

Rozwiązanie zadania komiwojażera przy pomocy algorytmu genetycznego.

1) Wersja sekwencyjna i zrównoleglona (pamięć wspólna).

Onaszkiewicz Przemysław, Gadawski Łukasz

30 listopada 2015

1 Cel zadania

Celem pierwszego etapu projektu jest implementacja wersji sekwencyjnej algorytmu genetycznego działającego na pojedynczym procesorze do rozwiązywania problemu komiwojażera. A także zastosowanie dyrektyw *OpenMP* w celu dokonania zrównoleglenia wykonania algorytmu na wielu procesorach z pamięcią wspólną.

2 Problem komiwojażera

Problem zdefiniowany jest następująco: mając listę miast oraz odległości między nimi, znajdź najkrótszą, dozwoloną drogę (początkiem oraz końcem drogi jest ten sam wierzchołek), zawierającą każde miasto dokładnie raz. Przy czym można zacząć od dowolnego miasta oraz kolejność miast jest dowolna. Inaczej problem polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Problem należy do zbioru problemów NP-trudnych.

3 Algorytm genetyczny

W 1975 roku przez Johna Hollanda został wynaleziony algorytm genetyczny, którego zadaniem było działanie analogiczne do procesu ewolucji na ziemi. Pseudokod prostego algorytmu genetycznego (ang. *Simple Genetic Algorithm*) został przedstawiony w pseudokodzie *Algorithm 1*.

Sterowanie algorytmem genetycznym odbywa się poprzez warunek stopu, którym może być satysfakcjonująca wartość funkcji celu lub ilość generacji. Typowy algorytm genetyczny rozpoczyna się inicjacją populacji wstępnej. W przypadku problemu komiwojażera **populacją** będzie lista osobników, natomiast każdy **osobnik** będzie konkretną listą miast, czyli jednym ze stanów z całej przestrzeni stanów. W przypadku problemu komiwojażera przestrzenią stanów jest zbiór wszystkich możliwych kombinacji dróg pomiędzy miastami. Następnie dokonywana jest ocena wartości każdego osobnika w populacji. W naszym

Algorithm 1 Simple Genetic Algorithm

```
1: procedure SIMPLE GENETIC ALGORITHM
2:    $t \leftarrow 0$ 
3:   initialize  $P^t$ 
4:   evaluate  $P^t$ 
5:   while (not stop_condition):
6:      $O^t \leftarrow$  reproduce  $P^t$ 
7:     crossover  $O^t$ 
8:     mutate  $O^t$ 
9:     evaluate  $O^t$ 
10:     $P^{t+1} \leftarrow O^t$ 
11:     $t \leftarrow t + 1$ 
```

przypadku będzie to obliczenie drogi każdego z zainicjowanych stanów. Kolejno następuje *reprodukcja* populacji, która zostanie opisana dokładniej w kolejnym punkcie. Następnie wykonywane są operacje krzyżowania oraz mutacji, czyli "urozmaicanie" aktualnej populacji, a w dalszej części wytypowanie populacji, która będzie stanowiła listę najlepszych osobników do reprodukcji w kolejnej generacji.

3.1 Reprodukacja

Reprodukacja polega na wytypowaniu osobników do populacji potomnych, ale tak aby preferować osobniki lepiej przystosowane, czyli posiadające lepszą wartość funkcji przystosowania. Ocena jakości osobników dokonywana jest na podstawie funkcji przystosowania. W przypadku problemu komiwojażera mniejsza droga jest lepszym rezultatem, a zatem dąży się do minimalizacji drogi, którą reprezentuje każdy osobnik. Podczas reprodukcji następuje losowy wybór osobników do populacji potomnej, ale prawdopodobieństwo wylosowania każdego z nich nie jest jednakowe. W przypadku osobników lepiej przystosowanych występuje większe prawdopodobieństwo. Wartość prawdopodobieństwa obliczana jest zgodnie z następującym wzorem:

$$p_i = \frac{\text{len_sum} - \text{len}_i}{\sum_i (\text{len_sum} - x_i)}, \text{ gdzie}$$

len_sum - suma długości wszystkich osobników w populacji,

len_i - długość i-tego osobnika, dla którego jest liczone prawdopodobieństwo,

Następnie mając prawdopodobieństwa osobników w populacji następuje obliczenie dystrybucyj prawdopodobieństwa. Podczas losowania osobników następuje losowanie liczby z przedziału (0.00, 1.00), a następnie wybranie osobnika, który odpowiada danej wartości dystrybucyj. Taki dobór nazywany jest selekcją ruletkową (ang. *round-wheel selection*).

