Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра ІПІ

Звіт

з лабораторної роботи №8 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

„ **Евристичні алгоритми**”

Виконав ІП-45 Янов Богдан Євгенійович

Перевірив Соколовський Владислав Володимирович

Київ 2025

Лабораторна робота №8

Евристичні алгоритми

Мета: вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку оцінити їх ефективність на різних структурах даних.

Задача комівояжера (Traveling Salesman Problem, TSP) - це класична задача, в якій потрібно знайти найкоротший можливий маршрут, що проходить через усі задані міста рівно один раз і повертається в початкове місто. Вона має широке застосування в логістиці, плануванні маршрутів та інших сферах.

Задача належить до класу NP-повних. Для знаходження точного розв'язку перебором потрібно перевірити всі можливі перестановки міст - $(n-1)!$ варіантів. Це робить повний перебір неможливим при великій кількості міст.

У цій роботі використано жадібний алгоритм найближчого сусіда, який працює за принципом:

1. Стартуємо з довільного міста.
2. На кожному кроці переходимо в найближче невідвідане місто.
3. Повторюємо крок 2, поки не відвідаємо всі міста.
4. Повертаємося до стартового міста.

Часова складність: O(n^2), що значно ефективніше, ніж перебір. Хоча цей алгоритм не гарантує оптимального рішення, він дозволяє швидко отримати допустимий маршрут навіть при великій кількості міст.

Псевдокод алгоритмів

```
validate_distance_matrix(dist) {
      if dist.empty() {
            return TspError::EmptyGraph
      }
      n = dist.len()
      for row in dist {
            if row.len() != n {
                  return TspError::InvalidDistanceMatrix
            }
      }
      for i in 0..n {
            if dist[i][i] != 0 {
                  return TspError::InvalidDistanceMatrix
            }
            for j from 0 to n {
                  if dist[i][j] != dist[j][i] {
                  return TspError::InvalidDistanceMatrix
                  }
            }
      }
}
```

```
find_nearest_neighbor_tour(dist, start) {
      let n = dist.len()
      if start >= n {
            return TspError::InvalidStartCity
      }
      visited = [] of len n
      tour = [] of len n+1
      total_dist = 0u32
      current = start
      visited[current] = true
      tour.push(current)

      for _ from 1 to n {
            next_city = None
            best_distance = max number
            for j, is_visited in visited enumerate() {
                  if !is_visited && dist[current][j] > 0 && dist[current][j] <
      best_distance {
                  best_distance = dist[current][j]
                  next_city = j
                  }
            }
            next = next_city or return TspError::NoValidPath
            visited[next] = true
            total_dist = total_dist + best_distance
            tour.push(next)
            current = next
      }
      if dist[current][start] == 0 {
            return TspError::NoValidPat)
      }
      total_dist = total_dist + dist[current][start]
            or return TspError::NoValidPath;
      tour.push(start)
      return (tour, total_dist)
}
```

Вхідні дані

	Amsterdam	Rotterdam	Den Haag	Utrecht	Eindhoven	Tilburg	Groningen	Breda	Nijmegen	Enschede	Apeldoorn	Haarlem	Arnhem	Amersfoort	Maastricht
Amsterdam	0	80	60	45	125	115	180	120	130	170	100	20	130	75	215
Rotterdam	80	0	25	65	115	95	255	55	115	215	150	100	185	120	185
Den Haag	60	25	0	75	130	110	270	75	135	230	160	65	180	130	195
Utrecht	45	65	75	0	115	100	200	110	90	150	50	65	75	20	180
Eindhoven	125	115	130	115	0	25	240	65	80	165	140	185	125	135	80

Tilburg	115	95	110	100	25	0	260	35	115	195	155	150	145	140	95
Groningen	180	255	270	200	240	260	0	300	190	175	170	210	180	170	345
Breda	120	55	75	110	65	35	300	0	105	205	165	115	155	130	120
Nijmegen	130	115	135	90	80	115	190	105	0	120	95	170	25	105	160
Enschede	170	215	230	150	165	195	175	205	120	0	115	195	145	130	265
Apeldoorn	100	150	160	50	140	155	170	165	95	115	0	115	60	55	200
Haarlem	20	100	65	65	185	150	210	115	170	195	115	0	145	85	225
Arnhem	130	185	180	75	125	145	180	155	25	145	60	145	0	65	180
Amersfoort	75	120	130	20	135	140	170	130	105	130	55	85	65	0	175
Maastricht	215	185	195	180	80	95	345	120	160	265	200	225	180	175	0

