Sprawozdanie - Projektowanie efektywnych algorytmów

Maksim Birszel nr indeksu 241353

12 stycznia 2020

Zadanie 2 - Implementacja algorytmu Tabu Search dla Problemu Komiwojażera

Prowadzący:
Dr. Inż
Zbigniew Buchalski

Termin zajęć:
Wtorek 9:15

1 Wstęp teoretyczny

Problem komiwojażera (ang. travelling salesman problem, TSP) – zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.
Nazwa pochodzi od typowej ilustracji problemu, przedstawiającej go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić, oraz odległość / cena podróży / czas podróży pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej / najtańszej / najszybszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w określonym punkcie. Jest on zaliczany to klasy problemów NP-trudnych.
Problem dzielimy na symetryczny oraz asymetryczny.
Symetryczny - odległości z miasta A do B są równe odległością z B do A
Asymetryczny - odległości z A do B różnią się od odległości z B do A

2 Algorytm Tabu Search

Przeszukiwanie tabu (Tabu search, TS) – metaheurystyka (algorytm) stosowana do rozwiązywania problemów optymalizacyjnych. Wykorzystywana do otrzymywania rozwiązań optymalnych lub niewiele różniących się od niego dla problemów z różnych dziedzin (np. planowanie, planowanie zadań). Za twórcę uznawany jest Fred W. Glover, który w latach 70-tych opublikował kilka prac związanych z tym sposobem poszukiwania rozwiązania.

Podstawową ideą algorytmu jest przeszukiwanie przestrzeni, stworzonej ze wszystkich możliwych rozwiązań, za pomocą sekwencji ruchów. W sekwencji ruchów istnieją ruchy niedozwolone, ruchy tabu. Algorytm unika oscylacji wokół optimum lokalnego dzięki przechowywaniu informacji o sprawdzonych już rozwiązaniach w postaci listy tabu (TL).

3 Podstawowe pojęcia

Heurystyka - technika znajdująca dobre rozwiązanie problemu (np. optymalizacji kombinatorycznej) przy rozsądnych (akceptowalnych z punktu widzenia celu) nakładach obliczeniowych, bez gwarancji osiągalności i optymalności rozwiązania oraz bez określenia jakości rozwiązania (odległości od rozwiązania optymalnego).
Metaheurystyka - ogólna metoda rozwiązywania dowolnych problemów obliczeniowych, które można opisać za pomocą zdefiniowanych przez tę metodę

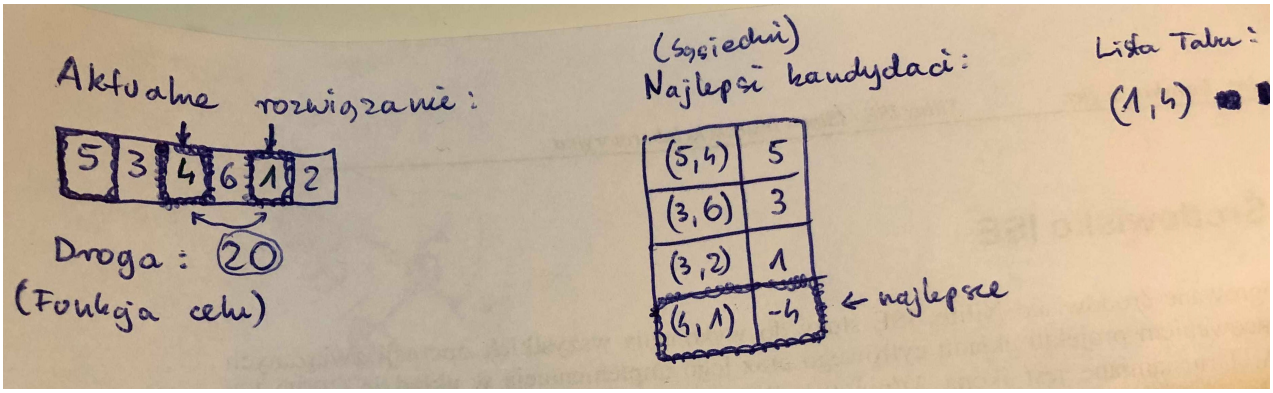
pojęć. Metaheurystyki definiują sposób tworzenia algorytmów do rozwiązywania konkretnych problemów.
Sąsiedztwo rozwiązania - nazywamy tak zbiór permutacji otrzymanych poprzez wykonanie zaburzeń (np. ruchów typu swap(i,j) – zamiany miejscami elementu i z elementem j).
Lista Tabu - zbiór wykonanych zaburzeń (np. ruchów typu swap(i,j)), jest to kolejka typu LIFO, czyli Last In First Out - tzn. w momencie dodania nowego elementu, ostatni z nich jest usuwany. Może mieć dowolną ustaloną z góry długość.
Kryterium aspiracji - czasem ruchy należące do listy Tabu mogą okazać się dużo lepsze od dostępnych, przez co pominięcie ich może prowadzić do pominięcia rozwiązania optymalnego. W takim wypadku jest dozwolone (czasami) wykorzystanie ruchu z listy Tabu. Kryterium aspiracji określa warunek, podczas którego jest to wykonywane.
Strategia dywersyfikacji - procedura przeszukiwania różnych obszarów lokalnych.

