Metoda symulowanego wyżarzania

w wersji równoległej

Część sekwencyjna i równoległa z pamięcią wspólną

Projekt PORR 2011Z

**Zespół projektowy:**

Jan Wilczak

Adrian Wiśniewski

Spis treści

[Treść zadania 3](#_Toc312105275)

[Definicja problemu 3](#_Toc312105276)

[Symulowane wyżarzanie 3](#_Toc312105277)

[Implementacja sekwencyjna 4](#_Toc312105278)

[Opis poszczególnych plików źródłowych 5](#_Toc312105279)

[Algorithm.cpp 5](#_Toc312105280)

[Random.cpp 5](#_Toc312105281)

[Timer.cpp 5](#_Toc312105282)

[TspProblem.cpp 5](#_Toc312105283)

[main.cpp 5](#_Toc312105284)

[Implementacja równoległa z pamięcią wspólną (OpenMP) 6](#_Toc312105285)

[Testy wydajnościowe 7](#_Toc312105286)

[Opcje programu 9](#_Toc312105287)

# Treść zadania

Metoda symulowanego wyżarzania w wersji równoległej. Rozwiązać zadanie komiwojażera. Sformułować zadania dla komiwojażera (lista miast, które należy odwiedzić) i rozwiązać zadanie stosując metodę symulowanego wyżarzania. Rozważyć przynajmniej trzy zadania o różnym stopniu złożoności (wymiarze). Ocenić przyspieszenie obliczeń w zależności od stopnia zrównoleglenia.

# Definicja problemu

Problem komiwojażera polega na znalezieniu najkrótszego cyklu Hamiltona w grafie. W prezentowanym rozwiązaniu przyjęliśmy, że graf ten jest grafem pełnym. Jego reprezentacją jest macierz n\*n liczb zmiennoprzecinkowych, której poszczególne komórki oznaczają odległość między miastem i-tym i j-tym. Odległości te mogą być różne w obu kierunkach. Rozwiązaniem jest wektor n liczb całkowitych nieujemnych oznaczających kolejno odwiedzane miasta oraz koszt cyklu.

# Symulowane wyżarzanie

Heurystyka ta polega na wygenerowaniu początkowego rozwiązania roboczego, a następnie wyborze dowolnego z jego sąsiadów i akceptacji zgodnie z następującym kryterium:

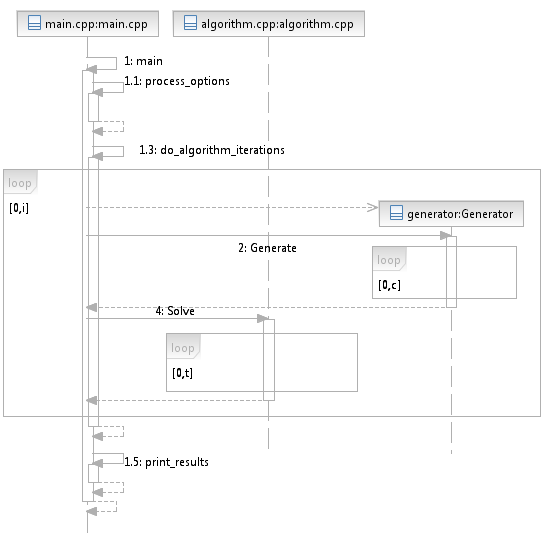
* Jeżeli sąsiad jest lepszy lub taki samo jako bieżące, zaakceptuj go;
* Jeżeli sąsiad jest gorszy, zaakceptuj go z prawdopodobieństwem zgodnym z rozkładem Boltzmanna, zależnym od różnicy kosztu obu rozwiązań i bieżącej temperatury. Temperatura z kolei zależy od schematu wyżarzania i maleje z każdą kolejną iteracją.

Zaakceptowane rozwiązanie staje się nowym punktem roboczym. Wybór nowego rozwiązania należy powtarzać aż do osiągnięcia założonej minimalnej temperatury. Najlepsze napotkane rozwiązanie staje się rozwiązaniem końcowym.

