Problem komiwojażera

Opis problemu

Dane jest n miast. Komiwojażer ma odwiedzić każde miasto dokładnie jeden raz wracając do punktu początkowego. Znamy koszty przejazdu pomiędzy każdą parą miast. Zaplanuj drogę komiwojażera tak, aby każde miasto zostało odwiedzone dokładnie jeden raz i całkowity koszt przejazdu był jak najmniejszy.

Specyfikacja problemu

Założenia: Graf G=(V,E), gdzie elementy zbioru V są wierzchołkami tego grafu, a E to zbiór krawędzi reprezentowanych jako pary wierzchołków pomiędzy którymi istnieje krawędź i niech |V|=n.

Macierz wag krawędzi.

Problem: Znaleźć w grafie G cykl Hamiltona o minimalnej sumie wag.

Problem komiwojażera jest NP-trudny, oznacza to, że nie jest znany wielomianowy algorytm rozwiązujący ten problem.

Naiwne rozwiązanie

1. Weź dowolny wierzchołek jako punkt startowy i końcowy.
2. Stwórz wszystkie (n-1)! permutacji miast.
3. Oblicz koszt każdej permutacji i zapamiętaj minimum z tych permutacji.
4. Zwróć permutację zawierającą najmniejszy koszt dotarcia do celu.

Naiwne rozwiązanie

1. Weź dowolny wierzchołek jako punkt startowy i końcowy.
2. Stwórz wszystkie (n-1)! permutacji miast.
3. Oblicz koszt każdej permutacji i zapamiętaj minimum z tych permutacji.
4. Zwróć permutację zawierającą najmniejszy koszt dotarcia do celu.

Złożoność czasowa: O(n!)

Dlaczego to rozwiązanie nas nie zadowala?

Załóżmy, że nasz komputer potrafi wykonywać (10^21) operacji na sekundę.

n - liczba miast => (n-1)! możliwych cykli

Porównajmy czas wykonywania poszczególnych operacji z czasem np.: 100*n^15

n	100*n^15 kroków	n! kroków
n = 20	3.27 seconds	2.4 milliseconds
n = 25	1.5 minutes	4.2 hours
n = 30	24 minutes	8.400 years
n = 35	4 hours	326 billion years

Warianty rozwiązania i ograniczenia problemu

Znajdujemy rozwiązanie bliskie optymalnemu. Algorytmy znajdujące rozwiązania bliskie optymalnemu można podzielić na dwie grupy:

Algorytmy włączania - zaczynają się od wyboru 1 dowolnego wierzchołka dodając w kolejnych iteracjach po 1 wierzchołku, za każdym razem rozszerzając cykl.

Algorytmy lokalne - Na wejściu dostają dowolny cykl Hamiltona i w każdej iteracji starają się go ulepszyć. Glownie wariacje algorytmu r-opt Lina-Kernighana.

W prezentacji omówione zostaną algorytmy lokalne.

Dla uproszczenia algorytmów zakładamy najgorszy przypadek: że graf jest pełny i skierowany.

Znane algorytmy lokalne

- Algorytm 2-optymalny.
- Algorytm 3-optymalny.

Algorytm 2-opt

Niezbędne definicje:

waga(T) - suma wszystkich krawędzi grafu T.

2 cykle Hamiltona nazywany 2-sąsiadującymi, jeżeli jeden można uzyskać z drugiego poprzez usunięcie 2 krawędzi i dodanie 2 krawędzi.

Cykl Hamiltona T nazywamy 2-optymalnym, jeżeli nie istnieje 2-sąsiadujący do T cykl T’, w którym waga(T’) < waga(T)

Opis algorytmu

1. Generujemy dowolny cykl Hamiltona TSP[n]
2. Przeszukaj wszystkie krawędzie w TSP i znajdź najlepszą parę krawędzi (i,i+1) , (j, j+1), taką, że usunięcie ich i dodanie krawędzi (i,j+1) i (i+1, j) stworzy najmniejszy cykl Hamiltona.
3. Jeżeli taka para istnieje usuń krawędzie (i,i+1), (j, j+1) i dodaj krawędzie (j, j+1), (i+1,j). Idź do kroku 2.
4. W.p.p. Zakończ algorytm i zwróć TSP.

Po zakończeniu algorytmu TSP jest cyklem 2-optymalnym.

