Planungstool für Katastrophenmanagement

Stephane Sandevski, 22 59 802

Marius Ureche, 22 60 716

1. Einleitung

Dieses Programm dient als Planungstool für Katastrophen, welches im Notfall von den Einsatzkräften genutzt werden, um eine schnelle und effiziente Reaktion zu ermöglichen.

Das Programm liest die Stadtkarte mit Hilfe verschiedener Adjazenzmatrizen ein, anhand derer dann die einzelnen Anforderungen ausgeführt werden, und in Krisensituationen eine schnelle und effektive Hilfeleistung sicherzustellen.

2. Struktur des Tools

Das Tool ist modular aufgebaut, wobei jedes Modul für eine spezifische Funktionalität zuständig ist.

Das Hauptmodul „menu.py“ beinhaltet unser Menü, welches über dem Terminal die einzelnen Funktionalitäten aufrufbar macht. Die Module „b1.py“ und „b2.py“ sind hierbei für das Einlesen, Ausgeben und Modifizieren des jeweiligen Stadtplanes verantwortlich. Die restlichen Dateien beinhalten die wesentlichen Algorithmen und Abläufe des Programms, welche im nächsten Teil detaillierter beschrieben werden.

3. Beschreibung der Dateien

Menu: (menu.py)

Der Code bietet ein interaktives Planungstool für das Katastrophenmanagement, das verschiedene Funktionen zur Verwaltung und Modifikation eines Stadtplans sowie zur Planung von Evakuierungsrouten und Versorgung von Einsatzkräften bereitstellt.

Funktionen und ihre Aufgaben

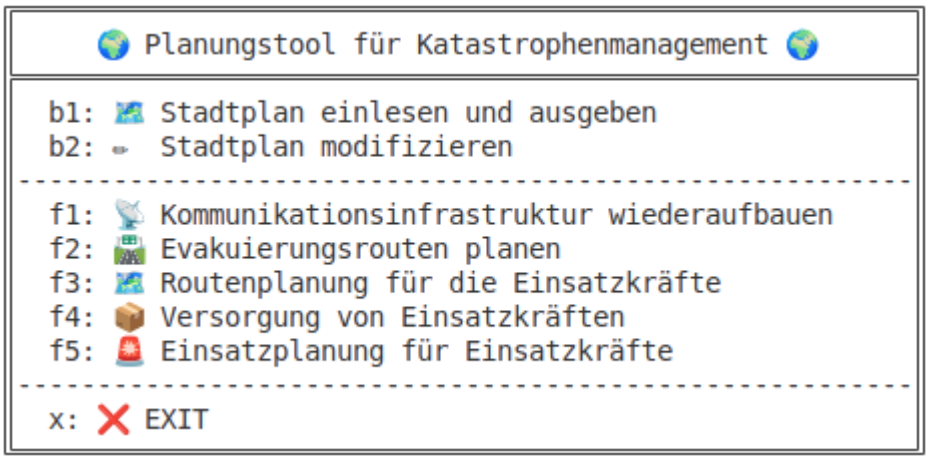
wait_for_return_to_menu():

Aufgabe: Wartet auf eine Benutzereingabe (Enter-Taste), um zum Hauptmenü zurückzukehren.

show_main_menu():

Aufgabe: Zeigt das Hauptmenü des Planungstools an, wobei verschiedene Optionen zur Auswahl angeboten werden.

main():
Aufgabe: Die Hauptfunktion des Programms, die eine Endlosschleife enthält, um das Hauptmenü anzuzeigen und Benutzereingaben zu verarbeiten. Abhängig von der Benutzerauswahl werden verschiedene Module und Funktionen des Planungstools ausgeführt:



Jede dieser Optionen ruft spezifische Funktionen oder Module auf, um die entsprechenden Aufgaben zu erledigen. Das Programm ermöglicht eine benutzerfreundliche Interaktion zur Verwaltung und Optimierung von Katastrophenszenarien.

B1: Stadtplan einlesen und ausgeben (b1.py):

Das Ziel des Codes ist es, verschiedene Adjanzmatrizen aus Textdateien zu lesen und diese darzustellen. Die Adjazenzmatrizen repräsentieren unterschiedliche Typen von Graphen (ungerichtet/gerichtet, gewichtet/ungewichtet). Aus der Text dateien mit den Matrizen werden dann 2D-Array (list of lists in python) umgewandelt für die weitere nutzung in andren Aufgaben.

