# Projektowanie Efektywnych Algorytmów

| **Kierunek** | **Termin** |
|---|---|
| *Informatyka* | *Czwartek 19:05* |
| **Temat** | **Problem** |
| *Przeszukiwanie tabu oraz simulowanie wyżarzanie* | *TSP* |
| **Prowadzący** | **data** |
| *mgr inż. Radosław Idzikowski* | *January 11, 2020* |

# 1 Opis problemu

Problem komiwojażera (ang. travelling salesman problem, TSP) jest zagadnieniem optymalizacyjnym, polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Do rozwiązania problemu tym razem autor posłużył się dwoma algorytmami aproksymacyjnymi o których szerzej w dalszej części. Do implementacji rozwiązania posłużył język C++, gdyż zdaniem autora pozwala on na wygodne opisywanie wysokopoziomowej abstrakcji oraz dostępne kompilatory potrafią generować efektywny kod. Testy przeprowadzane były na systemie Ubuntu Linux 19.10 64bit z kompilatorem clang 9.0 na procesorze Intel i5-3570K (4x3.8GHz) z 8GB RAM.

# 2 Struktury danych - sposób przechowywania informacji

## 2.1 Graf

Problem komiwojażera można zdefiniować dla różnej ilości wieszchołków, dalej nazywaną wielkością problemu oraz różnych wag poszczególnych dróg łączących te wierzchołki. Z tego względu wymaganym mechanizmem wczytywania informacji o problemie będzie plik, który zawiera wielkość grafu oraz wagi poszczególnych wierzchołków. Wagi podawane są w postaci macierzy, w której kolejne elementy w wierszu oddzielane są znakami białymi, a same wiersze oddzielane są znakiem nowej lini. Tak zapisane dane należy sparsować i zapisać w postaci pozwalającej na jak najprostszy oraz najszybszy sposób odczytu, ponieważ będzie ona intensywnie wykorzystywana przy rozwiązywaniu problemu. Autor zdecydował się na przechowywanie danych w postaci kwadratowej macierzy sąsiedztwa o rozmiarze równym ilości wierzchołków. I do jej reprezentacji została wykorzystana pojedyncza tablica typów int64_t o rozmiarze $N^2$, gdzie N to rozmiar problemu. Poniżej znajduje się definicja klasy MSTMatrix która implementuje przechowywanie macierzy sąsiedztwa.

Listing 1: Struktura danych MSTMatrix

```
1  namespace pea {
2    class MSTMatrix
3    {
4    private:
5      int64_t *m_data;
6      size_t m_size;
7
8    public:
9      MSTMatrix() noexcept
10       : MSTMatrix(0)
11     {}
12     MSTMatrix(int32_t size) noexcept;
13     MSTMatrix(MSTMatrix &&mm) noexcept;
14     ~MSTMatrix();
15
16     void
17     add_single(Edge edge) noexcept;
18
19     void
20     add(Edge edge) noexcept;
21
22     int32_t
23     get(size_t x, size_t y) const noexcept;
24
25     void
26     set(size_t x, size_t y, int64_t val);
27
```

```
28      // Clears and resizes matrix
29      void
30      resize(size_t newsize) noexcept;
31
32      void
33      display() const noexcept;
34
35      // Returns matrix with zero size on failure
36      static MSTMatrix
37      build_from_file(const char *filename);
38
39      constexpr auto
40      size() const noexcept
41      {
42        return this->m_size;
43      }
44
45      constexpr auto
46      data() const noexcept
47      {
48        return this->m_data;
49      }
50    };
51
52  }; // namespace pea
```

Jeden drobny szczegół pozostaje nie wyjaśniony - typ Edge, który klasa MSTMatrix przyjmuje jako argument przy metodzie dodawania krawędzi do grafu. Jej implementacja jest prosta i sprowadza się do przetrzymywania informacji numeru wierzchołka źródłowego i docelowego, oraz wagi opisywanego połączenia. Metoda MSTMatrix::add korzysta z jej pól, aby odpowiednio wypełnić informacje we własnej macierzy sąsiedztwa. Rozwiązanie problemu będzie zwracane jako typ Path ktory jest lekkim wrapperem na klase std::vector z standardowej biblioteki. Jego implementacja znajduje sie poniżej.

