Sztuczna inteligencja - lista 1

Mateusz Polito 266581

0. Informacje ogólne

Jako język programowania wybrałem Rust. Do obsługi grafów użyłem biblioteki petgraph. Do przechowywania danych o przystankach i połączeniach między nimi stworzyłem struktury

```rust
struct BusRoute {
  company: String,
  line: String,
  departure_time: MyTime,
  arrival_time: MyTime,
}
```

i

```rust
struct BusStop {
    id: Option<NodeIndex>,
    name: String,
    coords: Coords,
}

struct Coords {
    lat: f32,
    lon: f32,
}
```

Aby przechowywać czas stworzyłem strukturę:

```rust
struct MyTime {
    hour: i32,
    minute: i32,
}
```

1. Przetworzenie danych

Oczytywanie danych i budowa grafu odbywa się w poniższej funkcji

```rust
fn read_records(
    file_name: String,
) -> Result<(HashMap<NodeIndex, BusStop>, Graph<BusStop, BusRoute>), Box<dyn Error>> {
    let file = File::open(file_name)?;
    let mut rdr = csv::Reader::from_reader(file);
    let mut unique_stops: HashMap<NodeIndex, BusStop> = HashMap::with_capacity(1000);
    let mut stops_cache: HashSet<BusStop> = HashSet::with_capacity(1000);
    let mut graph = Graph::with_capacity(1000, 1000000);

    for (idx, result) in rdr.deserialize::<TransportRecordSerial>().enumerate() {
        let record: TransportRecord = result.unwrap().into();

        let start_stop = record.start_stop;
        let end_stop = record.end_stop;

        let mut start_id: Option<NodeIndex> = None;
        let mut end_id: Option<NodeIndex> = None;

        // add new bus stops to set, get their ids in graph
        if !stops_cache.contains(&start_stop) {
            let _insert = start_id.insert(graph.add_node(start_stop.clone()));
            stops_cache.insert(BusStop {
                id: start_id,
                ..start_stop
            });
```

```rust
        } else {
            start_id = Some(stops_cache.get(&start_stop).unwrap().id.unwrap());
        }

        if !stops_cache.contains(&end_stop) {
            let _insert = end_id.insert(graph.add_node(end_stop.clone()));
            stops_cache.insert(BusStop {
                id: end_id,
                ..end_stop
            });
        } else {
            end_id = Some(stops_cache.get(&end_stop).unwrap().id.unwrap());
        }

        // add edge between stops to graph
        graph.add_edge(
            start_id.expect("Start stop id not found"),
            end_id.expect("End stop id not found"),
            record.route,
        );

        if idx % 100000 == 0 {
            println!("Deserialized {} records", idx);
        }
    }
    stops_cache.into_iter().for_each(|stop| {
        unique_stops.insert(stop.id.unwrap(), stop);
    });
    return Ok((unique_stops, graph));
}
```

2. Algorytm Dijkstry

Algorytm Dijkstry wyszukuje dla wierzchołka początkowego optymalne ścieżki dojścia do innych wierzchołków.

Moje implementacje algorytmów Dijkstry i A* dzielą funkcje pomocnicze

```rust
/// for edges connecting two bus stops, get the best one
fn get_best_route_between<'a>(
    edges: EdgesConnecting<'a, &'a BusRoute, petgraph::Directed>,
    time: u16,
) -> Option<&'a &'a BusRoute> {
    let mut best_edge: Option<EdgeReference<'a, &'a BusRoute>> = None;
    edges.for_each(|edge| match best_edge {
        Some(best) => {
            if time <= edge.weight().departure_time.to_minutes()
                && edge.weight().departure_time < best.weight().departure_time
            {
                best_edge = Some(edge);
            }
        }
        None => {
            if time <= edge.weight().departure_time.to_minutes() {
                best_edge = Some(edge);
            }
        }
    });
    best_edge.map(|edge| edge.weight())
}

type Path = Vec<(BusStop, Option<BusRoute>)>;
```

```rust
// generate a path Vec
fn generate_path(
    path: Vec<NodeIndex>,
    graph: &Graph<BusStop, BusRoute>,
    distances: HashMap<NodeIndex, (u16, Option<BusRoute>)>,
) -> Path {
    let mut path_vec = Vec::new();
    for node in path {
        let stop = graph.node_weight(node).unwrap().clone();
        let route = distances[&node].1.clone();
        path_vec.push((stop, route));
    }
    path_vec
}
```