Funktionen und ihre Aufgaben

read_adjacency_matrix(file_path):

Diese Funktion liest eine Adjazenzmatrix aus einer Datei ein. Sie überspringt die erste Zeile (die Labels) und die erste Spalte jeder Zeile (ebenfalls Labels), um nur die numerischen Werte der Matrix zu extrahieren.

print_matrix(matrix, nodes):

Diese Funktion gibt eine Adjanzmatrix auf der Konsole aus. Sie druckt zunächst die Knoten Labels und anschließend jede Zeile der Matrix, wobei die Knotenlabels den Zeilen vorangestellt werden.

Die Adjazenzmatrizen werden aus vier verschiedenen Dateien geladen und anschließend können sie mit der print_matrix Funktion angezeigt werden.

B2: Stadtplan modifizieren (b2.py):

Der Code ermöglicht es, einen Stadtplan, dargestellt als ungerichteter, gewichteter Graph, zu modifizieren. Dies umfasst das Modifizieren, Hinzufügen und Entfernen von Kreuzungspunkten (Knoten) und Straßen (Kanten).

Funktionen und ihre Aufgaben

add_node(node, node_name):

Aufgabe: Fügt einen neuen Knoten (Kreuzungspunkt) zum Graphen hinzu, erweitert die Liste der Knoten und die Adjazenzmatrix.

add_edge(node1, node2, weight):

Aufgabe: Fügt eine neue Kante (Straße) zwischen zwei Knoten mit einem bestimmten Gewicht (Länge der Straße) hinzu, indem die entsprechenden Einträge in der Adjazenzmatrix gesetzt werden.

remove_node(node):

Aufgabe: Entfernt einen Knoten und alle zugehörigen Kanten aus dem Graphen, indem der Knoten aus der Liste entfernt und die entsprechende Zeile und Spalte aus der Adjazenzmatrix gelöscht werden.

test():

Aufgabe: Bietet ein interaktives Menü, um den Benutzer durch die verschiedenen Modifikationen des Stadtplans zu führen, einschließlich des Druckens, Hinzufügens und Entfernens von Knoten und Kanten.

F1: Kommunikationsinfrastruktur wiederaufbauen (f1.py):

Das Ziel ist es, dass beim Ausfall der Kommunikationsstruktur die wichtigsten Punkte wie Rettungsstationen, Krankenhäusern und das Rathaus miteinander verbunden werden sollen. Dafür verwenden wir den Prim Algorithmus, dessen Prinzip ist es, einen minimalen Spannbaum eines ungerichteten, gewichteten Graphen zu bestimmen. Dieser minimale Spannbaum stellt unsere Kommunikationsinfrastruktur dar.

Funktionen und ihre Aufgaben:

prim(graph, important_nodes):

Zu Beginn wird die Länge des Graphen als n definiert. Es werden mehrere Variablen initialisiert: mst als leere Liste für die Kanten des MST, visited für besuchte Knoten, min_heap als Prioritätswarteschlange, die mit dem ersten Knoten aus der Liste important_nodes initialisiert wird, sowie total_weight für die Gesamtkosten, anfangs auf 0 gesetzt. Die Variable prev_node speichert die vorherige Kante, um die Kanten des MST korrekt hinzuzufügen.

Hauptschleife des Algorithmus:

while min_heap:

In der Hauptschleife des Algorithmus werden nur die wichtigen Knoten berücksichtigt, die in important_nodes definiert sind. Der Prim-Algorithmus verwendet den Heap, um die Kante mit dem kleinsten Gewicht auszuwählen, die einen bereits im MST enthaltenen Knoten verbindet. Dies geschieht, bis der MST vollständig berechnet ist. Der berechnete MST wird schließlich von der visualize-Funktion angezeigt.

visualize():

Die visualize-Funktion nimmt den berechneten MST und visualisiert ihn mithilfe der networkx -Bibliothek. Dabei werden die wichtigen Knoten und deren Kanten im MST rot hervorgehoben. Die node_types und ihre Position im Graphen werden ebenfalls berücksichtigt. Der Graph ist als zweidimensionaler Array im Code definiert und kann auch über ein Menü eingelesen werden. Schließlich wird die visualize-Funktion aus der Datei F1.py aufgerufen, um den MST darzustellen

F2: Evakuierungsrouten planen (f2.py):

Ziel ist es, die Transportkapazitäten zu optimieren und die beste Route für die Evakuierung zu finden. Die Eingabe erfolgt in Form eines Graphen, welcher durch eine Adjazenzmatrix dargestellt wird. Die Knoten in diesem Graphen repräsentieren mögliche Standorte der Sammelstellen und Notlager, während die Kanten die Verbindungen zwischen diesen Standorten mit entsprechenden Entfernungen oder Kosten darstellen.