Listing 2: Struktura danych Path

```
1   #pragma once
2   namespace pea {
3
4     using point_type = size_t;
5
6     class Path : public std::vector<point_type>
7     {
8     public:
9       Path(std::initializer_list<point_type> il) noexcept
10        : std::vector<point_type>(il)
11      {}
12
13      template<typename... Args>
14      Path(Args &&... args) noexcept
15        : std::vector<point_type>(std::forward<Args>(args)...)
16      {}
17
18      static Path
19      generate_simple(size_t node_count) noexcept
20      {
21        std::vector<point_type> v(node_count);
```

```
22          std::iota(v.begin(), v.end(), 0);
23          return v;
24        }
25    };
26
27    constexpr cost_t cost_inf = std::numeric_limits<cost_t>::max();
28
29    cost_t
30    cost(const MSTMatrix &matrix, const Path &path) noexcept;
31
32  } // namespace pea
```

Został jeszcze jeden element który wymaga krótkiego wyjaśnienia. Jest to funkcja kosztu scieżki która zwraca koszt danej scieżki na podstawie argumentu w postaci macierzy sąsiedztwa oraz sekwencji kolejno odwiedzanych wierzchołków.

# 3 Algorytm genetyczny

Algorytmy genetyczne polegają na wykorzystują mechanizmy występujące w przyrodzie, a dokładniej próbuje symulować proces ewolucji biologicznej. Na potrzeby rozwiązania problemu definiuje się populacje, która w jest zbiorem możliwych rozwiązań problemów ( w tym przypadku jedną z poprawnych ścieżek). Na danej populacji przeprowadzamy operacje symulujące jej ewolucję w konsekwencji dochodząc do coraz to lepszych rezultatów. Taki wynik otrzymywany jest poprzez zdefiniowanie odpowiednich zasad i mechanizmów zachodzących w czasie tej symulacji. Głownycmi operacjami wykonywanymi w celu symulacji zjawiska ewolucji sa:

**Rozmnażanie** (krzyżowanie) w którym bierze przypadku bierze udział dwoje rodziców, rozwiązań z poprzedniej generacji. Wynikiem tej operacji jest nowe rozwiązanie, które dodane jest do aktualnej póli gatunku.

**Mutacje**, które polegają na wykonaniu losowych "przekłamań" w rozwiązaniu zaraz po utworzeniu nowego osobnika (rozwiązania). Efekt ten zachodzi z zadanych prawdopodobieństem.

**Selekcja**, polegająca na ocenie przystosowania danego osobnika (tutaj koszt scieżki - im mniejszy koszt tym lepiej przystosowany osobnik)

## 3.1 Implementacja - rozmnażanie

Rozmnażanie (ang. crossing) zostało zaimplementawane na dwa sposoby. Pierwszym z nich jest PMX Crossover (Partially-mapped crossover).

**PMX Crossover**

Polega on na losowym wyborze zakresu rozwiązania jednego z rodziców oraz podstawieniu go w to samo miejsce u dziecka. Reszta elementów w genotypie dziecka powinna zostać wypłeniona elementami drugiego rodzica. Takie wypłenienie jednak może spodowować powtarzające sie wierzchołki i dla tego problemu jest nie dopuszczalne. W celu naprawy rozwiązania ( pozbycia sie podwójnie występującuch wierzchołków) wierzchołki pobrane od drugiego z rodziców, jeżeli występują dwuktornie powinny zostać dopowiednio zamienione, tak, aby w ich miejscach pojawiły się wierzchołki dotychczas nie znajdujące sie w genotypie dziecka. Listing kodu z implementacją znajduje się poniżej.

Listing 3: Struktura danych Path

```
void
pmx_patch_(Path &p, const Path &p1, const Path &p2, size_t i, size_t xbegin, size_t xend) FORCE_IN
{
  auto oldnode = p[i];
  point_type newnode;

  const point_type *newnode_p = &p[i];
  while(1)
  {
    newnode = *newnode_p;
    newnode_p = std::find(&p2[xbegin], &p2[xend], oldnode);

    if (newnode_p == &p2[xend])
      break;

    oldnode = p1[newnode_p - &p2[0]];
  }

  p[i] = oldnode;
}

Path
cross_pmx(const Path &p1, const Path &p2)
{
  const auto path_size = std::size(p1);
  Path p(p1);

  // If range starts at the end, then its bad range, so avoid
  // that with shrinking span of path_size.
  auto xbegin = std::rand() % (path_size - 1);

  // xend should point to past last element
  auto xend   = std::rand() % (path_size + 1);

  if (UNLIKELY(xbegin == xend)) ++xend;
  else if (xbegin > xend) std::swap(xbegin, xend);

  std::fill(&p[xbegin], &p[xend], 1337);

  for (auto i = 0ull; i < xbegin; ++i) {
    pmx_patch_(p, p1, p2, i, xbegin, xend);
  }

  for (auto i = xend; i < path_size; ++i) {
    pmx_patch_(p, p1, p2, i, xbegin, xend);
  }

  std::copy(&p2[xbegin], &p2[xend], &p[xbegin]);

  return p;
}
```