Moja implementacja algorytmu Dijkstry:

```rust
fn dijkstra(
    stop_a: BusStop,
    stop_b: BusStop,
    beginning_time: MyTime,
    graph: &Graph<BusStop, BusRoute>,
    all_stops: &HashMap<NodeIndex, BusStop>,
) -> Path {
    println!("Dijkstra start");

    let stop_a_id = stop_a.id.unwrap();
    let stop_b_id = stop_b.id.unwrap();

    // separate set of all stops because removing elements from graph shifts indices
    let mut q_set: HashSet<NodeIndex> = all_stops
        .clone()
        .into_iter()
        .map(|(index, _)| index)
        .collect();
    let mut q_set_size = q_set.len();
    graph.node_indices().for_each(|idx| {
        q_set.insert(idx);
    });

    // filter out edges before departure for speedup
    let filtered_graph = graph.filter_map(
        |_, node| Some(node),
        |_, edge| {
            if edge.departure_time < beginning_time {
                None
            } else {
                Some(edge)
            }
        },
    );

    let mut distances: HashMap<NodeIndex, (u16, Option<BusRoute>)> =
        HashMap::with_capacity(q_set_size);
    q_set.clone().into_iter().for_each(|node| {
        distances.insert(node, (u16::MAX, None));
    });

    let mut predecessors: HashMap<NodeIndex, Option<NodeIndex>> =
        HashMap::with_capacity(q_set_size);
    q_set.clone().into_iter().for_each(|node| {
        predecessors.insert(node, None);
```

```
});

let mut current_stop_id: NodeIndex = stop_a_id;

distances.insert(current_stop_id, (beginning_time.to_minutes(), None));

while q_set_size > 0 {
    if q_set_size % 100 == 0 {
        println!("{} nodes in Q remaining", q_set_size);
    }

    // get node from q_set with the lowest distance
    let mut current_lowest_node = q_set.clone().into_iter().next();
    if current_lowest_node.is_some() {
        let mut current_lowest_distance = distances[&current_lowest_node.unwrap()].0;
        for &stop in &q_set {
            let checked_distance = distances[&stop].0;
            if checked_distance < current_lowest_distance {
                current_lowest_distance = checked_distance;
                current_lowest_node = Some(stop);
            }
        }
        current_stop_id = current_lowest_node.unwrap();
    } else {
        println!("No node found in Q! Loop should stop now!");
    }

    q_set.remove(&current_stop_id);
    q_set_size -= 1;

    let current_neighbors =
        filtered_graph.neighbors_directed(current_stop_id, petgraph::Direction::Outgoing);
    let current_distance = distances[&current_stop_id].0;

    // update the distances table with lower distances if found
    // update the predecessors table if lower distance found for neighbor
    current_neighbors.for_each(|neighbor| {
        let neighbor_edges = filtered_graph.edges_connecting(current_stop_id, neighbor);
        let best_bus_route_opt = get_best_route_between(neighbor_edges, current_distance);
        if let Some(best_bus_route) = best_bus_route_opt {
            let neighbor_weight = best_bus_route.arrival_time.to_minutes();
            if neighbor_weight < distances[&neighbor].0 {
                distances.insert(
                    neighbor,
                    (neighbor_weight, Some(best_bus_route.deref().clone())),
                );
                predecessors.insert(neighbor, Some(current_stop_id));
            }
        }
    });
}

let mut path = Vec::new();
let mut current_node = stop_b_id;

// Reconstruct the path from stop_b to stop_a
let mut i = 0;
while let Some(&pred) = predecessors.get(&current_node) {
    i += 1;
    path.push(current_node);
    if pred.unwrap() == stop_a_id {
```

```
            path.push(pred.unwrap());
            break;
        }
        current_node = pred.unwrap();
        if i > 1000 {
            println!("Path longer than 1000 stops, breaking");
            break;
        }
    }

    path.reverse();

    // return path in a nice format
    return generate_path(path, graph, distances);
}
```