Funktionen und ihre Aufgaben:

vergleichen_und_klassifizieren():

Diese Funktion prüft die Unterschiede zwischen dem ursprünglichen und dem aktualisierten Stadtplan. Einbahnstraßen und unpassierbare Straßen werden erkannt, indem die Kanten beider Graphen verglichen werden. Die Ergebnisse werden in Listen gespeichert und ausgegeben.

max_flow(graph, source, sink):

Berechnet den maximalen Fluss von einer Quelle zu einem Ziel im Graphen. Der Algorithmus sucht mit der Breitensuche (BFS) nach Pfaden, um den Fluss schrittweise zu erhöhen, bis keine weiteren Verbesserungen möglich sind. Am Ende wird der maximale Fluss zurückgegeben.

alle_maximalen_fluesse(aktualisierter_stadtplan, nodes, sammelstellen, notlager):

Diese Funktion berechnet den maximalen Fluss für alle Sammelstellen-Notlager-Kombinationen. Dafür wird die max_flow-Funktion genutzt. Die Ergebnisse werden gesammelt und ausgegeben, um die Gesamtsituation im Stadtplan abzubilden.

check_capacity(sammelstellen, notlager):

Prüft, ob die Notlager genug Kapazität haben, um alle Personen aus den Sammelstellen aufzunehmen. Dazu werden die Kapazitäten der Sammelstellen mit denen der Notlager verglichen. Gibt True oder False zurück.

check_bus_capacity(sammelstellen, busse, kapazitaet_bus):

Stellt sicher, dass genügend Busse vorhanden sind, um alle Personen aus den Sammelstellen zu transportieren. Die Kapazität der Busse wird mit der Gesamtzahl der Personen verglichen. Gibt zurück, ob die Busse ausreichen.

allocate_buses():

Teilt die verfügbaren Busse optimal den Sammelstellen und Notlagern zu. Mithilfe der max_flow-Funktion wird der Transport effizient geplant. Nach jeder Zuweisung werden die verbleibenden Kapazitäten aktualisiert und ausgegeben.

run_simulation():

Führt die Simulation durch ein Menü, das verschiedene Optionen zur Evakuierungsplanung bietet. Der Benutzer kann Graphen laden, Berechnungen durchführen und Ergebnisse ansehen, bis er die Simulation beendet.

F3: Routenplanung für die Einsatzkräfte (f3.py):

Die Grundidee ist es, dass die Einsatzkräfte ihren Standort angeben, den Standort des Einsatzortes und aus diesen 2 Punkten den kürzesten Weg herauszugeben. Der kürzeste Weg welcher mithilfe des Dijkstra Algorithmus berechnet wird, wird in der Kommandozeile angezeigt, zusätzlich haben wir uns noch überlegt dass das Einsatzteam diesen Einsatz annehmen oder ablehnen kann, wenn er ihn annimmt wird seine Verfügbarkeit auf False gesetzt. Wenn der Einsatz beendet ist, kann man das in der Kommandozeile bestätigen und das Einsatzkraft-Team ist wieder verfügbar.

Funktionen und ihre Aufgaben:

dijkstra(grpah,start, ziel):

Der Dijkstra-Algorithmus arbeitet mit einer Prioritätswarteschlange (Min-Heap), um den kürzesten Pfad von einem Startknoten zu allen anderen Knoten in einem gewichteten Graphen zu finden. Zu Beginn werden alle Distanzen auf unendlich gesetzt, außer für den Startknoten, dessen Distanz auf 0 gesetzt wird. Eine Warteschlange speichert Knoten, die noch besucht werden müssen, beginnend mit dem Startknoten.

In der Hauptschleife wird der Knoten mit der geringsten Distanz aus der Warteschlange entnommen. Falls dieser Knoten bereits besucht wurde, wird er übersprungen. Für jeden benachbarten Knoten wird die Distanz überprüft und ggf. aktualisiert, wenn ein kürzerer Pfad gefunden wird. Neue Knoten werden dann wieder in die Warteschlange aufgenommen.