Вихідний код

```
use std::fmt;

#[derive(Debug)]
enum TspError {
    EmptyGraph,
    InvalidStartCity,
    NoValidPath,
    InvalidDistanceMatrix,
}

impl fmt::Display for TspError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            TspError::EmptyGraph => write!(f, "Empty graph
provided"),
            TspError::InvalidStartCity => write!(f, "Invalid start
city index"),
            TspError::NoValidPath => write!(f, "No valid path
found"),
            TspError::InvalidDistanceMatrix => write!(f, "Invalid
distance matrix"),
```

```rust
        }
    }
}

fn validate_distance_matrix(dist: &[Vec<u32>]) -> Result<(),
TspError> {
    if dist.is_empty() {
        return Err(TspError::EmptyGraph);
    }

    let n = dist.len();
    for row in dist {
        if row.len() != n {
            return Err(TspError::InvalidDistanceMatrix);
        }
    }

    // symmetric and diagonal == zero
    for i in 0..n {
        if dist[i][i] != 0 {
            return Err(TspError::InvalidDistanceMatrix);
        }
        for j in 0..n {
            if dist[i][j] != dist[j][i] {
                return Err(TspError::InvalidDistanceMatrix);
            }
        }
    }

    Ok(())
}

fn find_nearest_neighbor_tour(
    dist: &[Vec<u32>],
    start: usize,
) -> Result<(Vec<usize>, u32), TspError> {
```

```rust
    validate_distance_matrix(dist)?;

    let n = dist.len();
    if start >= n {
        return Err(TspError::InvalidStartCity);
    }

    let mut visited = vec![false; n];
    let mut tour = Vec::with_capacity(n + 1);
    let mut total_dist = 0u32;

    let mut current = start;
    visited[current] = true;
    tour.push(current);

    for _ in 1..n {
        let mut next_city = None;
        let mut best_distance = u32::MAX;

        for (j, &is_visited) in visited.iter().enumerate() {
            if !is_visited && dist[current][j] > 0 &&
dist[current][j] < best_distance {
                best_distance = dist[current][j];
                next_city = Some(j);
            }
        }

        let next = next_city.ok_or(TspError::NoValidPath)?;
        visited[next] = true;
        total_dist = total_dist
            .checked_add(best_distance)
            .ok_or(TspError::NoValidPath)?;
        tour.push(next);
        current = next;
    }
```

```rust
    if dist[current][start] == 0 {
        return Err(TspError::NoValidPath);
    }

    total_dist = total_dist
        .checked_add(dist[current][start])
        .ok_or(TspError::NoValidPath)?;
    tour.push(start);

    Ok((tour, total_dist))
}

fn main() {
    let cities = vec![
        "Amsterdam",
        "Rotterdam",
        "Den Haag",
        "Utrecht",
        "Eindhoven",
        "Tilburg",
        "Groningen",
        "Breda",
        "Nijmegen",
        "Enschede",
        "Apeldoorn",
        "Haarlem",
        "Arnhem",
        "Amersfoort",
        "Maastricht",
    ];

    #[rustfmt::skip]
    let dist: Vec<Vec<u32>> = vec![
        //    Ams   Rot   DHA   Utr   Eind  Tilb  Gron  Breda  Nijm
Ens   Apel   Haar   Arn   Amfs  Maast
        vec![  0,   80,   60,   45,   125,  115,  180,  120,   130,
```

```
170,  100,  20,  130,  75,  215 ], // Amsterdam
      vec![  80,  0,   25,  65,  115,  95,  255,  55,  115,
215,  150,  100,  185,  120,  185 ], // Rotterdam
      vec![  60,  25,   0,  75,  130,  110,  270,  75,  135,
230,  160,  65,  180,  130,  195 ], // Den Haag
      vec![  45,  65,  75,   0,  115,  100,  200,  110,  90,
150,  50,  65,  75,  20,  180 ], // Utrecht
      vec![ 125, 115,  130,  115,  0,  25,  240,  65,  80,
165,  140,  185,  125,  135,  80 ], // Eindhoven
      vec![ 115,  95,  110,  100,  25,  0,  260,  35,  115,
195,  155,  150,  145,  140,  95 ], // Tilburg
      vec![ 180, 255,  270,  200,  240,  260,  0,  300,  190,
175,  170,  210,  180,  170,  345 ], // Groningen
      vec![ 120,  55,  75,  110,  65,  35,  300,  0,  105,
205,  165,  115,  155,  130,  120 ], // Breda
      vec![ 130, 115,  135,  90,  80,  115,  190,  105,  0,
120,  95,  170,  25,  105,  160 ], // Nijmegen
      vec![ 170, 215,  230,  150,  165,  195,  175,  205,  120,
0,  115,  195,  145,  130,  265 ], // Enschede
      vec![ 100, 150,  160,  50,  140,  155,  170,  165,  95,
115,  0,  115,  60,  55,  200 ], // Apeldoorn
      vec![  20, 100,  65,  65,  185,  150,  210,  115,  170,
195,  115,  0,  145,  85,  225 ], // Haarlem
      vec![ 130, 185,  180,  75,  125,  145,  180,  155,  25,
145,  60,  145,  0,  65,  180 ], // Arnhem
      vec![  75, 120,  130,  20,  135,  140,  170,  130,  105,
130,  55,  85,  65,  0,  175 ], // Amersfoort
      vec![ 215, 185,  195,  180,  80,  95,  345,  120,  160,
265,  200,  225,  180,  175,  0 ], // Maastricht
  ];

  match find_nearest_neighbor_tour(&dist, 0) {
      Ok((tour, total_dist)) => {
          print!("Path: ");
          for (i, &idx) in tour.iter().enumerate() {
              if i > 0 {
```

```
                print!(" → ");
            }
            print!("{}", cities[idx]);
        }
        println!(", Length: {} км", total_dist);
    }
    Err(e) => {
        eprintln!("Error: {}", e);
    }
  }
}
```

Приклад роботи

```
λ Partur lab8 → λ git master* → cargo run --release
    Finished `release` profile [optimized] target(s) in 0.00s
     Running `target/release/lab8`
Path: Amsterdam → Haarlem → Den Haag → Rotterdam → Breda → Tilburg → Eindhoven → Nijmegen → Arnhem → Apeldoorn → Utrecht → Amersfoort →
 Enschede → Groningen → Maastricht → Amsterdam, Length: 1325 км
λ Partur lab8 → λ git master* →
```

Висновок

У ході виконання роботи було реалізовано програму мовою Rust для розв’язання задачі комівояжера методом жадібного пошуку - найближчого сусіда. Було обрано 15 міст Нідерландів, для яких зафіксовано міжміські відстані (у випадку прямого сполучення), та побудовано маршрут, що проходить через усі міста з поверненням до початкової точки.