by outputting the destination of the emergency as inputted by the personnel. It outputs this style and not the address from Google API because if the API routes to the wrong location the officer can view and confirm the correct location. Additionally, this output then says exactly which officer will be deployed to the emergency to help prevent confusion based on who is deployed. Lastly, the equipment they need to bring is already displayed to them along with what the emergency is encase they need to make specific alterations regarding the situation.

5 Discussion

The results of this project is a really good proof of concept. It sets up the foundation of what this type of application can do along with features kept in mind for easy expansion. Comparing the results of the project to the initial objectives everything was met to some degree, while they could have been flushed out more. Starting off with routing of the objective of finding optimal paths I feel that utilizing the Google Maps API is the perfect way to do this as it is very realistic. While it could be replaced by Dijkstra's algorithm that would require creating a custom map of all the buildings on campus that would have left much less time for the rest of the algorithm. Along with the custom map

being less efficient and relaying much less data to the user.
The next objective was routing the closest personnel to the emergency to prioritize the response time of the officers. This set up utilized pulling officers from four locations on campus where they were stationed. This objective was perfectly met as it was a simple application of a greedy algorithmic approach.
The last objective was dynamically deploying emergency services across campus. Only half of this objective was completed through officer allocation to four zones. The methodology of how this was completed by utilizing the pre-existing 4 AED zones on campus as a basis for the zones allows potential integration of our system to be in use as the personnel would already be familiar with the zones. The part that was incomplete is allocating the equipment to each zone based upon past emergencies. We ran out of time right beforehand leading this to be dropped from the project.

6 Conclusion

The key findings of this project was that our system is a very good minimum viable product that sets the foundation for a system that if tuned would be great at a large scale. The algorithmic choice for each stage allowed the project to be deployable and impactful in the real world. Starting with choosing to use Google Maps API for routing it would allow personnel to receive routing in real time which adapts based upon traffic and events happening. Additionally if the project was to expand to other Northeastern Campuses it would be very easy to alter the search area from Boston campus to the new location. Which is much easier than having to create an entirely new map with edge weights for every building. Additionally, regarding the Boston campus, the use of Google Maps API also handles emergencies which might be slightly off campus.
For developing the project there is so many features we could have added to the project that would allow it to be more viable yet our biggest limitation was the amount of time we had in one semester to devote to the project. With the vast amount of scenarios that NUPD has to respond to there was no chance of being able to cover them all.
Based upon what we were unable to cover with this project there are a few categories that stick out as key locations for improvement. First off, there is the GUI not being a command terminal. This would allow it to be more interactive and easier to use. Another direction to go would allow the user more control of what types of emergencies and equipment they have instead of everything being hardcoded to allow for a more flushed out application. This goes hand-in-hand with the next place for improvement of adding more emergencies and equipment available. We decided to keep it rather limited as it went against the goal of project by doing what is essentially busy work. We would have spent hours working out all the equipment, their weights, their importance for each emergency, plus types of emergencies. This addition would heavily improve our final result yet it takes time away from the algorithmic design for the project.

7 References

"Google API" Console.cloud.google.com, console.cloud.google.com/apis/library.
"Google Maps Platform Documentation." Google for Developers, 2024, developers.google.com/maps/documentation?utm_experiment=39300572.

A Appendix A: Code

1. DynamicOfficerAllocation

```
/**
 * @brief allocates officers to locations across campus based upon past emergency data
 * @param numOfficers: how many officers are currently on duty
 * @retval None
 */
void DynamicOfficerAllocation(int numOfficers)
{
    /* The first location is where the officers are stationed
       The second location is the AED zone the station point is based upon
       Zone 1 ~ Columbus Place (Columbus South Sector)
       Zone 2 ~ Behrakis (West Campus Sector)
       Zone 3 ~ Curry (Academics Sector)
       Zone 4 ~ Marino (East Fenway Sector)
       Zones are stored in the index 1 less than their number
    */
    numOfficers = numOfficers - 4; // Fair to assume at least 4 officers are always on duty due to size of our university so one officer is assigned per zone
    while (numOfficers > 0)
    {
        int i = 1;
        int worstZone = 0;
        while (PastEmergencies[worstZone] == 0) // prevent divide by 0
        {
            worstZone++;
        }
        while (i < 4) // loop 4 times once for each zone
        {
```

```
            if (PastEmergencies[i] != 0) // prevent divide by 0
            {
                if (OfficersAllocated[i]/PastEmergencies[i] < OfficersAllocated[worstZone]/PastEmergencies[worstZone]) // if ratio worse then allocate officer
                {
                    worstZone = i;
                }
            }
            i++;
        }
        // allocate officer to zone and remove 1 from officers left to allocate
        OfficersAllocated[worstZone]++;
        numOfficers--;
    }
}
```