Działanie algorytmu dijkstra dla trasy Piastowska -> FAT

```
Stop a found: BusStop { id: Some(NodeIndex(123)), name: "Piastowska", coords: Coords { lat: 51.116207, lon: 17.0
Stop b found: BusStop { id: Some(NodeIndex(31)), name: "FAT", coords: Coords { lat: 51.094128, lon: 16.978354 }
Euclidean distance between a and b: 0
Dijkstra start
Size of Q set: 939
900 nodes in Q remaining
800 nodes in Q remaining
700 nodes in Q remaining
600 nodes in Q remaining
500 nodes in Q remaining
400 nodes in Q remaining
300 nodes in Q remaining
200 nodes in Q remaining
100 nodes in Q remaining
It took 5767ms

Piastowska -> [MPK Tramwaje] [19] PL. GRUNWALDZKI (12:00-12:04) -> [12] most Grunwaldzki (12:05-12:06) -> Urząd
Full route time: 32 minutes
Route line changes: 5
```

3. Algorytm A*

Algorytm A* jest algorytmem Dijkstry zoptymalizowanym pod szukanie ścieżki do konkretnego celu. Odzwierciedla to moja implementacja, która jest analogiczna do implementacji Dijkstry. Za pomocą parametru `limit_line_changes` można wybrać kryterium optymalizacyjne jako minimalizację liczby zmian linii, domyślnie jest to tak jak w Dijkstra minimalizacja czasu.

Jako funkcję estymacji kosztu wybrałem odległość Euklidesową:

```
/// euclidean distance between two sets of coordinates
fn euclidean_distance(a: &Coords, b: &Coords) -> f32 {
    let dx = a.lat - b.lat;
    let dy = a.lon - b.lon;
    (dx * dx + dy * dy).sqrt()
}
```

Implementacja A*:

```
fn astar(
    stop_a: BusStop,
    stop_b: BusStop,
    beginning_time: MyTime,
    graph: &Graph<BusStop, BusRoute>,
    all_stops: &HashMap<NodeIndex, BusStop>,
    limit_line_changes: bool,
) -> Path {
```

```rust
println!("Astar start");
if limit_line_changes {
    println!("Limiting line changes");
}

let stop_a_id = stop_a.id.unwrap();
let stop_b_id = stop_b.id.unwrap();

// separate set of all stops because removing elements from graph shifts indices
let mut q_set: HashSet<NodeIndex> = all_stops
    .clone()
    .into_iter()
    .map(|(index, _)| index)
    .collect();
let mut q_set_size = q_set.len();
graph.node_indices().for_each(|idx| {
    q_set.insert(idx);
});

// filter out edges before departure for speedup
let filtered_graph = graph.filter_map(
    |_, node| Some(node),
    |_, edge| {
        if edge.departure_time < beginning_time {
            None
        } else {
            Some(edge)
        }
    },
);
let mut distances: HashMap<NodeIndex, (u16, Option<BusRoute>)> =
    HashMap::with_capacity(q_set_size);
q_set.clone().into_iter().for_each(|node| {
    distances.insert(node, (u16::MAX, None));
});

let mut predecessors: HashMap<NodeIndex, Option<NodeIndex>> =
    HashMap::with_capacity(q_set_size);
q_set.clone().into_iter().for_each(|node| {
    predecessors.insert(node, None);
});

let mut current_stop_id: NodeIndex = stop_a_id;

println!("Size of Q set: {}", q_set_size);

distances.insert(current_stop_id, (beginning_time.to_minutes(), None));

while q_set_size > 0 {
    if q_set_size % 100 == 0 {
        println!("{} nodes in Q remaining", q_set_size);
    }

    if current_stop_id == stop_b_id {
        println!("Found stop b in Q set, breaking loop");
        break;
    }

    // get node from q_set with the lowest distance
    let mut current_lowest_node = q_set.clone().into_iter().next();
    if current_lowest_node.is_some() {
```

```
            let mut current_lowest_distance = distances[&current_lowest_node.unwrap()].0;
            for &stop in &q_set {
                let checked_distance = distances[&stop].0;
                if checked_distance < current_lowest_distance {
                    current_lowest_distance = checked_distance;
                    current_lowest_node = Some(stop);
                }
            }
            current_stop_id = current_lowest_node.unwrap();
        } else {
            println!("No node found in Q! Loop should stop now!");
        }

        q_set.remove(&current_stop_id);
        q_set_size -= 1;

        let current_neighbors =
            filtered_graph.neighbors_directed(current_stop_id, petgraph::Direction::Outgoing);
        let current_distance = distances[&current_stop_id].0;

        // update the distances table with lower distances if found
        // update the predecessors table if lower distance found for neighbor
        current_neighbors.for_each(|neighbor| {
            let neighbor_edges = filtered_graph.edges_connecting(current_stop_id, neighbor);
            let best_bus_route_opt = get_best_route_between(neighbor_edges, current_distance);
            if let Some(best_bus_route) = best_bus_route_opt {
                // Calculate the weight of the neighbor including the distance to the final stop
                // Add 30 if the criterion is to limit line changes and the line changes
                let neighbor_weight = {
                    let neighbor_stop = all_stops[&neighbor].clone();
                    let mut weight = best_bus_route.arrival_time.to_minutes()
                        + (12. * euclidean_distance(&neighbor_stop.coords, &stop_b.coords)) as u16;
                    if limit_line_changes {
                        let current_stop_route = distances[&current_stop_id].1.clone();
                        if current_stop_route.is_some()
                            && best_bus_route.line != current_stop_route.unwrap().line
                        {
                            weight += 30;
                        }
                    }
                    weight
                };

                if neighbor_weight < distances[&neighbor].0 {
                    distances.insert(
                        neighbor,
                        (neighbor_weight, Some(best_bus_route.deref().clone())),
                    );
                    predecessors.insert(neighbor, Some(current_stop_id));
                }
            }
        });
    }

    let mut path = Vec::new();
    let mut current_node = stop_b_id;

    // Reconstruct the path from stop_b to stop_a
    let mut i = 0;
    while let Some(&pred) = predecessors.get(&current_node) {
        i += 1;
```

```
        path.push(current_node);
        if pred.unwrap() == stop_a_id {
            path.push(pred.unwrap());
            break;
        }
        current_node = pred.unwrap();
        if i > 1000 {
            println!("Path longer than 1000 stops, breaking");
            break;
        }
    }

    path.reverse();

    //return the path in a nice format
    return generate_path(path, graph, distances);
}
```