Pseudokod algorytmu 2-opt

```
dmat <- macież wag
#define dist(a,b) dmat[city[a]][city[b]]
city[n] <- dowolny cykl Hamiltona (city[1] -> city[2]->...->city[n]-> city[1])
cost_TSP <- Σ (i=0 to  n-1) dist(i, i%n)
do {
      found_something = false;
      for(i = 0; i < cities-2; i++) {
            for(j = i+2; j < cities; j++) {
                  new_cost_TSP = cost_TSP - dist(i,i+1)-dist(j,j+1) + dist(i,j) +dist(i+1,j+1)
                  if(new_cost_TSP > cost_TSP) {
                        cost_TSP = new_cost_TSP;
                        mini = i; minj = j; found_something = true
                  }
            }
      }
      if found_something { cost_TSP = new_cost_TSP; swap(city[mini], city[minj]))};
}while found_something;
```

Pseudokod algorytmu 2-opt

```
dmat <- macież wag
#define dist(a,b) dmat[city[a]][city[b]]
city[n] <- dowolny cykl Hamiltona (city[1] -> city[2]->...->city[n]-> city[1])
cost_TSP <- ∑ (i=0 to  n-1) dist(i, i%n)
do {
      found_something = false;
      for(i = 0; i < cities-2; i++) {
            for(j = i+2; j < cities; j++) {
                  new_cost_TSP = cost_TSP - dist(i,i+1)-dist(j,j+1) + dist(i,j) +dist(i+1,j+1)
                  if(new_cost_TSP > cost_TSP) {
                        cost_TSP = new_cost_TSP;
                        mini = i; minj = j; found_something = true
                  }
            }
      }
      if found_something { cost_TSP = new_cost_TSP; swap(city[mini], city[minj]))};
}while found_something;
```

Potrzebna pamięc O(n^2)

Długość cyklu liczymy tylko 1 raz: czas: O(n)

Przechodzimy wszystkie pary wierzchołków: czas: O(n^2)

*O(n^2 * liczba iteracji)*

Wnioski

Spodziewana liczba iteracji wynosi : $n^7 * (\log_2)n$

Pesymistyczny czas działania algorytmu 2-opt dla grafu pełnego i skierowanego wynosi $O(n^{10})$.

Jednak rozwiązanie optymalne algorytm znajduje już po n^2 iteracji.

Dodatkowa ilość potrzebnej pamięci: $O(n)$ - Pamięć potrzebna dla tablicy TSP, na przechowanie cyklu.

Całkowita ilość potrzebnej pamięci $O(n^2 + n)$ - Macierz wag krawędzi + tablica TSP.

Algorytm 3-opt

Niezbędne definicje:

waga(T) - suma wszystkich krawędzi grafu T.

3 cykle Hamiltona nazywany 3-sąsiadującymi, jeżeli jeden można uzyskać z drugiego poprzez usunięcie 3 niesąsiednich krawędzi i dodanie 3 krawędzi.

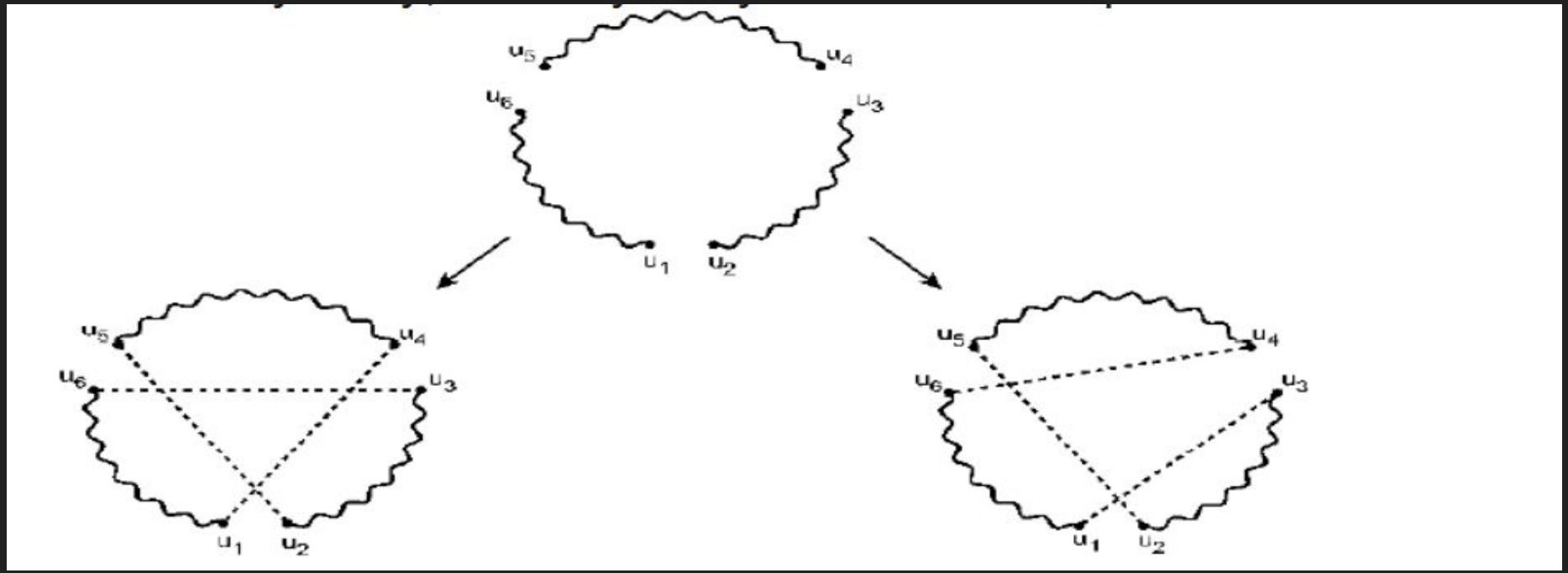
Cykl Hamiltona T nazywamy 3-optymalnym, jeżeli nie istnieje 3-sąsiadujący do T cykl T’, w którym waga(T’) < waga(T)