2. solveKnapsack

```
/**
 * @brief 0-1 Knapsack problem solver
 * @param items: vector of equipment importances and inventory relevant to emergency
 * @param maxWeight: how much officer can carry based on their strength
 * @retval vector<string>
 */
std::vector<std::string> solveKnapsack(const std::vector<Equipment> &items, int maxWeight)
{
    int n = items.size();
    // 2D vector representing maximum importance possible with the first "i" items considering a weight limit of "w"
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(maxWeight + 1, 0));

    // Fill the DP table
    for (int i = 1; i <= n; ++i) // Iterate over each item
    {
        for (int w = 1; w <= maxWeight; ++w) //
```

```cpp
 Iterate over each weight limit
        {
            if (items[i - 1].weight <= w && items[i - 1].quantity != 0) // If item can fit in the knapsack with the current weight limit and is available
            {
                dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - items[i - 1].weight] + items[i - 1].importance); // Choose the maximum
            }
            else // If not, do not include
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    // Go back to find chosen items
    std::vector<std::string> chosenItems;
    int w = maxWeight;

    // Traverse the dp table to reconstruct the solution
    for (int i = n; i > 0 && w > 0; --i)
    {
        if (dp[i][w] != dp[i - 1][w]) // If current != previous, the item was included
        {
            chosenItems.push_back(items[i - 1].name); // Add item to the list
            w -= items[i - 1].weight; // Reduce the remaining weight
        }
    }

    return chosenItems;
}
```

3. DeployOfficerToIncident

```cpp
/**
 *  @brief Used to select the neareast available officer closest to a emergency based on zone
 *  @param emergencyLocation: where the emergency
```

```
    occurs
 *  @retval pair<int, std::string>
 */
std::pair<int, std::string> DeployOfficerToIncident(
   std::string emergencyLocation)
{
    // Initialize variables to track relevant info
    double shortestPath = std::numeric_limits<double
       >::max();
    int selectedZone = -1;
    size_t selectedIndex = -1;

    // Vector to store deployment zone data
    std::vector<std::tuple<int, std::string, double>>
       deploymentZones;
    std::vector<std::string> zoneNames = {"Columbus␣
       Place␣and␣Alumni␣Center",
                                        "Behrakis␣
                                           Health␣
                                           Sciences␣
                                           Center",
                                        "Curry␣Student
                                           ␣Center",
                                        "Marino␣
                                           Recreation␣
                                           Center"};

    // Calculate distance of each zone to the
       emergency and store it
    for (size_t i = 0; i < zoneNames.size(); ++i)
    {
        double distance = getPolyLineDistance(
           zoneNames[i], emergencyLocation);
        deploymentZones.push_back(std::make_tuple(i,
           zoneNames[i], distance));
    }

    // Sort deployment zones by distances in ascending
        order with the help of a lambda function
    std::sort(deploymentZones.begin(), deploymentZones
       .end(),
              [](const std::tuple<int, std::string,
                 double> &a, const std::tuple<int,
                 std::string, double> &b){
              return std::get<2>(a) < std::get<2>(b)
                 ;
```

```
            });

    // Iterate through sorted deployment zones and find the first available officer
    for (const auto &[zoneIndex, zoneName, distance] : deploymentZones)
    {
        // Check all officers in the current zone
        for (size_t officerIndex = 0; officerIndex < officerZones[zoneIndex].size(); ++officerIndex)
        {
            Officer &officer = officerZones[zoneIndex][officerIndex];

            // If an officer is found update the selected zone and officer info
            if (officer.isAvailable)
            {
                selectedZone = zoneIndex; // Record zone index
                selectedIndex = officerIndex; // Record officer index
                shortestPath = distance; // Update the shortest distance
                break;
            }
        }

        // If a selected officer is found, exit the loop.
        if (selectedZone != -1) break;
    }

    // If some officer is successfully chosen
    if (selectedZone != -1 && selectedIndex != -1)
    {
        Officer &selectedOfficer = officerZones[selectedZone][selectedIndex]; // Get selected officer
        selectedOfficer.isAvailable = false; // Change availability attribute

        // Print deployment details
        std::cout << "Deployed officer with badge ID "
            << selectedOfficer.ID
```

```
                  << " from " << zoneNames[selectedZone]
                      << " to " << emergencyLocation
                  << " with distance " << shortestPath
                     << " meters." << std::endl;
        return {1, zoneNames[selectedZone]}; // Returning 1 indicates success
    }
    else // No available officers found
    {
        std::cout << "Critical campus-wide emergency present! No available officers to deploy!" << std::endl;
        return {0, ""}; // Returning 0 indicates failure.
    }
}
```