Am Ende gibt der Algorithmus die Liste dist mit den kürzesten Entfernungen und prev mit den Vorgängerknoten jedes Knotens im kürzesten Pfad zurück.

routenberechnung(prev, ziel):

Nach der Berechnung der kürzesten Distanzen mit Dijkstra wird der Pfad vom Zielknoten zum Startknoten anhand der prev-Liste rekonstruiert. Die Variable current wird auf das Ziel gesetzt, und eine leere Liste path wird angelegt, um den Pfad zu speichern. In einer Schleife wird der aktuelle Knoten zum Pfad hinzugefügt, und anschließend wird current auf den Vorgängerknoten gesetzt, der in der prev-Liste gespeichert ist. Dies wird fortgesetzt, bis der Startknoten erreicht ist (wenn current auf None gesetzt wird). Da der Pfad in umgekehrter Reihenfolge aufgebaut wird (vom Ziel zum Start), wird der Pfad nach der Schleife umgekehrt, sodass er die korrekte Reihenfolge vom Start- zum Zielknoten enthält. Der berechnete Pfad wird schließlich zurückgegeben.

run_einsatzplanun():

Die Eingabe für die Dijkstra-Funktion erfolgt durch zwei ganzzahlige Werte, die über die Kommandozeile abgefragt werden: start_input definiert den Startpunkt im Graphen, während ziel_input das Ziel festlegt. Das Ziel entspricht der Einsatzstelle, und der Startpunkt stellt die Position der Einsatzkraft dar. Diese beiden Werte werden in der Dijkstra-Funktion übergeben, um die kürzeste Entfernung zwischen Start und Ziel zu berechnen.

Die Rückgabe der Dijkstra-Funktion besteht aus den Variablen distanz und prev. distanz gibt die kürzeste Entfernung zwischen dem Start- und Zielknoten an, während prev die Vorgängerknoten enthält, die verwendet werden, um den kürzesten Pfad zu rekonstruieren.

Der berechnete Pfad wird dann als Liste von Knoten in der Variable path gespeichert, die den ab zulaufenden Weg angibt, zum Beispiel in der Form A → B → C.

Im nächsten Schritt wird eine Umsatzabfrage durchgeführt, bei der die Einsatzkraft gefragt wird, ob sie den Einsatz durchführen möchte. Falls die Antwort positiv ist, wird die Verfügbarkeit der Einsatzkraft auf False gesetzt. Bei Beendigung des Einsatzes wird ihre Verfügbarkeit auf True zurückgesetzt. Diese Änderung wird ebenfalls in der Kommandozeile angezeigt, um den aktuellen Status der Verfügbarkeit der Einsatzkräfte zu reflektieren.

Die Funktion zur Einsatzplanung wird im Menü durch den Aufruf von `f3.run_einsatzplanung()` ausgeführt.

F4: Versorgung von Einsatzkräften (f4.py):

Die Eingaben umfassen eine ungerichtete, gewichtete Adjazenzmatrix, bei der die Knoten Standorte der Einsatzstellen und Verpflegungspunkte darstellen und die Kanten die Verbindungen zwischen diesen Standorten mit entsprechenden Entfernungen oder Kosten repräsentieren. Zusätzlich werden die Koordinaten der Einsatzstellen und mögliche Verpflegungspunkte eingelesen.

Mithilfe des Dijkstra's Algorithmus und einem vereinfachten K-Means wird die optimale Positionierung der k Verpflegungspunkte ermittelt, um die durchschnittliche Entfernung der Einsatzkräfte zu minimieren und eine gleichmäßige Verteilung zu gewährleisten.

Funktionen und ihre Aufgaben:

`dijsktra(graph, start_node)`:

Berechnet die kürzesten Distanzen von einem Startknoten zu allen anderen Knoten im Graphen. Der Algorithmus nutzt eine Priority-Queue, um die Knoten nach minimaler Entfernung zu besuchen, und erstellt Pfade durch das Verfolgen der Elternknoten. Am Ende gibt es die Distanzen und die Pfade zurück.

vergleich_distance(random_nodes, einsatzstellen):

Vergleicht die Distanzen von zufällig ausgewählten Knoten zu den Einsatzstellen. Dafür wird für jeden Knoten der Dijkstra-Algorithmus ausgeführt. Die nächsten Knoten werden den Einsatzstellen zugewiesen, und es wird eine Zuweisungstabelle sowie die durchschnittlichen Entfernungen berechnet.

run_verpflegungspunkte():

Erlaubt es, eine Anzahl zufälliger Knoten auszuwählen, die nicht zu den Einsatzstellen gehören. Diese Knoten werden dann optimal den Einsatzstellen zugewiesen, basierend auf den Distanzen, die durch die vergleich_distance Funktion berechnet werden.

