Proiect IA - 2024
Gestionarea transporturilor intr-o firma de curierat

Parvana Vlad-Stefan
Radion Ciprian

1. Descrierea problemei considerate

Această aplicație software a fost dezvoltată pentru a optimiza rutele de transport pentru un serviciu de curierat, care operează în 10 orașe principale din România: București, Cluj-Napoca, Timișoara, Iași, Constanța, Brașov, Craiova, Galați, Oradea și Sibiu. Problema principală abordată este determinarea unei rute optime pentru livrările de colete, pornind de la un sediu principal și vizitând mai multe orașe destinație, într-o ordine care minimizează distanța totală parcursă. Acest lucru include și determinarea celui mai eficient drum între două orașe date, luând în considerare infrastructura de drumuri existentă.

2. Aspecte teoretice privind algoritmul

Această aplicație utilizează doi algoritmi principali de optimizare:

Algoritmul A* (A-star): Este folosit pentru a găsi cel mai scurt drum între două orașe date. A* este un algoritm de căutare informată, care combină costul real al drumului parcurs (distanța rutieră) cu o estimare heuristică a distanței rămase până la destinație (distanța geografică în linie dreaptă). Această combinație permite găsirea rapidă a celui mai scurt drum, evitând explorarea inutilă a unor rute mai puțin probabile.

Particle Swarm Optimization (PSO): Este utilizat pentru a optimiza ordinea în care sunt vizitate orașele destinație. PSO este un algoritm metaheuristic inspirat din comportamentul social al stolurilor de păsări, care caută soluția optimă prin iterarea și ajustarea pozițiilor particulelor într-un spațiu de căutare. În acest caz, fiecare particulă reprezintă o posibilă ordine a orașelor, iar poziția sa este codificată printr-un random key encoding. Algoritmul modifică viteza și poziția particulelor pe baza celui mai bun rezultat al particulelor și a celui mai bun rezultat global.

3. Modalitatea de rezolvare

Pentru a rezolva problema optimizării rutelor, aplicația a fost dezvoltată urmând următorii pași principali:

1. Configurarea mediului: Inițializarea datelor despre orașe (coordonate geografice) și drumuri (distanțe).

2. Interfața Grafică (GUI): Implementarea unei interfețe intuitive cu ajutorul bibliotecii Tkinter, care permite utilizatorului să interacționeze cu sistemul.

3. Adăugarea sediilor: Utilizatorul are posibilitatea de a adăuga sedii principale pe hartă.

4. Selecția destinațiilor: Utilizatorul poate alege sediul de plecare și orașele destinație dintr-o listă.

5. Calculul rutei optime: Sistemul utilizează PSO pentru a determina ordinea optimă a orașelor și A* pentru a calcula drumurile între fiecare două orașe consecutive.

6. Afișarea rezultatelor: Ruta optimă este afișată grafic pe hartă, iar rezultatele sunt prezentate sub forma textului, inclusiv lista de orașe vizitate și distanța totală parcursă.

4. Listarea părților semnificative din codul sursă însoțite de explicații și comentarii

- Clasa CourierSystem: Aceasta este clasa principală a aplicației și conține metodele pentru inițializarea datelor, configurarea interfeței grafice, gestionarea plasării sediilor și calculul rutei optime.

```python
class CourierSystem:
    """
    Clasa ce implementeaza sistemul de transport a unei firme de curierat utilizand
    A* pentru gasirea drumurilor si Partiecle Swarm Optimization pentru determinarea ordinii optime
    a vizitarii oraselor + Random Key Encoding
    """
```

```
def __init__(self, root):
def setup_gui(self):
def update_destination_cities(self, event=None):
def load_map(self):
def enable_hq_placement(self):
def place_headquarters(self, event):
def get_clean_city_name(self, city: str) -> str:
def calculate_route(self):
def get_neighbors(self, city: str) -> List[str]:
def calculate_geographical_distance(self, city1: str, city2: str) -> float:
def get_road_distance(self, city1: str, city2: str) -> float:
def a_star(self, start: str, goal: str) -> Tuple[List[str], float]:
def pso_tsp(self, cities: List[str], start_city: str) -> List[str]:
```

- Metoda a_star(self, start, goal): Această metodă implementează algoritmul A* pentru a găsi cel mai scurt drum între două orașe. Utilizează distanța geografică în linie dreaptă ca euristică și distanțele rutiere actuale ca costuri.