Działanie algorytmu A* dla trasy Piastowska -> FAT o godzinie 12:00 i minimalizacji czasu:

```
Stop a found: BusStop { id: Some(NodeIndex(123)), name: "Piastowska", coords: Coords { lat: 51.116207, lon: 17.0
Stop b found: BusStop { id: Some(NodeIndex(31)), name: "FAT", coords: Coords { lat: 51.094128, lon: 16.978354 }
Euclidean distance between a and b: 0.08545551
Astar start
Size of Q set: 939
900 nodes in Q remaining
800 nodes in Q remaining
Found stop b in Q set, breaking loop
It took 4358ms

Piastowska -> [MPK Tramwaje] [19] PL. GRUNWALDZKI (12:00-12:04) -> [12] most Grunwaldzki (12:05-12:06) -> [MPK A
Full route time: 32 minutes
Route line changes: 4
```

Działanie algorytmu A* dla trasy Piastowska -> FAT o godzinie 12:00 i minimalizacji przesiadek:

```
Stop a found: BusStop { id: Some(NodeIndex(123)), name: "Piastowska", coords: Coords { lat: 51.116207, lon: 17.0
Stop b found: BusStop { id: Some(NodeIndex(31)), name: "FAT", coords: Coords { lat: 51.094128, lon: 16.978354 }
Euclidean distance between a and b: 0.08545551
Astar start
Limiting line changes
Size of Q set: 939
900 nodes in Q remaining
800 nodes in Q remaining
700 nodes in Q remaining
Found stop b in Q set, breaking loop
It took 4458ms

Piastowska -> [MPK Tramwaje] [19] Górnickiego (12:04-12:06) -> Ogród Botaniczny (12:06-12:08) -> pl. Bema (12:08
Full route time: 101 minutes
Route line changes: 2
```