F5: Einsatzplanung für Einsatzkräfte (f5.py):

Die Grundidee ist mithilfe von Listen, welche die Einsatzkräfte, Teams und Einsatzstellen beinhalten, anhand eines Greedy Algorithmus Teams zu Einsatzstellen zuzuweisen.

class Einsatzkraft:

Mithilfe eines Konstruktors(__init__) wird ein neues Objekt der Klasse erstellt. Beim Erstellen eines neuen Objekts werden die folgenden Parameter übergeben

name: Name der Einsatzkraft

faehigkeiten: Eine Liste von Fähigkeiten, die die Einsatzkraft besitzt.

Ressourcen: Eine Liste von Ressourcen, die der Einsatzkraft zur Verfügung stehen.

standort: Der Standort der Einsatzkraft.

verfügbar: boolescher Wert, der angibt, ob die Einsatzkraft aktuell verfügbar ist.

Die Methode __repr__ ist zuständig, eine stringbasie Darstellung des Objekts zu liefern, wenn das Objekt als String ausgegeben wird

class Team:

Ein Team ist ein Zusammenschluss aus Einsatzkräften.

Mithilfe eines Konstruktors(__init__) wird ein neues Objekt der Klasse erstellt. Beim Erstellen eines neuen Objekts werden die folgenden Parameter übergeben

name: Name des Team

einsatzkraefte: Eine Liste von Einsatzkräften im Team

verfuegbar: ob das Team verfügbar ist.

Die Methode __repr__ gibt das Team, ihrer Eisatzstelle und ihre Verfügbarkeit aus.

class Einsatzstelle:

Mithilfe eines Konstruktors(__init__) wird ein neues Objekt der Klasse erstellt. Beim Erstellen eines neuen Objekts werden die folgenden Parameter übergeben.

Name: Der Name der Einsatzstelle

benoetigte_faehigkeiten: benötigte Fähigkeiten für die Einsatzstelle

benoetigte_ressourcen: benötigte Ressourcen für die Einsatzstelle

standort: der Standort der Einsatzstelle

Die Methode __repr__ gibt die Einsatzstelle, benötigte ressourcen, benötigte Fähigkeiten und den Standort

get_teams_and_einsatzstellen():

Die get_teams_and_einsatzstellen Funktion definiert die Einsatzkräfte für die jeweiligen Teams in einer Liste, und weist ihnen den Namen, Team A und Team B zu. Zusätzlich werden ein die Einsatzstellen in einer Liste festgehalten.

Return teams, einsatzstellen gibt die teams und einsatzstellen aus.

plan_einsatz(team_name=None):

Die Funktion nimmt optional einen Parameter team_name, der den Namen eines bestimmten Teams angibt, das zugewiesen werden soll. Wenn dieser Parameter nicht angegeben wird (team_name = None), werden alle Teams berücksichtigt.

Mit der Funktion get_teams_and_einsatzstellen() eine Liste von Teams und Einsatzstellen abgerufen.

Wenn der Parameter team_name angegeben wurde, wird die Liste teams auf das Team mit dementsprechenden Namen gefiltert. Falls kein Team mit diesem Namen gefunden wird, gibt die Funktion eine Fehlermeldung aus und beendet sich.

Wenn ein Team gefunden wird, wird die Liste auf dieses eine Team beschränkt (teams = teams[0:1) .

Eine leere Dictionary-Variable zugeordnete_einsatzstellen wird erstellt, um später die Zuordnung der Teams zu den Einsatzstellen zu speichern.

For einsatzstelle in einsatzstellen, wird eine Liste passende_teams erstellt, um Teams zu speichern die für die jeweilige Einsatzstelle geeignet sind. Jedes Team wird geprüft, ob alle benötigten Fähigkeiten der Einsatzstelle in den Fähigkeiten der Einsatzkräfte des Teams enthalten sind, ob alle benötigten Ressourcen der Einsatzstelle in den Ressourcen der Einsatzkräfte des Teams enthalten sind.