Python

```
  def a_star(self, start: str, goal: str) -> Tuple[List[str], float]:
      """
      Implementare a algoritmului A* pentru gasirea celui mai scurt drum intre doua
orașe,
      folosind distanta geografica ca euristică și distanta rutiera ca cost real
      """
      frontier = [(0, start, [start])]  # (f_score, oras, path) - Coada de prioritate
      explored = set()  # Set pentru orasele deja vizitate
      g_score = {start: 0}  # Costul real până la fiecare oraa

      while frontier:
          f, current, path = heapq.heappop(frontier) # Obtinem nodul cu cel mai mic f_score

          if current == goal:
              # Împarțim distanta totala la 2 pentru a corecta dublarea
              return path, g_score[current] / 2 # returnăm path-ul și costul

          if current in explored:
```

```
            continue # Daca nodul a fost vizitat, trecem la următorul

        explored.add(current)

        for neighbor in self.get_neighbors(current):
            if neighbor in explored:
                continue # Daca vecinul a fost vizitat, trecem la următorul

            road_distance = self.get_road_distance(current, neighbor) # Calculam costul de la nodul curent la vecin
            tentative_g = g_score[current] + road_distance # Costul estimat pana la vecin

            if neighbor not in g_score or tentative_g < g_score[neighbor]: # Daca gasim un drum mai bun catre vecin
                g_score[neighbor] = tentative_g # Actualizam costul real
                h_score = self.calculate_geographical_distance(neighbor, goal) # Calculam euristica
                f_score = tentative_g + h_score # Calculam f_score
                heapq.heappush(frontier, (f_score, neighbor, path + [neighbor])) # Adaugam nodul în coada de prioritate

    return None, float('inf') # Daca nu am gasit niciun drum, returnam None și infinit
```

- Metoda pso_tsp(self, cities, start_city): Această metodă implementează algoritmul Particle Swarm Optimization (PSO) pentru problema călătorului comercial (TSP), determinând ordinea optimă de vizitare a orașelor.

Python

```
def pso_tsp(self, cities: List[str], start_city: str) -> List[str]:
    """
    Implementarea algoritmului Particle Swarm Optimization (PSO) pentru rezolvarea problemei Traveling Salesman Problem (TSP), folosind Random Key Encoding
    """
    clean_cities = [self.get_clean_city_name(city) for city in cities]
```

```
        clean_start_city = self.get_clean_city_name(start_city)

        num_cities = len(clean_cities)
        if num_cities <= 1:
            return cities

        # Parametri PSO
        num_particles = 30 # Numarul de particule
        num_iterations = 50 # Numarul de iterații
        w = 0.7  # inertia
        c1 = 2.0  # factorul cognitiv
        c2 = 2.0  # factorul social

        particles = np.random.uniform(low=0.0, high=1.0, size=(num_particles,
num_cities)) # Initializam particulele cu chei aleatorii
        velocities = np.zeros((num_particles, num_cities)) # Initializam vitezele particulelor
cu 0

        pbest = particles.copy() # Retinem cele mai bune poziții ale particulelor
        pbest_fitness = np.full(num_particles, float('inf')) # Initializam fitness-ul celor mai
bune poziții cu infinit

        gbest = particles[0].copy()  # Retinem cea mai buna pozitie globala
        gbest_fitness = float('inf') # Initializam fitness-ul celei mai bune poziții globale cu
infinit

        def random_key_to_route(keys: np.ndarray) -> List[str]:
            """
            Converteste cheile aleatorii intr-o ruta valida
            """
            city_keys = list(zip(keys, cities)) # Asociem cheile cu orasele
            sorted_pairs = sorted(city_keys, key=lambda x: x[0]) # Sortam perechile după
cheie
            route = [pair[1] for pair in sorted_pairs] # Obtinem ruta sortata după chei

            if start_city in route:
                start_idx = route.index(start_city)
```

```
            route = route[start_idx:] + route[:start_idx]  # Asiguram că ruta începe cu sediul

            return route

        def calculate_fitness(route: List[str]) -> float:
            """
            Calculeaza lungimea totala a rutei
            """
            if self.get_clean_city_name(route[0]) != clean_start_city: # Verificam dacă ruta începe cu sediul
                return float('inf')

            total_distance = 0
            for i in range(len(route)):
                city1 = self.get_clean_city_name(route[i])
                city2 = self.get_clean_city_name(route[(i + 1) % len(route)])
                path, dist = self.a_star(city1, city2) # Calculam costul drumului între orașe
                if path is None:
                    return float('inf')
                total_distance += dist # Adaugam costul la distanța totala
            return total_distance # Returnam distanța totală

            # Procesul de optimizare PSO

        for iteration in range(num_iterations):
            for i in range(num_particles):
                particles[i] = particles[i] + velocities[i] # Actualizam pozitiile particulelor
                particles[i] = np.clip(particles[i], 0, 1)  # Restrangem cheile între 0 și 1

                current_route = random_key_to_route(particles[i]) # Obtinem ruta corespunzatoare poziției
                current_fitness = calculate_fitness(current_route) # Calculam fitness-ul rutei

                if current_fitness < pbest_fitness[i]: # Daca fitness-ul curent e mai bun decat cel mai bun al particulei
                    pbest[i] = particles[i].copy()
                    pbest_fitness[i] = current_fitness
```

```
            if current_fitness < gbest_fitness: # Daca fitness-ul curent e mai bun decât cel mai bun global
                gbest = particles[i].copy()
                gbest_fitness = current_fitness

        for i in range(num_particles):
            r1, r2 = np.random.rand(2) # Generam 2 numere aleatorii pentru componentele cognitiva și sociala
            cognitive_velocity = c1 * r1 * (pbest[i] - particles[i]) # Calculam componenta cognitiva
            social_velocity = c2 * r2 * (gbest - particles[i]) # Calculam componenta socială
            velocities[i] = w * velocities[i] + cognitive_velocity + social_velocity # Actualizam vitezele particulelor

    return random_key_to_route(gbest) # Returnam ruta corespunzatoare celei mai bune poziții globale
```