4 Przeszukiwanie Tabu

Implementacja algorytmu przeszukiwania Tabu z opcjonalnym maksymalnym rozmiarem listy (max_tabu_size)

```
fn tabu_search(
    graph: &Graph<BusStop, BusRoute>,
    all_stops: &HashMap<NodeIndex, BusStop>,
    start_stop: BusStop,
    stations_list: Vec<BusStop>,
    time_at_start: MyTime,
    limit_line_changes: bool,
    max_iterations: usize,
```

```
    max_tabu_size: Option<u32>,
) -> Path {
    struct Solution {
        path: Vec<BusStop>,
        cost: u16,
        full_path: Path,
    }

    impl Solution {
        fn new(path: Vec<BusStop>, cost: u16, full_path: Option<Path>) -> Self {
            Solution {
                path,
                cost,
                full_path: full_path.unwrap_or_default(),
            }
        }

        fn clone(&self) -> Solution {
            Solution {
                path: self.path.clone(),
                cost: self.cost,
                full_path: self.full_path.clone(),
            }
        }
    }

    // calculate the number of changes in a path.
    fn get_number_of_changes(path: &Path) -> u16 {
        let mut number_of_changes = 0;
        let mut curr_line: Option<String> = None;
        path.iter().for_each(|(_, route_opt)| {
            if let Some(route) = route_opt {
                if let Some(line) = curr_line.clone() {
                    if line != route.line {
                        number_of_changes += 1;
                    }
                    curr_line = Some(route.line.clone());
                } else {
                    curr_line = Some(route.line.clone());
                }
            }
        });
        number_of_changes
    }

    // generate neighbors for a given solution.
    fn generate_neighbour(solution: &Solution, start_stop: BusStop) -> Vec<Solution> {
        let mut neighbours = Vec::new();
        for i in 0..solution.path.len() - 1 {
            for j in i + 1..solution.path.len() - 2 {
                let mut new_path = solution.path.clone();
                new_path.remove(0);
                new_path.pop();
                new_path.swap(i, j);
                new_path.insert(0, start_stop.clone());
                new_path.push(start_stop.clone());
                neighbours.push(Solution::new(new_path, u16::MAX, None));
            }
        }
        neighbours
    }
```

```rust
// calculate the cost for a solution.
fn calculate_cost_for_solution(
    solution: &mut Solution,
    graph: &Graph<BusStop, BusRoute>,
    all_stops: &HashMap<NodeIndex, BusStop>,
    time_at_start: MyTime,
    limit_line_changes: bool,
) {
    solution.full_path.clear();

    let mut current_time = time_at_start;

    for i in 0..solution.path.len() - 1 {
        let stop_a = solution.path[i].clone();
        let stop_b = solution.path[i + 1].clone();

        let astar_path = astar(
            stop_a,
            stop_b,
            current_time,
            graph,
            all_stops,
            limit_line_changes,
        );
        current_time = astar_path.last().cloned().unwrap().1.unwrap().arrival_time;
        let sub_path_to_station = if i != 0 {
            &astar_path[1..]
        } else {
            &astar_path
        };

        solution.full_path.extend_from_slice(sub_path_to_station);
    }
    // Set the total cost of the solution
    solution.cost = current_time.to_minutes();
    if limit_line_changes {
        solution.cost += 30 * get_number_of_changes(&solution.full_path);
    }
}

let ran_gen = &mut rand::thread_rng();
let mut random_path = stations_list.clone();
random_path.shuffle(ran_gen);
random_path.insert(0, start_stop.clone());
random_path.push(start_stop.clone());
let mut best_solution = Solution::new(random_path, u16::MAX, None);
calculate_cost_for_solution(
    &mut best_solution,
    graph,
    all_stops,
    time_at_start,
    limit_line_changes,
);
let mut tabu_list: VecDeque<Vec<BusStop>> = VecDeque::new();
tabu_list.push_back(best_solution.path.clone());

// Main loop of the algorithm.
for _ in 0..max_iterations {
    let neighbours = generate_neighbour(&best_solution, start_stop.clone());
    let mut best_neighbour_cost = u16::MAX;
```

```
        let mut best_neighbour = None;

        for neighbour in neighbours {
            let neighbour_path = neighbour.path.clone();
            if !tabu_list.contains(&neighbour_path) {
                let mut neighbour = neighbour;
                calculate_cost_for_solution(
                    &mut neighbour,
                    graph,
                    all_stops,
                    time_at_start,
                    limit_line_changes,
                );
                tabu_list.push_back(neighbour_path);

                if neighbour.cost < best_neighbour_cost {
                    best_neighbour = Some(neighbour.clone());
                    best_neighbour_cost = neighbour.cost;
                }
            }
        }

        if let Some(neighbour) = best_neighbour {
            if neighbour.cost < best_solution.cost {
                best_solution = neighbour;
            }
        }
        if let Some(max_tabu_size) = max_tabu_size {
            if tabu_list.len() > (max_tabu_size as usize) + stations_list.len() {
                tabu_list.pop_front();
            }
        }
    }
    println!("Cost: {}", best_solution.cost);
    best_solution.full_path
}
```