Wenn beide Bedingungen erfüllt sind und das Team noch verfügbar ist, wird das Team zur liste passende_teams hinzugefügt. Dieses Team aus der Liste wird ausgewählt und der Einsatzstelle zugeordnet, das Team und alle seine Einsatzkräfte werden auf nicht verfügbar gesetzt (team.verfuegbar = False und kraft.verfuegbar = False)

Nachdem alle Einsatzstellen überprüft wurden, gibt die Funktion eine Übersicht der Zuweisungen aus, die anzeigt, welches Team welcher Einsatzstelle zugewiesen wurde.

4. Verwendete Datenstrukturen

Heap

Für die Aufgabenstellungen F1, F3, F5 haben wir eine Heap-Datenstruktur verwendet, im genaueren ein Binary Min-Heap, diese ist eine spezielle Art des binären Heaps. Die Elemente sind so angeordnet, dass für jedes Elternelement der Wert des Eltern Knotens kleiner oder gleich dem Wert seiner Kindknoten ist. So ist der kleinste Wert immer an der Spitze des Heaps.
In unseren Aufgabenstellungen ist das besonders hilfreich, wenn man das kleinste Element aus einer Sammlung extrahieren möchte. Dieses Prinzip verwenden wir für die Implementierung von Prioritätswarteschlangen.

List

Listen sind flexibel und bieten schnellen Zugriff auf Elemente über Indizes. Sie eignen sich gut für die Speicherung und Bearbeitung von geordneten Datensammlungen wie Knoten, Kapazitäten und Pfaden.
Verwendung in Funktion 2 und 4.

2D-Array (list of lists)

2D-Arrays sind geeignet für die Darstellung von Adjazenzmatrizen, da sie eine einfache und effiziente Möglichkeit bieten, Verbindungen und Kapazitäten zwischen Knoten zu speichern und zu bearbeiten.

Implementierung in alle Funktionen.

Dictionary (dict)

Dictionaries ermöglichen schnellen Zugriff auf Werte über Schlüssel. Sie sind nützlich, um die verbleibende Kapazität der Notlager zu verfolgen.
Implementierung in F2 und F4

5. Verwendete Algorithmen und ihre Laufzeit

n = Anzahl der Knoten im Graphen

E = Anzahl der Kanten im Graphen

F1: Prim

Der Prim Algorithmus ist eine Form des greedy Algorithmus,der dazu verwendet wird den Minimalen Spannbaum(MST) eines ungerichteten, gewichteten Graphen zu berechnen.Der Algorithmus beginnt mit einem beliebigen Startknoten, dieser wird mit einem Gewicht von 0 in die Min-Heap-Datenstruktur eingefügt und wächst schrittweise den Spannbaum, indem er in jedem Schritt die Kante mit dem geringsten Gewicht auswählt, die einen bereits besuchten Knoten mit einem noch nicht besuchten Knoten verbindet.

Min-heap
heapopp = O(n log n)
heappush = O(E log n)

best case:

o(nlogn) Wenn Graph wenige Kanten hat , realistischer bei Adjanzlisten, weil benachbarte Knoten bereit betrachtet werden müssen, was bei wenigen Kanten schneller geht.

average case:

$O(n^2)$ weil bei einer Adjazenzmatrix n * n einträge betrachtet werden müssen

worst case:

$O(n^2)$ für den worst case gilt es wie beim average dass auch hier im schlimmsten Fall alle n * n einträge der Adjazenzmatrix betrachtet werden

F2: Ford-Fulkerson

Der Ford-Fulkerson-Algorithmus wird verwendet, um den maximalen Fluss in einem Flussnetzwerk zu berechnen. Dabei wird von einer Quelle (source) zu einem Senke (sink) der größtmögliche Fluss ermittelt, der durch das Netzwerk transportiert werden kann. Der Algorithmus arbeitet iterativ und verbessert den Fluss, indem er in jedem Schritt einen sogenannten augmentierenden Pfad findet und den Fluss entlang dieses Pfades erhöht. Für die Pfadsuche wird häufig die Breitensuche (BFS) verwendet, um den schnellsten Pfad zu finden.

BFS für Pfadsuche

BFS-Komplexität: O(E), wobei E die Anzahl der Kanten im Graphen ist. Dies bedeutet, dass in jedem Schritt alle Kanten untersucht werden müssen, um einen augmentierenden Pfad zu finden.

Best Case:

O(E) – Wenn der Graph wenige Kanten hat, wird in jedem Schritt nur eine geringe Anzahl von Kanten untersucht.