W zaimplementowanym programie generator rozwiązań początkowych wybiera cykl będący losową permutacją miast. Schemat wyżarzania jest funkcją liniową i z każdą kolejną iteracją temperatura maleje o stały krok. Natomiast sąsiedztwo rozwiązania definiujemy poprzez operację zamiany kolejności dwóch dowolnych miast występujących w cyklu. Obliczenie kosztu sąsiada nie wymaga ponownego liczenia kosztu całego cyklu, a jedynie odjęcia kosztów krawędzi usuwanych z rozwiązania i dodania kosztów krawędzi wprowadzanych. Złożoność obliczeniowa pojedynczego przebiegu metody to O(c+t), gdzie c to liczba miast, a t liczba kroków potrzebnych do osiągnięcia minimalnej temperatury. Pierwszy składnik bierze się z konieczności generacji i obliczenia kosztu początkowego cyklu, natomiast drugi odpowiada ilości kroków symulowanego wyżarzania, z których każdy ma koszt stały. Metoda jest uruchamiana wielokrotnie z różnymi punktami startowymi. Ilość iteracji zależy od argumentu przekazywanego aplikacji. Całkowita złożoność obliczeniowa wynosi więc O(i\*(c + t)).

# Implementacja sekwencyjna

Program został napisany przy użyciu języka C++ i skompilowany z użyciem kompilatorów msvc oraz g++. Składa się z pięciu plików źródłowych. Struktura wywołań najważniejszych funkcji programu została ujęta na poniższym diagramie:



Ilustracja 1: Diagram sekwencji programu

## Opis poszczególnych plików źródłowych

### Algorithm.cpp

Zawiera definicje następujących klas:

* SimulatedAnnealing::Solution – pojedyncze rozwiązanie problemu
* SimulatedAnnealing::InitialGenerator – generator rozwiązań początkowych
* SimulatedAnnealing::Schedule – schemat wyżarzania (implementacja liniowa)
* SimulatedAnnealing::Variation – operator wariacji generujący sąsiada rozwiązania.

Oraz metodę Solve rozwiązującą problem komiwojażera zgodnie z heurystyką symulowanego wyżarzania o następującej sygnaturze:

|  |
| --- |
| void Solve(const TspProblem &problem, Schedule &schedule,  Variation &variation, Solution &solution); |

Listing Deklaracja metody Solve

### Random.cpp

Implementacja generatora liczb losowych, niezależnego od platformy i bezpiecznego w środowisku wielowątkowym. Korzysta z funkcji rand w systemie operacyjnym Windows i rand\_r w pozostałych zgodnych ze standardem POSIX.

### Timer.cpp

Implementacja czasomierza wysokiej precyzji niezależnego od platformy.

### TspProblem.cpp

Klasa opisująca problem komiwojażera za pomocą macierzy odległości. Umożliwia także zapisywanie i odczytywanie danych z pliku.

### main.cpp

Zawiera punkt wejściowy programu. Odpowiada za przetworzenie opcji podanych w linii komend, uruchomienie algorytmu, wypisanie wyników oraz mierzenie czasu wykonania poszczególnych części programu.

# Implementacja równoległa z pamięcią wspólną (OpenMP)

Implementacja równoległa korzysta z tego samego kodu źródłowego co sekwencyjna. Dodano trzy dyrektywy OpenMP pozwalające na zrównoleglenie programu. Pierwsze dwie to parallel for oraz critical dotyczące pętli głównej algorytmu, uruchamiającej kolejne przebiegi symulowanego wyżarzania dla różnych punktów początkowych. Pozwalają podzielić kolejne iteracje pętli między wątkami oraz wyłonić najlepsze rozwiązanie w sekcji krytycznej. Trzecia dyrektywa – threadprivate – jest używana przez generator liczb losowych do przechowywania ziarna między kolejnymi wywołaniami.

|  |
| --- |
| void do\_algorithm\_iterations(const TspProblem &problem, int iterations, SimulatedAnnealing::Solution &bestSolution, int seed) {  bestSolution.cost = std::numeric\_limits<float>::max();  #pragma omp parallel for  for(int i = 0; i < iterations; ++i) {  // create all needed elements  // initialize random generator  // generate initial solution  // solve problem  SimulatedAnnealing::Solve(problem, schedule, variation, solution);  // update best solution  #pragma omp critical  {  if(solution.cost < bestSolution.cost ) {  bestSolution = solution;  }  }  }  } |