3.2 Mutacja

Operacja mutacji wykonywana jest z określoną wartością prawdopodobieństwa, która jest parametrem algorytmu. Zgodnie z prawdopodobieństwem podejmowana jest decyzja o zmutowaniu konkretnego osobnika. Proces polega na losowej zamianie pozycji dwóch miast w początkowej liście miast. Dzięki temu zostaje wprowadzone całkowicie losowe zróżnicowanie dwóch osobników. Przeważnie

wartość tego parametru ustawiona jest na niską wartość, tak aby wprowadzać różnorodność w każdym pokoleniu, ale aby również nie zakłócać polepszania wartości funkcji dostosowania w pokoleniu.

3.3 Krzyżowanie

Operacja krzyżowania również ma na celu wprowadzenie różnorodności w osobnikach w kolejnych pokoleniach. Jest wykonywana z określoną wartością prawdopodobieństwa, która jest parametrem algorytmu. Wszystkie osobniki w populacji są dobierane w pary. Następnie dla każdej z par zgodnie z parametrem jest podejmowana decyzja o dokonaniu operacji krzyżowania. W dalszej kolejności losowane zostają indeksy krzyżowania. Elementy osobników pomiędzy tymi indeksami zostają zamienione między osobnikami. A elementy osobników znajdujące się poza indeksami zostają przepisane począwszy od wylosowanego indeksu. Geny(miasta), które już znajdują się w osobniku są pomijane. T gwarantuje że osobnik po krzyżowaniu będzie osobnikiem prawidłowym (każde miasto będzie odwiedzane tylko raz).

4 Implementacja

4.1 Tworzenie mapy miast

W naszej implementacji mapa została odwzorowana jako obiekt posiadający wielkość, zbiór miast oraz strukturę danych *mapa* gdzie kluczem jest para miasto-miasto oraz odległość między taką na mapie.

4.1.1 Wczytanie z pliku

Mapę odległości między miastami można odczytać z pliku tekstowego o strukturze przedstawionej poniżej. W pierwszej linii pliku znajdują się wierzchołki grafu oddzielone tabulatorami. W kolejnych liniach znajdują się krawędzie grafu(po jednej krawędzi na linię) w postaci wierzchołek - wierzchołek - wartość. Separatorem w pliku jest tabulator. Przykład pliku przedstawiono poniżej.

0	1	2	3
0	1	100	
0	2	10	
0	3	10	
1	2	20	
1	3	70	
2	3	50	

4.1.2 Losowa generacja mapy

Możliwa jest losowa generacja mapy poprzez wywołanie metody :

```
shared_ptr<Map> Map::ConstructMapOfSize(int mapSize,  
                                         int lowestPossibleDistance = 0, int highestPossibleDistance = 200)
```

Podana metoda konstruuje mapę miast o zadanej wielkości, która jest grafem pełnym. Krawędzie grafu które są odległościami między miastami mają losowe

wartości pomiędzy wartościami zadanymi w parametrach *lowestPossibleDistance* i *highestPossibleDistance*.

4.2 Konfiguracja programu

Konfiguracja programu jest możliwa przy pomocy pliku *app.properties*. Umożliwia on zdefiniowanie następujących parametrów:

- **population_size** - wielkość populacji w każdej iteracji algorytmu,
- **propability_of_crossover** - prawdopodobieństwo krzyżowania,
- **propability_of_mutation** - prawdopodobieństwo mutacji,
- **generation_number** - liczba pokoleń, czyli iteracji algorytmu, w przypadku naszej implementacji jest to warunek stopu.

Przykładowa zawartość pliku konfiguracyjnego została zamieszczona poniżej.

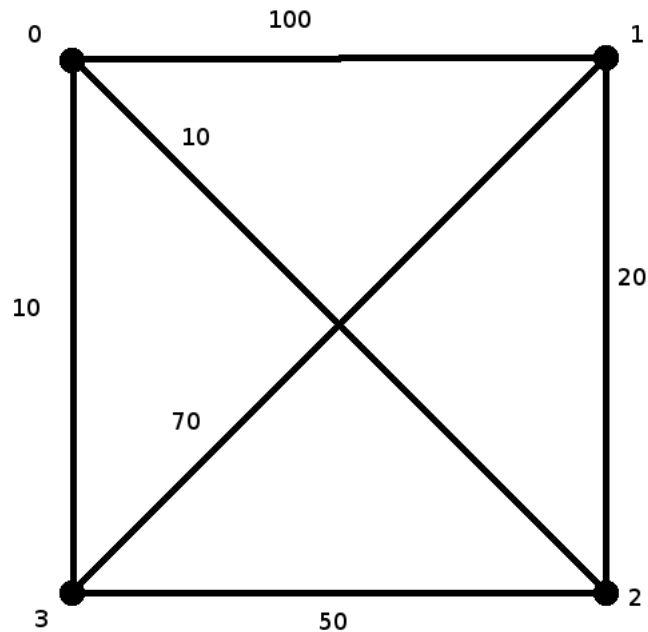
```
population_size=5
propability_of_crossover=0.20
propability_of_mutation=0.01
generation_number=3
```

5 Testy

Przeprowadzono testy z działania programu. Dzielą się one na dwie części. Pierwszą z nich jest Sprawdzenie poprawności algorytmu na niewielkich grafach. Kolejną jest przetestowanie efektywności zrównoleglenia algorytmu.