- Metoda calculate_route(self): Aceasta coordonează procesul de calculare a rutei, folosind PSO pentru a stabili ordinea vizitării orașelor și A* pentru găsirea drumului între ele.

Python

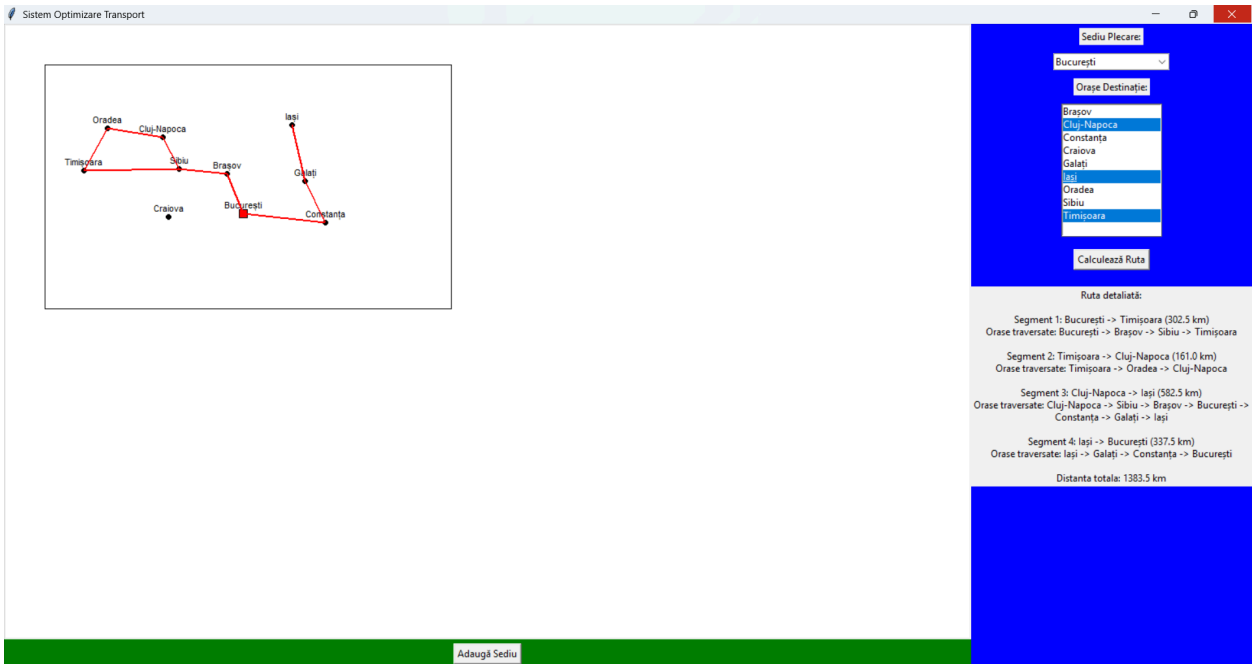
```
  def calculate_route(self):
      """
      Calculeaza și afișeaza ruta optima, inclusiv orașele intermediare, folosind PSO pentru ordinea orașelor
      și A* pentru determinarea rutelor între ele.
      """
      start_city = self.hq_select.get()
      if not start_city:
          messagebox.showerror("Eroare", "Selectati un sediu de plecare!")
          return

      selected_indices = self.dest_listbox.curselection()
      if not selected_indices:
          messagebox.showerror("Eroare", "Selectati cel puțin un oras destinatie!")
          return
```

```
        # Obținem orasele destinatie
        destinations = [self.dest_listbox.get(idx) for idx in selected_indices]
        cities = [start_city] + destinations

        # Calculam ruta optima folosind PSO
        optimal_route = self.pso_tsp(cities, start_city)

        # Calculam distanța totala și construim path-ul complet folosind A*
        total_distance = 0
        detailed_route = []
        segment_distances = []

        for i in range(len(optimal_route)):
            city1 = optimal_route[i]
            city2 = optimal_route[(i + 1) % len(optimal_route)]

            # Folosim numele curate pentru calculul rutei
            clean_city1 = self.get_clean_city_name(city1)
            clean_city2 = self.get_clean_city_name(city2)

            path, dist = self.a_star(clean_city1, clean_city2)