**OX Crossover**

Drugim z zastosowanych algorytmów krzyżowania jest OX Crossover, który podobnie jak poprzednik polega na losowym wyborze częsci pierwszego z rodziców, oraz przekopiowaniem go do rozwiązania-dziecka. Następnie zaczynając terować w rodzicu drugim od końca przekopiownego obszaru do jego początku (w trakcie iteracji jeżeli napotkamy koniec, zaczynamy od samego początku) i kolejno wstawiamy wierzchołki które jescze nie wystąpiły. Listing kodu z implementacją znajduje się poniżej.

Listing 4: Struktura danych Path

```
ScoredPath
cross_ox(const ScoredPath &p1, const ScoredPath &p2)
{
  const auto path_size = std::size(p1);
  ScoredPath p(path_size);

  // If range starts at the end, then its bad range, so avoid
  // that with shrinking span of path_size.
  auto xbegin = std::rand() % (path_size - 1);

  // xend should point to past last element
  auto xend   = std::rand() % (path_size + 1);

  if (UNLIKELY(xbegin == xend)) ++xend;
  if (xbegin > xend) std::swap(xbegin, xend);

  // Fill with invalid nodes
  std::fill_n(std::begin(p), p.size(), 999999);

  // Copy random part from p1 to child
  std::copy(&p1[xbegin], &p1[xend], &p[xbegin]);

  // Copy each node from p2 that is not in child
  auto j = xend;
  for (auto &e : p2) {
    if (std::find(std::begin(p), std::end(p), e) == std::end(p)) {
      if (j == path_size)
        j = 0;

      p[j] = e;
      ++j;
    }
  }

  return p;
}
```

# 4 Eksperymenty obliczeniowe

Czas wykonywanych operacji był mierzony na systemie opisanym w wprowadzeniu. Do pomiaru czasu wykorzystano standardową bibliotekę języka C++, std::chrono. Pomiar czasu wykonywany był w nanosekundach. Każdy pomiar wykonywany był 50 razy i ostatecznym wynikiem była średnia czasu z wszystkich prób. Dodatkowo przed dokonywaniem takiego pomiaru uruchamiano 10 razy przebieg mierzonego kawałka kodu bez mierzenia jego czasu po to aby wytrenować układy przewidywania odwoływań do pamięci, tj. branch prediction, cache prefetching. Dodatkowo każdy z algorytmów przed pomiarem został dostrojony (za pomocą paramterów specyficznych dla każdego algorytmu opisanych we wcześniejszych paragrafach) tak aby czas jego wykonania nie odbiegał znacznie od innych algorytmów. Konsekwencją tego są czasy wykonania bardzo zbliżone do siebie, co jest powodem pominięcia przez autora analizy złożoności czasowej. Omówione natomiast zostały ich rezultaty, tj. porównanie wygenerowanej scieżki z najlepszą możliwa dla danego zestawu danych. Poniżej znajduje się kod realizujący mierzenie czasu.

Listing 5: Funkcje pomocnicze do liczenia czasu wykonania algorytmów

```
template<typename Function, typename... Args>
decltype(auto)
measure(Function &&f, Args &&... args) noexcept
{
  auto start = std::chrono::high_resolution_clock::now();
  std::invoke(std::forward<Function>(f), std::forward<Args>(args)...);
  auto end = std::chrono::high_resolution_clock::now();

  return end - start;
}

template<typename Function, typename... Args>
std::chrono::nanoseconds
measure_nano(Function &&f, Args &&... args)
{
  using namespace std::chrono_literals;

  constexpr auto tries = 50;
  auto total_time = 0ns;

  for (auto i = 0; i < tries; ++i) {
    total_time += std::chrono::duration_cast<std::chrono::nanoseconds>(
      measure(std::forward<Function>(f), std::forward<Args>(args)...));
  }

  return total_time / tries;
}

template<typename DataStructure, typename Operation, typename... Args>
std::chrono::nanoseconds
measure_operation_nano(int32_t initial_size,
                       DataStructure &&data_structure,
                       Operation &&op,
                       Args &&... args)
{
  std::invoke(&std::remove_reference_t<DataStructure>::generate,
              std::forward<DataStructure>(data_structure),
              0,
              100,
```

```
40                 initial_size);
41
42     return measure_nano(std::forward<Operation>(op),
43                         std::forward<DataStructure>(data_structure),
44                         std::forward<Args>(args)...);
45 }
46
47 template<typename... Args>
48 void
49 measure_and_log(const char *m_name, const char *f_name, Args &&... args)
50 {
51     FILE *f_out = fopen(f_name, "a");
52     auto time = measure_nano(std::forward<Args>(args)...);
53     fmt::print(f_out, "{};{};\n", m_name, time.count());
54     fclose(f_out);
55 }
```