4 Pseudokod algorytmu Tabu Search

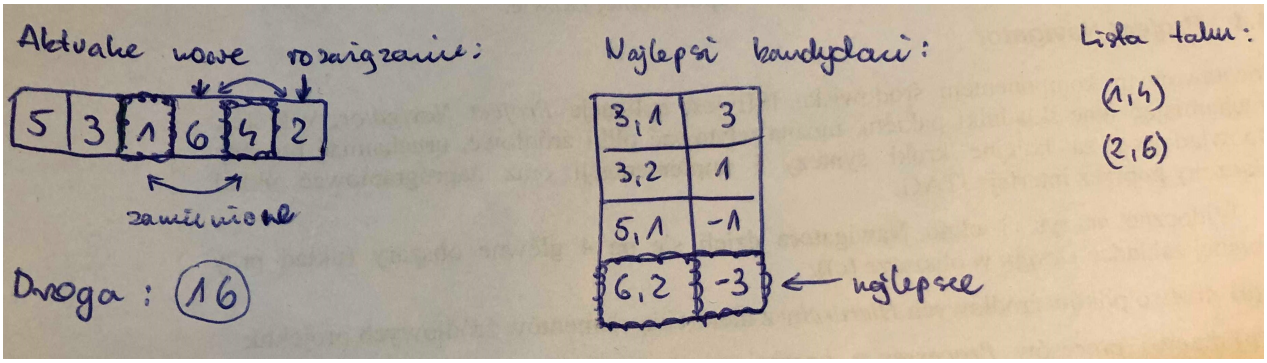
```
sBest ← s0
bestCandidate ← s0
tabuList ← []
tabuList.push(s0)
while (not stoppingCondition())
    sNeighborhood ← getNeighbors(bestCandidate)
    for (sCandidate in sNeighborhood)
        if ( (not tabuList.contains(sCandidate)) and (fitness(sCandidate) > fitness(bes
            bestCandidate ← sCandidate
        end
    end
    if (fitness(bestCandidate) > fitness(sBest))
        sBest ← bestCandidate
    end
    tabuList.push(bestCandidate)
    if (tabuList.size > maxTabuSize)
        tabuList.removeFirst()
    end
end
return sBest
```