4. getPolylineDistance

```
std::string getPolyLine(std::string origin, std::string destination)
{
    // Construct the URL for the Directions API
    std::string polyUrl = "https://maps.googleapis.com/maps/api/directions/json?"
                    "origin=" + origin +
                    "&destination=" + destination +
                    "&mode=walking"
                    "&location=Northeastern+University" // limit search results to northeastern area
                    "&radius=3000" // Limit search radius to within 3km of Northeastern
                    "&key=" + apiKey;

    std::string polyline;

    std::string readBufferPoly;
    CURL* curlPoly;
    CURLcode resPoly;
    curlPoly = curl_easy_init();

    if (curlPoly) {
        // curl code is required and formats the data recieved from Google API into a parsable
```

```
            json
        curl_easy_setopt(curlPoly, CURLOPT_URL, polyUrl.c_str());
        curl_easy_setopt(curlPoly, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curlPoly, CURLOPT_WRITEDATA, &readBufferPoly);
        curl_easy_setopt(curlPoly, CURLOPT_FAILONERROR, 1L);
        resPoly = curl_easy_perform(curlPoly);
        curl_easy_cleanup(curlPoly);
        // Parse the JSON response to extract the polyline
        nlohmann::json jsonResponse = nlohmann::json::parse(readBufferPoly);
        if (!jsonResponse["routes"].empty()) {
            polyline = jsonResponse["routes"][0]["overview_polyline"]["points"].get<std::string>();
        } else {
            std::cerr << "No␣routes␣found." << std::endl;
        }
    }
    return urlEncode(polyline);
}

/**
 *  @brief Used to get the length of polyline
 *  @param origin: where the polyline will start
 *  @param destination: where the polyline will route to
 *  @retval double
 */
double getPolyLineDistance(std::string origin, std::string destination)
{
    std::replace(origin.begin(), origin.end(), '␣', '+'); // replace all ' ' to '+'
    std::replace(destination.begin(), destination.end(), '␣', '+'); // replace all ' ' to '+'

    origin = autocompleteAddress(origin);
    destination = autocompleteAddress(destination);
    // Construct the URL for the Directions API
    std::string polyUrl = "https://maps.googleapis.com
```

```
/maps/api/directions/json?"
                    "origin=" + origin + // Add NEU to address for accuracy
                    "&destination=" + destination + // Add NEU to address for accuracy
                    "&mode=walking"
                    "&location=Northeastern+University" // limit search results to northeastern area
                    "&radius=3000" // Limit search radius to within 3km of Northeastern
                    "&key=" + apiKey;

    std::string polyline;

    std::string readBufferPoly;
    CURL* curlPoly;
    CURLcode resPoly;
    curlPoly = curl_easy_init();

    double distance = 0;

    if (curlPoly) {
        // curl code is required and formats the data recieved from Google API into a parsable json
        curl_easy_setopt(curlPoly, CURLOPT_URL, polyUrl.c_str());
        curl_easy_setopt(curlPoly, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curlPoly, CURLOPT_WRITEDATA, &readBufferPoly);
        curl_easy_setopt(curlPoly, CURLOPT_FAILONERROR, 1L);
        resPoly = curl_easy_perform(curlPoly);
        curl_easy_cleanup(curlPoly);

        // Parse the JSON response to extract the polyline
        nlohmann::json jsonResponse = nlohmann::json::parse(readBufferPoly);
        if (!jsonResponse["routes"].empty()) {
            polyline = jsonResponse["routes"][0]["overview_polyline"]["points"].get<std::
```

```
                string>();
            distance = jsonResponse["routes"][0]["legs
                "][0]["distance"]["value"].get<double
                >();
        } else {
            std::cerr << "No routes found." << std::
                endl;
        }
    }
    return distance;
}
```

5. WriteCallback

```
/**
 * @brief Used by curl to help format Google API query
 * @retval size_t
 */
static size_t WriteCallback(void* contents, size_t
    size, size_t nmemb, void* userp) {
    ((std::string*)userp)->append((char*)contents,
        size * nmemb);
    return size * nmemb;
}
```

6. Equipment Struct

```
/**
* @brief Equipment structure that makes up every
    equipment available on campus
* @param name: equipments' offical reference name
* @param weight: how much it weighs limits the 0-1
    Knapsack
* @param importance: determines what choice to make
    for the 0-1 Knapsack
* @param quantity: how much of the equipment there is
    at a zone.
*/
struct Equipment
{
    std::string name;
    int weight;
    int importance;
    int quantity;
};
```

7. Equipment Struct

```cpp
/**
 * @brief Officer structure that makes up every
     officer on campus
 * @param ID: officer's badge #
 * @param isAvailable: true = currently at location &&
      false = currently on response to emergency
 */
struct Officer
{
    std::string ID;
    bool isAvailable;
};
```