Poniżej znajdują się wyniki pomiarów zgromadzone dla różnych wariantów każdego z algorytmów.

| Przeszukiwania tabu - metada zamiany (swap) - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 4678691 | 2085 | 2085 | 0 |
| 21 | 8122241 | 2707 | 2707 | 0 |
| 24 | 11574596 | 1298 | 1272 | 26 |
| 26 | 14488670 | 937 | 937 | 0 |
| 29 | 19540981 | 1656 | 1610 | 46 |
| 42 | 55657465 | 777 | 699 | 78 |
| 58 | 140285446 | 30986 | 25395 | 5591 |

| Przeszukiwania tabu - metada wstawiania (insert) - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 5995145 | 3382 | 2085 | 1297 |
| 21 | 10298361 | 4673 | 2707 | 1966 |
| 24 | 14680920 | 3315 | 1272 | 2043 |
| 26 | 18122428 | 2426 | 937 | 1489 |
| 29 | 24541917 | 4855 | 1610 | 3245 |
| 42 | 68625855 | 2934 | 699 | 2235 |
| 58 | 179318868 | 113213 | 25395 | 87818 |

| Przeszukiwania tabu - metada odwrócenia (reverse) - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 5845171 | 2085 | 2085 | 0 |
| 21 | 10171492 | 2707 | 2707 | 0 |
| 24 | 14407409 | 1272 | 1272 | 0 |
| 26 | 18198326 | 937 | 937 | 0 |
| 29 | 24290646 | 1615 | 1610 | 5 |
| 42 | 71764416 | 699 | 699 | 0 |
| 58 | 169588431 | 25395 | 25395 | 0 |

| Simulowane wyżarzanie - liniowa regulacja temperatury - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 143268183 | 2427 | 2085 | 342 |
| 21 | 145967248 | 3568 | 2707 | 861 |
| 24 | 148627322 | 1957 | 1272 | 685 |
| 26 | 150143893 | 1551 | 937 | 614 |
| 29 | 153477613 | 2508 | 1610 | 898 |
| 42 | 163647317 | 1897 | 699 | 1198 |
| 58 | 177813088 | 57551 | 25395 | 32156 |

| Simulowane wyżarzanie - geometryczna regulacja temperatury - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 150094480 | 2085 | 2085 | 0 |
| 21 | 152922148 | 2757 | 2707 | 50 |
| 24 | 155708966 | 1290 | 1272 | 18 |
| 26 | 157297797 | 965 | 937 | 28 |
| 29 | 160790359 | 1683 | 1610 | 73 |
| 42 | 171444619 | 747 | 699 | 48 |
| 58 | 186285346 | 28461 | 25395 | 3066 |

| Simulowane wyżarzanie - logarytmiczna regulacja temperatury - start losowy | | | | |
|---|---|---|---|---|
| Wielkość problemu | Czas wykonania (ns) | Koszt znalezionej drogi | Koszt najlepszej drogi | błąd |
| 17 | 2090093 | 2085 | 2085 | 0 |
| 21 | 2243627 | 2760 | 2707 | 53 |
| 24 | 2072114 | 1330 | 1272 | 58 |
| 26 | 2103146 | 1032 | 937 | 95 |
| 29 | 2152118 | 1730 | 1610 | 120 |
| 42 | 2366575 | 958 | 699 | 259 |
| 58 | 2634296 | 27326 | 25395 | 1931 |

# 5 Wnioski

Algorytm genetyczny, mimo polegania w dużym stopniu na procesach losowych daje dobre rezultaty. Strategią krzyżowania, która najlepiej sprawdziła się w tym przypadku jest OX Crossover w połączeniu z selekcja w trybie rankingowym. Bardzo ważnym elementem algorytmu okazało się wymiana aktualnej populacji mimo otrzymywania chilowo gorszych wyników z jednoczesnym zachowaniem częsci najlepszej populacji z generacji poprzedniej. Prostota implementacji tego algorytmu oraz jego efektywność zdecydowanie umieszcza go na pozycji algorytmów którego powinny być rozważone przy rozwiązywaniu innych problemów.