            if path is None:
                messagebox.showerror("Eroare", f"Nu s-a putut gasi o ruta intre {city1} și
{city2}")
                return

            if i < len(optimal_route) - 1:
                detailed_route.extend(path[:-1])
            else:
                detailed_route.extend(path)

            segment_distances.append((city1, city2, dist))
            total_distance += dist

        # Construim textul rezultatului
        result_text = "Ruta detaliată:\n"
```

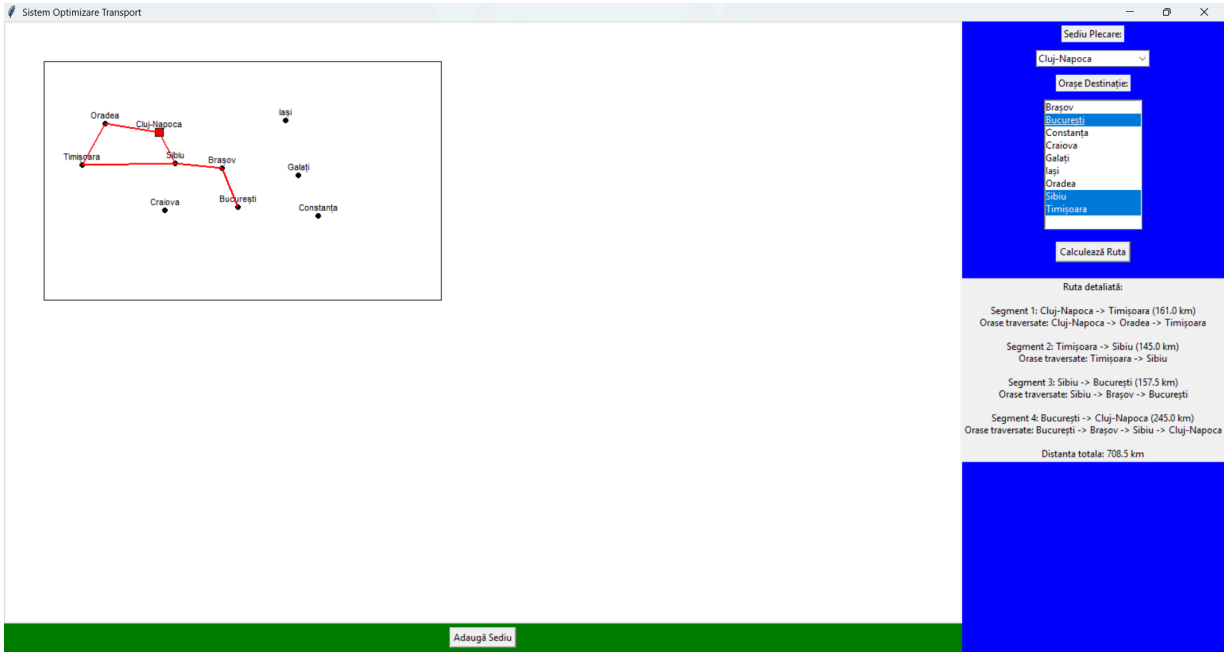
```
        for i in range(len(segment_distances)):
            city1, city2, dist = segment_distances[i]
            clean_city1 = self.get_clean_city_name(city1)
            clean_city2 = self.get_clean_city_name(city2)
            path, _ = self.a_star(clean_city1, clean_city2)

            result_text += f"\nSegment {i + 1}: {city1} -> {city2} ({dist:.1f} km)\n"
            result_text += f"Orase traversate: {' -> '.join(path)}\n"

        result_text += f"\nDistanta totala: {total_distance:.1f} km"
        self.result_label.config(text=result_text)

        # Desenam ruta pe harta
        self.map_canvas.delete("route")
        for i in range(len(detailed_route) - 1):
            city1, city2 = detailed_route[i], detailed_route[i + 1]
            x1 = 50 + (self.cities[city1][1] - 20) * 40
            y1 = 50 + (49 - self.cities[city1][0]) * 40
            x2 = 50 + (self.cities[city2][1] - 20) * 40
            y2 = 50 + (49 - self.cities[city2][0]) * 40
            self.map_canvas.create_line(x1, y1, x2, y2, fill='red', width=2, tags="route")
```