dla przystanku początkowego Młodych Techników i przystanków pomiędzy Dubois, Plac Grunwaldzki, Rondo, Wrocławski Park Przemysłowy

```
cargo run --release -- tabu "młodych techników"
12:00 t "dubois, pl. grunwaldzki, rondo, wrocławski park przemysłowy"
[usunięto część linii]
Astar start
Size of Q set: 939
900 nodes in Q remaining
Found stop b in Q set, breaking loop
Astar start
Size of Q set: 939
900 nodes in Q remaining
Found stop b in Q set, breaking loop
Astar start
Size of Q set: 939
900 nodes in Q remaining
Found stop b in Q set, breaking loop
Cost: 789
It took 125741ms

Młodych Techników -> [MPK Tramwaje] [10] PL. JANA PAWŁA II (12:00-12:02)
-> Rynek (12:02-12:05) -> Zamkowa (12:05-12:06) ->
Świdnicka (12:06-12:08) -> GALERIA DOMINIKAŃSKA (12:08-12:10)
-> [MPK Autobusy] [D] Urząd Wojewódzki (Impart) (12:10-12:13) ->
```

```
most Grunwaldzki (12:13-12:14) -> PL. GRUNWALDZKI (12:14-12:16)
 -> [MPK Tramwaje] [19] Piastowska (12:16-12:18) ->
Górnickiego (12:18-12:20) -> Ogród Botaniczny (12:20-12:22)
-> pl. Bema (12:22-12:24) -> Dubois (12:24-12:26) ->
Pomorska (12:26-12:28) -> Kępa Mieszczańska (12:28-12:31)
-> [14] PL. JANA PAWŁA II (12:31-12:33) -> pl. Orląt Lwowskich (12:33-12:35)
-> [MPK Autobusy] [106] Renoma (12:35-12:37) ->
 [MPK Tramwaje] [7] Arkady (Capitol) (12:38-12:40) -> [18] Zaolziańska (12:40-12:42)
-> Wielka (12:42-12:43) -> Rondo (12:43-12:44) ->
[MPK Autobusy] [D] Arkady (Capitol) (12:44-12:50) ->
 [MPK Tramwaje] [6] Renoma (12:50-12:52)
-> [MPK Autobusy] [148] pl. Orląt Lwowskich (12:52-12:55) ->
 Dworzec Świebodzki (12:55-12:57) -> Smolecka (12:57-12:59) ->
Śrubowa (12:59-13:00) -> Wrocławski Park Przemysłowy (13:00-13:01) ->
 [149] Śrubowa (13:02-13:04) ->
[142] pl. Strzegomski (Muzeum Współczesne) (13:05-13:08) ->
 Młodych Techników (13:08-13:09)
Full route time: 69 minutes
Route line changes: 11
```

Algorytmy działają dosyć powoli, dlatego stworzyłem flamegraph aby zobaczyć, która funkcja jest najbardziej kosztowna. Okazało się, że znajdywanie krawędzi łączących 2 wierzchołki (graph.edges_connecting) zajmuje zdecydowaną większość czasu trwania programu. Zmiana biblioteki do grafów, lub napisanie własnej wymagałoby restrukturyzacji znaczącej części programu. Zamiast tego, postanowiłem zastosować bibliotekę Rayon do wprowadzenia wielowątkowości do funkcji tabu_search.

