Projektowanie Efektywnych Algorytmów

Kierunek	**Termin**
Informatyka	*Czwartek 19:05*
Temat	**Problem**
Metoda przeglądu zupełnego, podzialu i ograniczeń, oraz programowanie dynamiczne.	*TSP*
Prowadzący	**data**
Patryk Wlazłyń	*December 17, 2019*

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 wymagana jest implementacja kilku struktur danych o których szerzej w dalszej częsci. 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, 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 Metoda rozwiązania problemu

3.1 Przegląd zupełny

Metoda przeglądu zupełnego jest trywialna w koncepcji. Polega ona na przejściu przez wszystkie możliwe sposoby przebycia ścieżki w trakcie zapisując tą o dotychczasowo najniższym koszcie. Tak, aby można było później ją odczytać. Z definicji problem wymaga odwiedzenia każdego wierzchołka dokładnie jeden raz, więc kolejnymi sposobami na przejscie przez graf są zwyczajnie permutacje zbioru wierzchołków. Wystarczy zatem wygenerować każdą permutację i sprawdzić która jest najbardziej opłacalna. To oznacza, że wystarczy każdą permutację wysłać do funkcji kosztu. Generowanie permutacji zostaje rozwiązane za pomocą własnej implementacji metody wykorzystanej w standardowej bibliotece pod nazwą std::next_permutation, która jako argument przyjmuje posortowany kontener i generuje jego kolejną permutację w kolejności rosnącej zwracając false przy wygenerowaniu kompletnie posortowanego kontenera (stan początkowy). Implementacja funkcji realizującej przegląd zupełny znajduje się poniżej.

Listing 3: Kod metody przeglądu zupełnego

```
1  namespace pea {
2    Path
3    bt(const MSTMatrix &matrix, cost_t *costp)
4    {
5      Path p = Path::generate_simple(matrix.size());
6      Path res;
7      cost_t rescost = std::numeric_limits<decltype(rescost)>::max();
8
9      do {
10       cost_t tmp_cost = cost(matrix, p);
11       if (tmp_cost < rescost) {
12         rescost = tmp_cost;
13         res = p;
14       }
15     } while (::next_permutation(std::begin(p), std::end(p)));
16
17     if (costp)
18       *costp = rescost;
19
20     return res;
21   }
22 } // namespace pea
```

3.2 Metoda podziału i ograniczeń

Tytułowa metoda polega na przechodzeniu po grafie jak po strukturze drzewiastej. W tym przypadku autor zastosował algorytm DFS (depth first search) w wersji rekurencyjnej. Algorytm polega na wybraniu dowolnego wierzchołka i potraktowania go jako korzenia drzewa i explorowania drzewa najgłębiej jak się da i dopiero gdy spotykamy koniec drzewa wrócić do rozważenia ścieżek wcześniej pominiętych. W trakcie przemirzeania drzewa należy sumować aktualny koszt przy każdym wykonanym kroku. I jeżeli w trakcie przechodzenia zdarzy się tak, że aktualny koszt drogi jest w danym momencie większy od ostatniego wyniku składającego się z pełnej ścieżki, wówczas kontynuowanie przeglądu aktualnej ścieżki jest pozbawione sensu, gdyż droga wraz z podróżowaniem nie może stać się krótsza. W takim przypadku należy wrócić do przeglądania pominiętych scieżek. Do wykonania opisanego przeszukiwania autor zdecydował się zastosować tablicę wartości true / false aby oznaczać już odwiedzone wierzchołki. Poniżej znajduje się rekurencyjna implementacja wspomnianaego algorytmu.

Listing 4: Kod metody podziału i ograniczeń

```
namespace pea {
  void
  tspdfs::TSP(size_t v) noexcept
  {
    size_t u;
    lpath.push_back(v);

    if (lpath.size() < matrix.size()) {

      visited[v] = true;

      for (u = 0; u < matrix.size(); u++) {

        if (!visited[u]) {
          lcost += matrix.get(v, u);
          if (lcost < rcost)
            TSP(u);
          lcost -= matrix.get(v, u);
        }
      }

      visited[v] = false;

    } else {
      // Path is complete, close it
      lcost += matrix.get(v, 0);

      // If local result is better than currently best
      // Copy local to best
      if (lcost < rcost) {
        rcost = lcost;
        rpath = lpath;
      }

      lcost -= matrix.get(v, 0);
    }

    lpath.pop_back();
  }
} // namespace pea
```