5. Rezultatele obținute prin rularea programului în diverse situații, capturi ecran și comentarii asupra rezultatelor obținute

- Test 1: Un scenariu testat a implicat adăugarea unui sediu în București și selecția orașelor Cluj-Napoca, Timișoara și Iași ca destinații. Rezultatul obținut a generat o rută care vizitează aceste orașe într-o ordine optimă, minimizând distanța totală parcursă.



- Test 2: Un alt test a fost realizat cu sediul plasat în Cluj-Napoca și având ca destinații Timișoara, Sibiu și București. În acest caz, algoritmul a determinat o rută optimă diferită, reflectând schimbarea poziției sediului de plecare.



Rezultatele testelor indică eficacitatea algoritmilor în determinarea rutelor optime în diferite scenarii, luând în considerare atât ordinea de vizitare a orașelor cât și distanța minimă dintre ele. Capturi de ecran ar fi incluse aici în documentația fizică.

6. Concluzii

Acest sistem de optimizare a transportului utilizează algoritmi avansați, precum A* și PSO, pentru a oferi o soluție eficientă și practică pentru planificarea rutelor de curierat. Interfața grafică intuitivă, împreună cu rezultatele exacte, fac din această aplicație un instrument util pentru managementul operațiunilor de livrare. Performanța algoritmilor este demonstrată prin diverse scenarii testate, confirmând că aplicația poate gestiona cu succes multiple destinații și configurații diferite.

7. Bibliografie

- http://florinleon.byethost24.com/curs_ia.html
- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.geeksforgeeks.org/particle-swarm-optimization-pso-an-overview/
- https://kalami.medium.com/understanding-random-key-encoding-a-simple-approach-to-solving-complex-optimization-problems-ab17ee028f66
- https://www.geeksforgeeks.org/python-gui-tkinter/

8. O listă cu ce a lucrat fiecare membru al echipei

- Parvana Vlad-Stefan: A implementat algoritmii A* și PSO, a creat structura generală a aplicației și a realizat unele elemente de interfata. S-a ocupat de logica și implementarea backend-ului.

- Radion Ciprian: A lucrat la implementarea interfeței grafice cu ajutorul Tkinter, la adăugarea de funcționalități pentru plasarea sediilor pe hartă, la interacțiunea cu utilizatorul și s-a ocupat de testarea interfeței și funcționalităților. A contribuit la implementarea backend-ului, mai precis, la corectarea unor erori de functionare, adaugarea de Random Key Encoding si realizarea testelor benchmark.