Average Case:

O(V * E) – Im Durchschnitt muss der Algorithmus mehrere Iterationen durchführen, wobei jede Iteration den Graphen nach einem augmentierenden Pfad durchsucht. Hierbei müssen in der Regel O(E) Kanten pro Iteration betrachtet werden und der Algorithmus läuft O(V) Iterationen, da jeder Knoten im schlimmsten Fall einmal besucht wird.

Worst Case:

O(E * F) – Im schlimmsten Fall, wenn der Fluss sehr klein ist, kann es erforderlich sein, den Algorithmus so lange auszuführen, bis der gesamte Fluss (F) transportiert wurde. Hierbei wird in jedem Schritt ein Pfad mit minimalem Fluss gefunden.

F3/F4: Dijkstra

Für den Dijkstra-Algorithmus verwenden wir einen Min-Heap, weil der kleinste Wert immer an der Wurzel des Heaps liegt. Dadurch können wir schnell den Knoten mit der geringsten Entfernung finden und effizient verarbeiten. Da der Min-Heap die Knoten nach ihrer Distanz sortiert, können wir sie in der richtigen Reihenfolge abarbeiten, was den Algorithmus schneller macht und die Laufzeit optimiert.

Min-heap
heapopp = O(n log n)
heappush = O(E log n)

Daraus folgt die Zeitkomplexität des Dijkstra-Algorithmus mit einer Min-Heap-Datenstruktur ist: O((E + n) logn)

Worst Case:

Im schlimmsten Fall müssen alle Kanten überprüft werden, und für jeden Knoten wird eine Extraktion mit heappop aus der Prioritätswarteschlange durchgeführt. Daraus folgt:

O((E + n) log n)

Worst Case tritt auf wenn viele Kanten überprüft werden müssen

Average Case:

Im Durchschnitt müssen nicht alle Kanten überprüft werden, sodass die Anzahl der Iterationen über die Knoten im Durchschnitt geringer ist als im Worst Case.

Die average Zeitkomplexität ist daher immer noch:

O((E + n) log n)

Der Algorithmus könnte für alle Knoten extrahieren, aber aufgrund der geringeren Anzahl an Kanten wird er etwas schneller sein als im Worst Case

Best Case:

Ein Bester Fall wäre wenn der Startknoten der Zielknoten wäre, dann ist die Komplexität bei O(1)

Der beste Fall tritt auf, wenn der Algorithmus einen Graphen vorliegen hat, welcher nur über wenige Kanten verfügt, trotzdem muss er auch wie im Durchschnittsfall :

O((E + n) log n)

ablaufen. welcher aber schneller ausgeführt wird, wegen der Gruppengröße.

F5: Greedy

Ein Greedy-Algorithmus geht nach dem einfachen Prinzip vor, dass er im jetzigen Zeitpunkt den optimalen Wert nimmt und diesen auch beibehält. In unserem Programm wird geschaut, dass alle Voraussetzungen vorhanden sind, also das ein Team die gebrauchten Ressourcen und Fähigkeiten besitzt, wenn das der Fall ist, wird das Team in eine Prioritätswarteschlange eingetragen, aus dieser Prioritätswarteschlange wird dann der erste Eintrag zum passenden Einsatz zugeteilt.
n = Anzahl der Teams
m = Anzahl der Einsatzstellen
k = Anzahl der Einsatzkräfte pro Teams
f = Anzahl der benötigten Fähigkeiten pro Einsatzstelle
s = Anzahl der benötigten Ressourcen pro Einsatzstelle

Überprüfung der Anforderungen:

$O(k * (f + s))$

Min-heap Operationen:
heapopp = $O(m \log n)$
heappush = $O(m \log n)$

Worst Case:
Wenn für jede Einsatzstelle alle Teams in der Prioritätswarteschlange eingefügt werden müssen und es für jede eine heappop Operation gibt. Daraus folgt:

$O(m * n * k * (f + s)) + O(m * \log n)$

Average Case:
Der Durchschnitt hängt von der Anzahl der Teams ab, die in die Prioritätswarteschlange eingefügt werden. Daraus folgt:

$O(m * n * k(f + s)) + O(m * \log n)$

Best Case:
Wenn es keine Teams für Einsatzstellen gibt, ist die Prioritätswarteschlange leer. Wir haben also nur die Überprüfung der Anforderungen:

$O(m * n * k * (f+s))$