3.3 Metoda programowania dynamicznego

Metoda programowania dynamicznego polega na takim rozwiązywaniu problemu aby po drodzę rozwiązywać podproblemy których wyniki można później wykorzystać aby przyspieszyć obliczenia kolejnych etapów algorytmu, W tym przypadku autor skorzystał z algorytmu Helda-Karpa, który do rozwiązania poruszanego problemu korzysta z zależności problemu komiwojażera, że *Każda podścieżka scieżki bedącej optymalnym rozwiązaniem jest optymalnym rozwiązaniem*, co można zparafrazować jako właściwosc, że znajdując optymalną scieżkę dla części węzłów mamy gwarancje, że jest ona częścią pełnej (składającej się ze wszystkich wierzchołków) optymalnej scieżki, która jest rozwiązaniem poruszanego problemu. Wadą tego podejścia do problemu jest to, że wyniki podścieżek muszą cały czas być w pamięci na wypadek gdyby były potrzebne jako podścieżki do kolejnego rozwiązania, a mnogość takich podrozwiązań powodują duże zużycie pamięci, które cechuje się złożonością $2^N * N$. Autor postanowił przechowywać wyniki podproblemów w postaci tablicy, właśnie 2^N na N. Tablica przechowuje wyniki dla każdego podzbioru problemu, który reprezentowany jest maską bitową złożoną z N bitów (dla każdego wierzchołka jeden bit), gdzie bit o wartości true oznacza, że wierzchołek wchodzi w skład zbioru a wartość false, odwrotnie. Dodatkowo dla każdego takiego zbioru wyznaczany jest jego wierzchołek końcowy (stąd kolejnych N możliwości zakończenia). Poniżej znajduje się implementacja algorytmu.

Listing 5: kod metody programowania dynamicznego

```
#include "combgen.hpp"
#include "hk.hpp"
#include <fmt/format.h>

namespace pea {

  static Path
  hk_find_optimal_path(const MSTMatrix &matrix, MemoTable &memo) noexcept
  {
    constexpr point_type start_node = 0;
    auto node_count = matrix.size();
    std::bitset<max_problem_size> state = (1 << node_count) - 1;
    Path path;
    path.resize(matrix.size());
    size_t last = start_node;

    for (size_t i = node_count - 1; i >= 1; --i) {

      cost_t minDist = cost_inf;
      size_t end_node = start_node;

      for (size_t e = 0; e < node_count; ++e) {

        if (!hk_isset(e, state) || e == start_node)
          continue;

        cost_t newDist = memo.get(e, state.to_ulong());
        newDist += matrix.get(e, last);

        if (newDist < minDist) {
          minDist = newDist;
          end_node = e;
        }
      }

      last = end_node;
```

```
37        path[i] = end_node;
38        state.reset(end_node);
39      }
40
41      return path;
42    }
43
44    static Path
45    hk_solve(const MSTMatrix &matrix, MemoTable &memo)
46    {
47      constexpr point_type start_node = 0;
48
49      // Because implementation uses bitset it requires it.
50      // std::vector cen be used, but lets be honest --
51      // bitset is much better than std::vector<bool>,
52      // the drawback is the static aspect of it ;/
53      assert(matrix.size() <= max_problem_size);
54      if (matrix.size() > max_problem_size)
55        return {};
56
57      auto node_count = matrix.size();
58
59      for (size_t r = 3; r <= node_count; ++r) {
60        for (auto &subset : comb<max_problem_size>(r, node_count)) {
61          if (!hk_isset(start_node, subset))
62            continue;
63          for (size_t next = 0; next < node_count; ++next) {
64
65            if (next == start_node || !hk_isset(next, subset))
66              continue;
67
68            std::bitset<max_problem_size> state = subset;
69            state.reset(next);
70
71            cost_t minDist = cost_inf;
72
73            for (size_t e = 0; e < node_count; ++e) {
74              if (e == start_node || e == next || !hk_isset(e, subset))
75                continue;
76
77              cost_t newDist =
78                memo.get(e, state.to_ulong()) + matrix.get(e, next);
79
80              if (newDist < minDist)
81                minDist = newDist;
82
83            }
84
85            memo.set(next, subset.to_ulong(), minDist);
86          }
87        }
88      }
89
90      return hk_find_optimal_path(matrix, memo);
91    }
```

```
92
93    Path
94    hksolve(const MSTMatrix &matrix)
95    {
96      auto node_count = matrix.size();
97      MemoTable memo(node_count, (1 << node_count));
98
99      hk_setup(matrix, memo);
100     Path p = hk_solve(matrix, memo);
101
102     return p;
103   }
104 } // namespace pea
```