Zmieniony kod w tabu_search:

```rust
let tabu_list: Mutex<VecDeque<Vec<BusStop>>> = Mutex::new(VecDeque::new());
tabu_list.lock().unwrap().push_back(best_solution.path.clone());

// Main loop of the algorithm.
for _ in 0..max_iterations {
    let neighbours = generate_neighbour(&best_solution, start_stop.clone());
    let best_neighbour_cost = Mutex::new(u16::MAX);
    let best_neighbour = Mutex::new(None);

    neighbours.into_par_iter().for_each(|neighbour|{
        let neighbour_path = neighbour.path.clone();
        if !tabu_list.lock().unwrap().contains(&neighbour_path) {
            let mut neighbour = neighbour;
            calculate_cost_for_solution(
                &mut neighbour,
                graph,
                all_stops,
                time_at_start,
                limit_line_changes,
            );
            tabu_list.lock().unwrap().push_back(neighbour_path);

            if best_neighbour_cost.lock().unwrap().gt(&neighbour.cost) {
                let _ = best_neighbour.lock().unwrap().insert(neighbour.clone());
                best_neighbour_cost.lock().unwrap().clone_from(&neighbour.cost);
            }
        }
    });
    let locked_best_neighbour = best_neighbour.lock().unwrap();
    if  locked_best_neighbour.is_some() {
        if <std::option::Option<Solution> as Clone>::clone(&locked_best_neighbour).unwrap().cost <
         ↪  best_solution.cost {
            best_solution = <std::option::Option<Solution> as
↪  Clone>::clone(&locked_best_neighbour).unwrap();
        }
    }
```

```
    if let Some(max_tabu_size) = max_tabu_size {
        if tabu_list.lock().unwrap().len() > (max_tabu_size as usize) + stations_list.len() {
            tabu_list.lock().unwrap().pop_front();
        }
    }
}
```

Jak widać, wielowątkowy iterator zastosowałem tylko na wewnętrznej pętli sprawdzającej sąsiednie rozwiązania. Główną modyfikacją w kodzie było opakowanie niektórych zmiennych w mutexy.

Ten sam zestaw argumentów po wprowadzeniu wielowątkowości:

```
cargo run --release -- tabu "młodych techników"
12:00 t "dubois, pl. grunwaldzki, rondo, wrocławski park przemysłowy"
[usunięto część linii]
Astar start
Size of Q set: 939
Found stop b in Q set, breaking loop
900 nodes in Q remaining
Found stop b in Q set, breaking loop
Cost: 786
Tabu finished, took 36443ms

Młodych Techników -> [MPK Tramwaje] [22] pl. Strzegomski (Muzeum Współczesne) (12:00-12:01) ->
[13] Dolmed (12:06-12:08) -> Śrubowa (12:08-12:09) -> Wrocławski Park Przemysłowy (12:09-12:10) ->
[MPK Autobusy] [132] Śrubowa (12:12-12:13) -> [MPK Tramwaje] [23] Smolecka (12:14-12:15) ->
Dworzec Świebodzki (12:15-12:17) -> pl. Orląt Lwowskich (12:17-12:19) ->
[MPK Autobusy] [127] Tęczowa (12:20-12:21) -> Grabiszyńska (12:21-12:23) -> Zaporoska (12:23-12:25) ->
[126] Rondo (12:25-12:27) -> [MPK Tramwaje] [17] Wielka (12:27-12:29) ->
[20] Zaolziańska (12:29-12:30) -> Arkady (Capitol) (12:30-12:33) ->
[MPK Autobusy] [D] GALERIA DOMINIKAŃSKA (12:35-12:40) -> Urząd Wojewódzki (Impart) (12:40-12:43) ->
[MPK Tramwaje] [4] most Grunwaldzki (12:43-12:45) -> PL. GRUNWALDZKI (12:45-12:47) ->
[19] Piastowska (12:47-12:49) -> Górnickiego (12:49-12:51) -> Ogród Botaniczny (12:51-12:53) ->
pl. Bema (12:53-12:55) -> Dubois (12:55-12:57) -> Pomorska (12:57-12:59) ->
[MPK Autobusy] [142] Mosty Pomorskie (12:59-13:00) -> Rynek (13:00-13:02) ->
PL. JANA PAWŁA II (13:02-13:04) -> Młodych Techników (13:04-13:06)
Full route time: 66 minutes
Route line changes: 11
```

Na 12-wątkowym procesorze osiągnięto ~4-krotne przyspieszenie wprowadzając niewiele zmian w kodzie źródłowym.