5 Przykład praktyczny

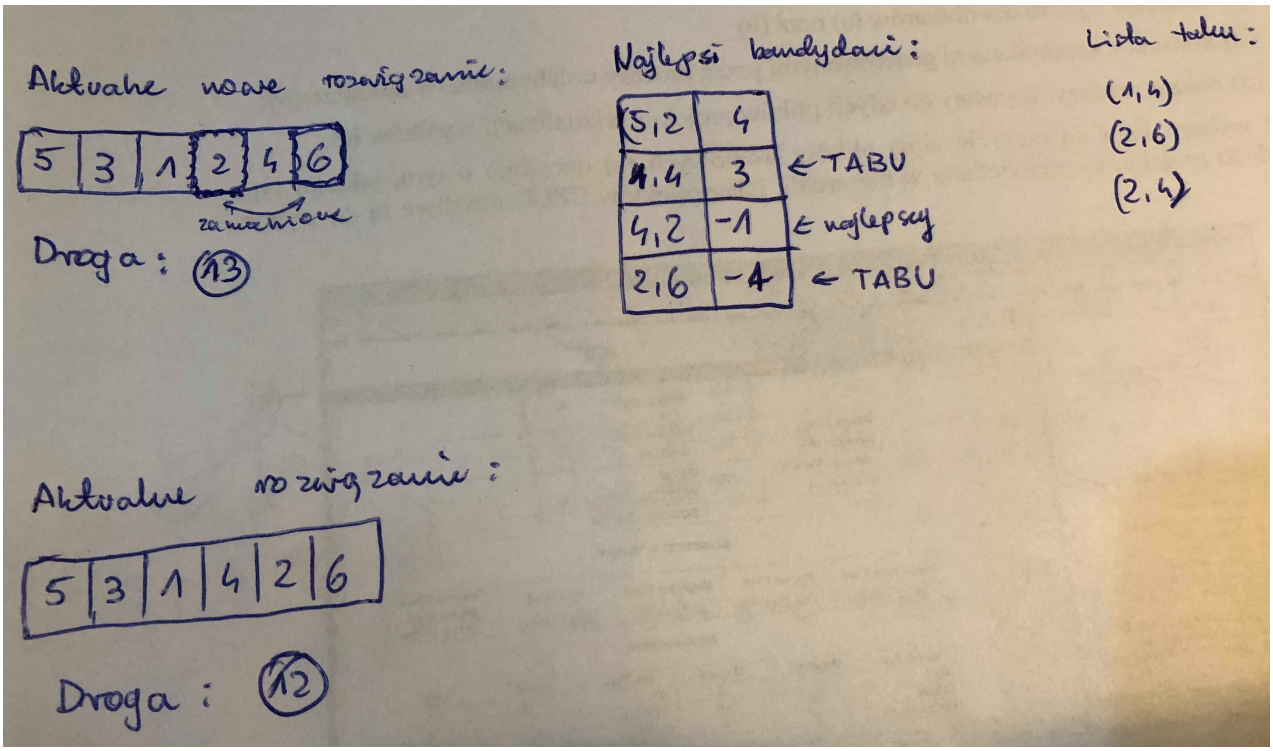
Zdjęcie przedstawia aktualne losowe rozwiązanie dla 6 elementowej instancji problemu. Obok jest wypisana 4 elementowa lista najlepszych kandydatów (sąsiadów). W naszym wypadku najmniejsza wartość oznacza najlepszy wynik, stąd zamiana elementów (4,1) daje nam optymalny wynik. Następnie element przeciwny do tego zostaje dodany do listu ruchów zakazanych - listy Tabu:



W kolejnym kroku generujemy nowe (lepsze) rozwiązanie złożone z rozwiązania poprzedniego, ale z zamienionymi elementami 4 oraz 1 oraz zaktualizowaną funkcją celu (w naszym wypadku drogą). Obok wypisujemy najlepszych kandydatów obecnego rozwiązania z wybranym najlepszym z nich, którego przeciwieństwo dodajemy do listy Tabu:



W kolejnym kroku znów zamieniamy wybrane wcześniej elementy (miasta) i szukamy najlepszego kandydata do ponownej zamiany. Są nim elementy (2,6). Niestety widzimy, że znajdują się one na liście Tabu, więc musimy z nich zrezygnować na rzecz następnego w kolei najlepszego kandydata:



6 Implementacja algorytmu

Funkcja obliczająca koszt aktualnej ścieżki:

```
int cost = 0;

for (int i = 0; i < N - 1; i++)
cost += matrix_distance[permutation.at(i)][permutation.at(i + 1)];

// Dodanie kosztu wierzchołka 0
cost += matrix_distance[permutation.at(N - 1)][0];

return cost;
```