Opis algorytmu

1. Generujemy dowolny cykl Hamiltona TSP[n]
2. Przeszukaj wszystkie krawędzie w TSP i znajdź 3 najlepsze krawędzie (i,i+1) , (j, j+1), (k, k+1), taką, że usunięcie ich i połączenie 3 innymi stworzy najmniejszy cykl Hamiltona.
3. Jeżeli takie 3 krawędzie istnieją usuń je i zastąp odpowiednimi (dokonaj 3-wymiany), aby stworzyć cykl o mniejszej wadze krawędzi. Idź do kroku 2.
4. W.p.p. Zakończ algorytm i zwróć TSP.

Po zakończeniu algorytmu TSP jest cyklem 3-optymalnym.

3 – wymiana polega na usunięciu 3 krawędzi z cyklu, i ich ponownym połączeniu 3 powstałych dróg w cykl Hamiltona, przy czym w odróżnieniu od 2-wymiany, możemy to wykonać na kilka sposobów:



Pseudokod algorytmu 3-opt

```
dmat <- macież wag
#define dist(a,b) dmat[city[a]][city[b]]
city[n] <- dowolny cykl Hamiltona (city[1] -> city[2]->...->city[n]-> city[1])
cost_TSP <- ∑ (i=0 to  n-1) dist(i, i%n)
do {
      found_something = false;
      for(i = 0; i < cities-2; i++) {
            for(j = i+2; j < cities; j++) {
                  for(k = j+2; k<cities; k++){
                        flow = 0;
                        new_cost1 = cost_TSP - dist(i,i+1) - dist(j,j+1) - dist(k,k+1) +
                                               dist(i,j+1) + dist(i+1,k) + dist(j,k+1)
                        new_cost2 = cost_TSP - dist(i,i+1) - dist(j,j+1) - dist(k,k+1) +
                                               dist(i,j) + dist(j+1,k+1) + dist(i+1,k)
                        if(new_cost1 > cost_TSP) {
                              cost_TSP = new_cost1; flow = 1;
                              mini = i; minj = j; mink =k; found_something = true;
                        }
...
```

Pseudokod algorytmu 3-opt

```
...
                        elsif(new_cost2 > cost_TSP){
                              cost_TSP = new_cost2; flow = 2;
                              mini = i; minj = j; mink =k; found_something = true;
                        }
                  }
            }
      }
      if found_something {
            swap3(city[mini], city[minj], city[mink], flow))
      }
}while found_something;
```

Wnioski

Spodziewana liczba iteracji wynosi : $n^7 * (\log_2)n$

Pesymistyczny czas działania algorytmu 3-opt dla grafu pełnego i skierowanego wynosi $O(n^{11})$.

Jednak rozwiązanie optymalne algorytm znajduje już po n^3 iteracji.

Dodatkowa ilość potrzebnej pamięci: $O(n)$ - Pamięć potrzebna dla tablicy TSP, na przechowanie cyklu.

Całkowita ilość potrzebnej pamięci $O(n^2 + n)$ - Macierz wag krawędzi + tablica TSP.