Powyższy listing zawiera definicję dwóch kluczowych funkcji. Jedną z nich jest hk_solve która jest głowną funkcją rozwiązującą rozważany problem. hk_solve wylicza koszty dla każdego podzbioru pełnej scieżki dla każdego jej możliwego zakończenia i zapisuje jej wynik, który wykorzystywany jest w kolejnych iteracjach algorytmu, przy obliczaniu bardziej licznych zbiorów. Gdy funkcja ta zakończy działanie w tablicy mamy już zapisane optymalne rozwiązania dla każdego podzbioru kończoncego sie w każdym z punktów. Kolejnym krokiem jest odtworzenie z tych danych optymalnej scieżki dla całego grafu. W tym celu wykorzystana jest funkcja hk_find_optimal_path. Wspomniana funkcja przechodzi niejako od tyłu, z gotowej scieżki dobiera najbardziej opłacalne zakończenia i wybiera je jako ostateczny wynik. Po przjściu przez wszystkie poziomy drzewa otrzymujemy optymalną scieżkę którą możemy zwrócić jako wynik.

4 Eksperymenty obliczeniowe

Czas wykonywanych operacji był mierzony na systemie opisanym w wprowadzeniu. Do pomiaru czasu wykorzystano standardową bibliotekę języka C++, biblioteke 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. Poniżej znajduje się kod realizujący mierzenie czasu.

Listing 6: kod metody programowania dynamicznego

```
1  #pragma once
2  #include <chrono>
3  #include <fmt/format.h>
4  #include <functional>
5  #include <utility>
6
7  namespace pea {
8    template<typename Function, typename... Args>
9    decltype(auto)
10   measure(Function &&f, Args &&... args) noexcept
11   {
12     auto start = std::chrono::high_resolution_clock::now();
13     std::invoke(std::forward<Function>(f), std::forward<Args>(args)...);
14     auto end = std::chrono::high_resolution_clock::now();
15
16     return end - start;
17   }
18
19   template<typename Function, typename... Args>
```

```
20    std::chrono::nanoseconds
21    measure_nano(Function &&f, Args &&... args)
22    {
23      using namespace std::chrono_literals;
24
25      constexpr auto tries = 50;
26      auto total_time = 0ns;
27
28      for (auto i = 0; i < tries; ++i) {
29        total_time += std::chrono::duration_cast<std::chrono::nanoseconds>(
30          measure(std::forward<Function>(f), std::forward<Args>(args)...));
31      }
32
33      return total_time / tries;
34    }
35
36    template<typename DataStructure, typename Operation, typename... Args>
37    std::chrono::nanoseconds
38    measure_operation_nano(int32_t initial_size,
39                           DataStructure &&data_structure,
40                           Operation &&op,
41                           Args &&... args)
42    {
43      std::invoke(&std::remove_reference_t<DataStructure>::generate,
44                  std::forward<DataStructure>(data_structure),
45                  0,
46                  100,
47                  initial_size);
48      return measure_nano(std::forward<Operation>(op),
49                          std::forward<DataStructure>(data_structure),
50                          std::forward<Args>(args)...);
51    }
52
53    // Doing measurements and logs results.
54    // passes args to measure_operation_nano.
55    template<typename... Args>
56    void
57    measure_and_log(const char *m_name, const char *f_name, Args &&... args)
58    {
59      FILE *f_out = fopen(f_name, "a");
60      auto time = measure_nano(std::forward<Args>(args)...);
61      fmt::print(f_out, "{};{};\n", m_name, time.count());
62      fclose(f_out);
63    }
64  } // namespace pea
```

Metoda przeglądu zupełnego	
Wielkość problemu	Czas wykonania (ns)
2	328
3	691
4	3026
5	17928
6	80026
7	442466
8	3953679
9	39880219
10	442507917
11	5353492295

Metoda przeszukiwanuia w głąb	
Wielkość problemu	Czas wykonania (ns)
2	225
3	699
4	1469
5	6033
6	21570
7	84161
8	179542
9	730287
10	2420364
11	5190784
12	92879242
13	546526194
14	103633643

Metoda dynamiczna Helda-Karpa	
Wielkość problemu	Czas wykonania (ns)
2	205
3	854
4	2780
5	3429
6	8669
7	22836
8	51377
9	117697
10	288131
11	651784
12	1481028
13	3447888
14	7751161
15	17600188
16	41210060
17	94666222
18	230991845