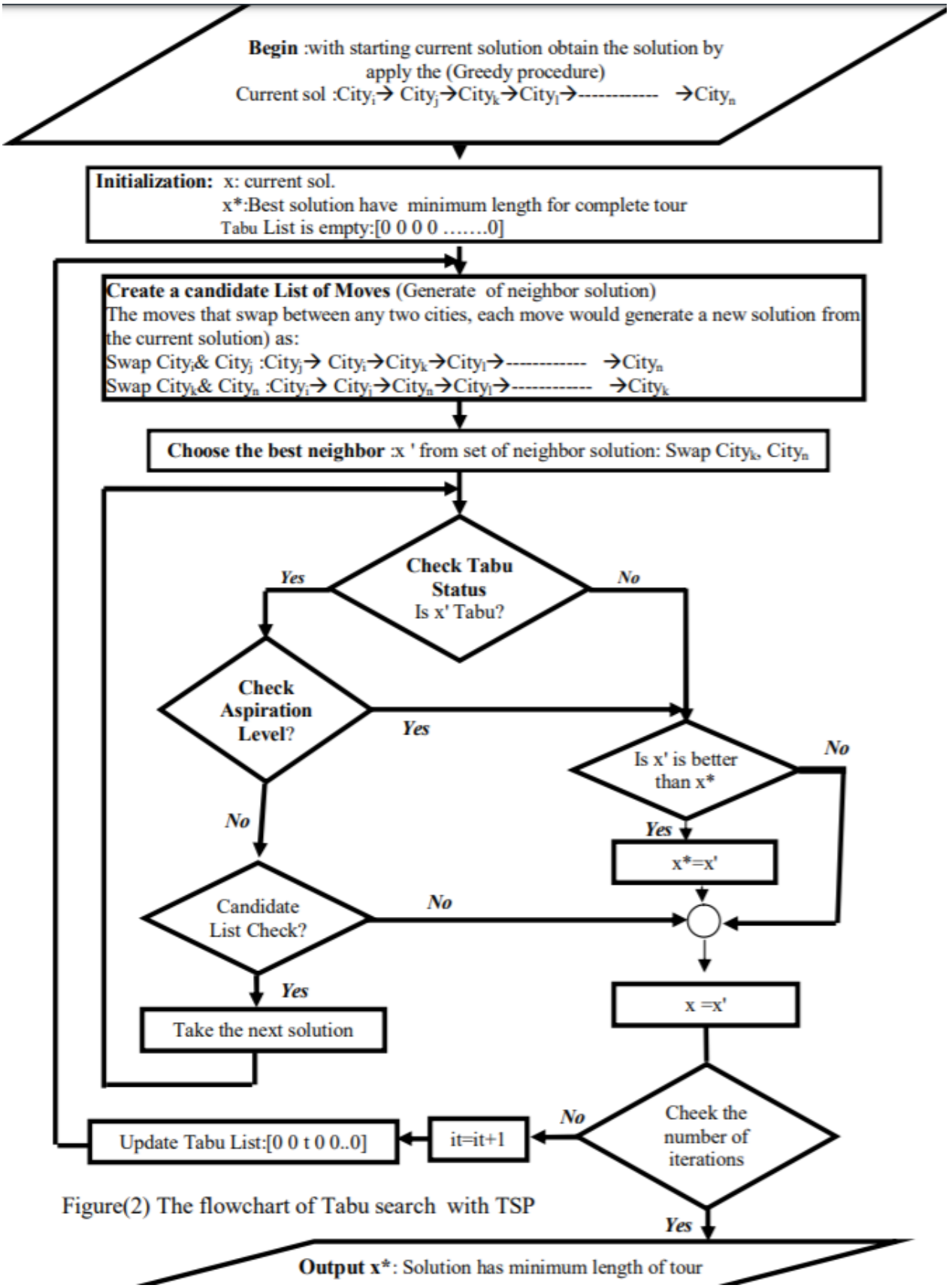
Funkcja losująca dwa miasta i zamieniająca je miejscami:

```
// Generujemy dwa miasta
int random_city = rand() % (N - 1) + 1;
int another_random_city = rand() % (N - 1) + 1;

// Muszą być różne od siebie
while (random_city == another_random_city)
another_random_city = rand() % (N - 1) + 1;
```

```
// Zamieniamy miasta
swap(route[random_city], route[another_random_city]);
```

Funkcja generująca najlepszego sąsiada:

```
// Przypisanie byle jakiego początkowego sąsiada
vector<int> best_neighbour = permutation;
// Aktualne (i później końcowe) miasta sąsiada, które zostały zeswapowane
int best_city_x, best_city_y;
int best_cost_neighbours = INT_MAX;
int cost_current;

// Zaczynamy od 1, bo zero musi być zawsze na początku
for (int i = 1; i < N; i++)
{
for (int j = 2; j < N; j++)
{
// Sąsiadem nie jest to samo rozwiązanie, a stanie się tak, gdy i == j
// if (i == j) continue;

// Wykonaj tylko dla tych krawędzi, które nie są na zakazanej liście
if (tabu_list[i][j] == 0)
{
vector<int> new_perm = permutation;
swap(new_perm[i], new_perm[j]);
cost_current = Calculate_cost(new_perm);

if (cost_current < best_cost_neighbours)
{
best_cost_neighbours = cost_current;
best_neighbour = new_perm;
best_city_x = i;
best_city_y = j;
}

}
}
}

// Ustawienie kadencji
tabu_list[best_city_x][best_city_y] = lifetime;

return best_neighbour;
```