Listing Zrównoleglona funkcja do\_algorithm\_iterattions

|  |
| --- |
| namespace Random {  static unsigned seed;  #pragma omp threadprivate ( seed )  unsigned Next() {  return rand\_r(&Random::seed);  }  } |

Listing Wykorzystanie dyrektywy threadprivate przez generator liczb losowych

# Testy wydajnościowe

Testy wykonano w celu zbadania faktycznego przyspieszenia programu na szesnastordzeniowej maszynie Sun. Porównano wersję sekwencyjną programu oraz równoległą z różną ilością wątków. Program uruchamiano dla rozmiaru problemu 100, 1 000 oraz 10 000 z następującymi ustawieniami:

* Ilość iteracji algorytmu = 32;
* Ilość kroków schematu wyżarzania = 5 000 000;
* Pozostałe opcje przyjmują wartości domyślne.

Ilustracja 2: Wykres czasu obliczeń w zależności od wielkości problemu dla różnych opcji zrównoleglenia.

Wyniki zostały zawarte w powyższym wykresie, na którym doskonale widać liniową złożoność obliczeniową algorytmu. Program można podzielić na trzy części: przygotowanie danych, obliczenia oraz wypisani wyników. Jedynie druga z wymienionych części jest zrównoleglona. Pozostałe dla zadanej wielkości problemu mają stały czas wykonania, niezależny od wersji i użytej ilości wątków. Korzystając z prawa Amdahla można obliczyć[[1]](#footnote-1), że maksymalny współczynnik przyspieszenia dla kolejnych badanych rozmiarów problemów wynosi około 32/32/22. Pierwsze dwa wyniki świadczą o tym, że czas poświęcony części sekwencyjnej jest pomijalny dla małych rozmiarów problemów, a każdą iterację może obsłużyć inny wątek. Natomiast przy większych rozmiarach zaczyna ujawniać się kwadratowa złożoność obliczeniowa związana z przygotowaniem macierzy odległości, znajdująca się w części sekwencyjnej, która skutecznie ogranicza maksymalne przyspieszenie.

Wersja równoległa uruchomiona na wszystkich 16 rdzeniach chodziła 12/9/8 razy szybciej niż wersja sekwencyjna[[2]](#footnote-2), która oczywiście wymagała najwięcej czasu. Wydajność spadała liniowo wraz ze zmniejszaniem liczby używanych rdzeni. Włączenie opcji dynamicznego podziału pętli spowodowało aż 20 procentowy wzrost czasu wykonania. Jest to zrozumiałe, ponieważ iteracje algorytmu są do siebie bardzo zbliżone i wymagają podobnej ilości czasu, dlatego włączenie tej opcji generuje dodatkowy narzut związany z komunikacją. Ponadto uruchomienie wersji równoległej z tylko jednym dostępnym rdzeniem nie obniżyło wydajności programu.

# Opcje programu

Program przyjmuje następujące argumenty w dowolnej kolejności:

|  |  |  |
| --- | --- | --- |
| Argument | Opis | Wartość domyślna |
| -c liczba\_całkowita | Liczba miast | brak |
| -i liczba\_całkowita | Liczba iteracji metody | 1 |
| -r liczba\_całkowita | Ziarno generatora liczb losowych | time(0) |
| -t liczba\_rzeczywista | Temperatura początkowa | 10^4 |
| -m liczba\_rzeczywista | Temperatura minimalna | 0,01 |
| -s liczba\_całkowita | Liczba kroków schematu wyżarzania | 10^6 |
| -f nazwa\_pliku | Plik z problemem do wczytania | brak |
| -o nazwa\_pliku | Plik do którego zapisać problem | brak |

Program wymaga podania albo ilości miast albo pliku wejściowego. Nie wszystkie kombinacje dozwolonych argumentów są sensowne. W przypadku wykrycia sprzeczności w podanych opcjach program wypisuje komunikat o błędzie i przerywa działanie.

1. Wszystkie dane i obliczenia znajdują się w załączonym skoroszycie. [↑](#footnote-ref-1)
2. Należy zwrócić uwagę, że użyto dwukrotnie większej liczby iteracji niż liczba dostępnych rdzeni procesora. Dla 16 iteracji i rozmiaru problemu 10 000 maksymalne przyspieszenie wynosi 11x, a program osiągnął 7x. [↑](#footnote-ref-2)