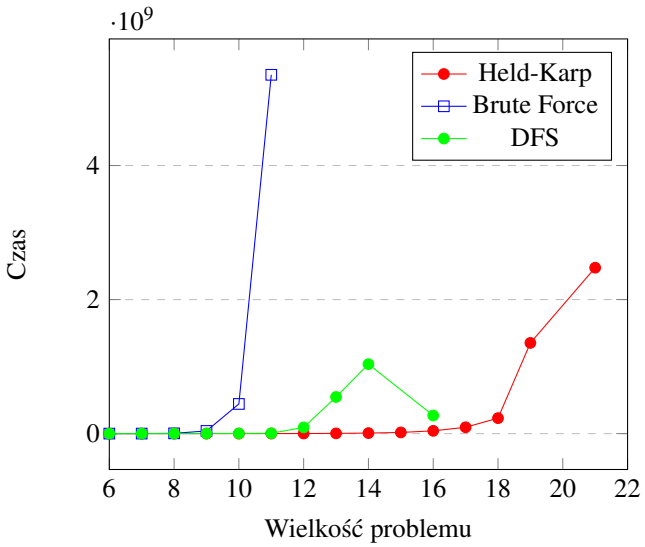

Figure 1: Czas wykonywania się algorytmów w zależności od wielkości problemu

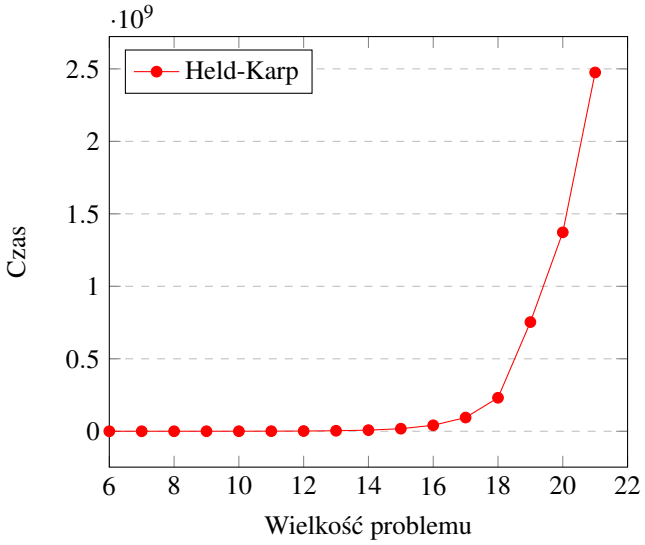

Figure 2: Czas wykonywania się algorytmów w zależności od wielkości problemu

5 Wnioski

Algorytm przeglądu zupełnego okazał się być (jak oczekiwano) najmniej efektywnym sposobem rozwiązania problemu. Biorąc pod uwage implementacje generowania permutacji nie jest też metoda najprostszą do zaimplementowania z tych prezentowanych. Algorytm polega na porównaniu każdej możliwej scieżki wiec jego złożoność to n!, gdyż jest to równoważne z wygenerowaniem każdej możliwej permutacji zbioru wierzchołków. Metoda jest zwyczajnie nie praktyczna.

Algorytm przeglądania drzewa w głąb okazał się być najprostszy do zaimplementowania. Dla każdego nie odwiedzonego jeszcze sąsiada wywołujemy rekurencyjnie procedure. Dzięki wprowadzeniu dodatkowego sprawdzania aktualnego kosztu drogi możemy wcześniej stwierdzić już na danym etapie, że dane rozwiązanie nie może dać lepszego rezultatu niż najlepszy dotychas znany, wówczas możemy "odciąć" to rozwiązanie i przejść do sąsiedniego rozgałęzienia. Ta metoda jednak nie daje dużo lepszych rezultatów czasowych. Jest co prawda nieco szybsza, a tempo wzrostu w praktyce okazuje się dużo bardziej "gładkie", ale nadal nie daje zadowalających rezultatów. Dodatkowo dla różnych danych dostajemy kompletnie różne złożoności, które wydają się nie być opisywalne żadną uniwersalną złożonością. Wynika to z tego, że sytuacje w których faktycznie "odcinamy" nie opłacalne rozgałęzienia mogą pojawiać się bardzo późno w trakcie przeszukiwania co prowadzi do tego, że i tak algorytm bedzie musiał przejść przez większość drzewa.

Algorytm programowania dynamicznego Helda-Karpa cechuje sie złożonością czasową $O(2^n n^2)$. Jego wadą jest jednak duża złożoność pamięciowa ze względu na alokacje dużej tablicy do zapisywania wyników dla podzbiorów wierzchołków, która wynosi $O(2^n n)$. Właśnie przez swoją złożoność pamięciową oraz (nadal) długi czas wykonania algorytm nie przydaje sie w praktycznych rozwiązaniach.