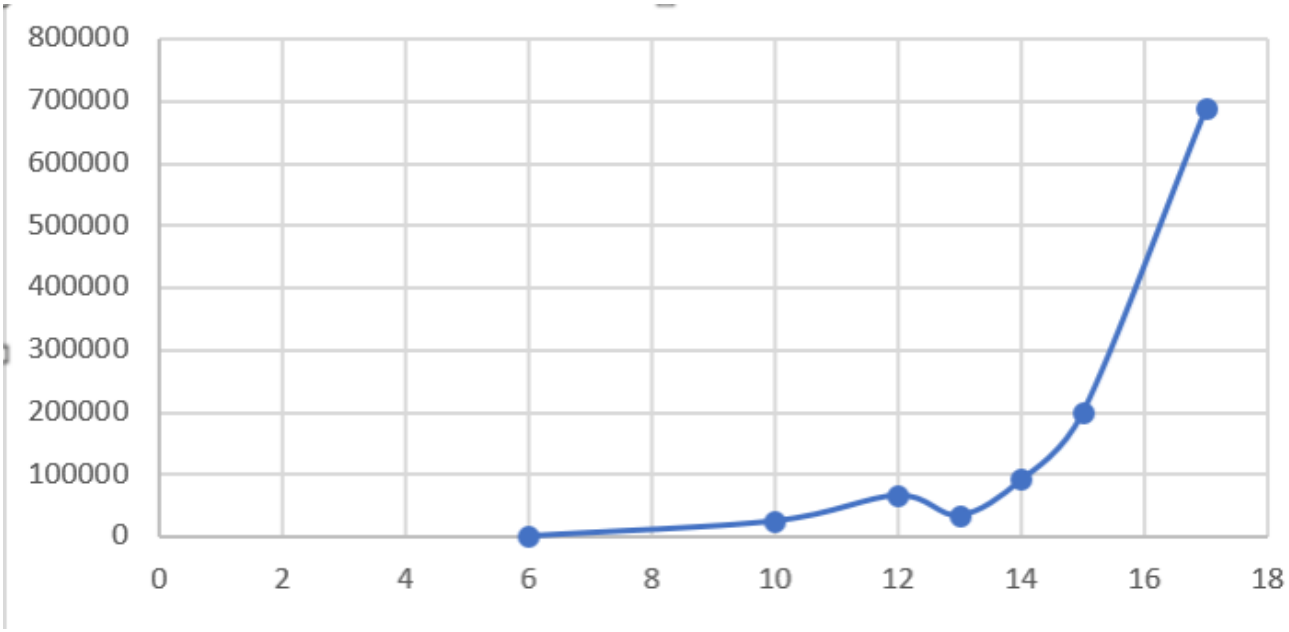
Figure(2) The flowchart of Tabu search with TSP

7 Plan eksperymentu

Dane oraz rozmiary testowanych danych pochodziły ze strony Pana Jarosława Mierzwy. Podobnie użyta metoda pomiaru czasu - funkcja QueryPerformanceCounter.

8 Wyniki eksperymentu

Czas obliczenia najkrótszej ścieżki w sekundach w zależności od ilości miast dla algorytmu Tabu Search: (mikrosekundy/ilość miast)



9 Wnioski

Z racji tego, że algorytm Tabu search jest algorytmem metaheurystycznym nie daje nam gwarancji uzyskania optymalnego rozwiązania. Każdorazowe wykonanie go może dać różne rozwiązanie od poprzednich, jednak wszystkie podczas testów oscylowały wokół tego minimalnego. Z tego powodu możemy uznać ten algorytm za bardzo efektywny. Jego czas rośnie przy większych instancjach z tego powodu, że musimy przejrzeć większą ilość miast, jednak nie komplikuje to problemu dużo bardziej. Dużo też zależy od zastosowanych parametrów dywersyfikacji oraz aspiracji.