5.1 Testy poprawności

Jako dane wejściowe sprawdzające poprawność działania algorytmu wykorzystano dwa, małe, tendencyjnie stworzone grafy, w których można z łatwością dokonać weryfikacji działania algorytmu. Schematy podanych grafów przedstawiono poniżej.



Rysunek 1: Graf wejściowy zawierający 4 wierzchołki

Po przetworzeniu pliku wejściowego zawierającego graf 1 Algorytm wyznaczył najkrótszą ścieżkę o wartości 110. Należy pamiętać że ostatnim odcinkiem pokonywanym przez kuriera jest droga z ostatniego odwiedzonego miasta do miasta z którego wyruszył. W tym przypadku jest to krawędź pomiędzy wierzchołkami 1 i 3 o wartości 70. Najważniejsze fragmenty pliku wejściowego zostały przedstawione poniżej.

POPULATION SIZE: 30

```

map size: 4
nodes:0 1 2 3
m[first city: 0 sec city: 1] = 100
m[first city: 0 sec city: 2] = 10
m[first city: 0 sec city: 3] = 10
m[first city: 1 sec city: 2] = 20
m[first city: 1 sec city: 3] = 70
m[first city: 2 sec city: 3] = 50

```

Initial population initialization time: 7154 microseconds

START reproduce

```
START reproduce
0 'st generation production time: 22464 microseconds

START reproduce
1 'st generation production time: 14053 microseconds

START reproduce
2 'st generation production time: 9021 microseconds

START reproduce
3 'st generation production time: 362 microseconds

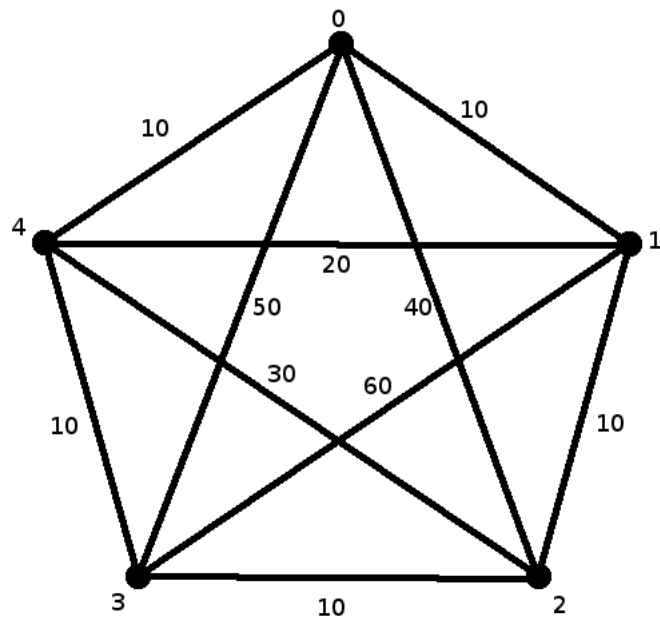
START reproduce
4 'st generation production time: 339 microseconds

START reproduce
5 'st generation production time: 357 microseconds

whole loop time 46596

END: 3->0->2->1

PAth LEN: 110
```



Rysunek 2: Graf wejściowy zawierający 5 wierzchołków

Po przetworzeniu pliku z grafem wejściowym przedstawionego na rysunku 2 program wyznaczył ścieżkę o wartości 50. Najważniejsze fragmenty pliku wejściowego zostały przedstawione poniżej.

```
map size: 5
nodes:0 1 2 3 4
m[first city: 0 sec city: 1] = 10
m[first city: 0 sec city: 2] = 40
m[first city: 0 sec city: 3] = 50
m[first city: 0 sec city: 4] = 10
m[first city: 1 sec city: 2] = 10
m[first city: 1 sec city: 3] = 60
m[first city: 1 sec city: 4] = 20
m[first city: 2 sec city: 3] = 10
m[first city: 2 sec city: 4] = 30
m[first city: 3 sec city: 4] = 10
```

Initial population initialization time: 8892 microseconds

START reproduce

```

START reproduce
0 'st generation production time: 14998 microseconds

START reproduce
1 'st generation production time: 14012 microseconds

START reproduce
2 'st generation production time: 14433 microseconds

START reproduce
3 'st generation production time: 11731 microseconds

START reproduce
4 'st generation production time: 12254 microseconds

START reproduce
5 'st generation production time: 18946 microseconds

whole loop time 86374

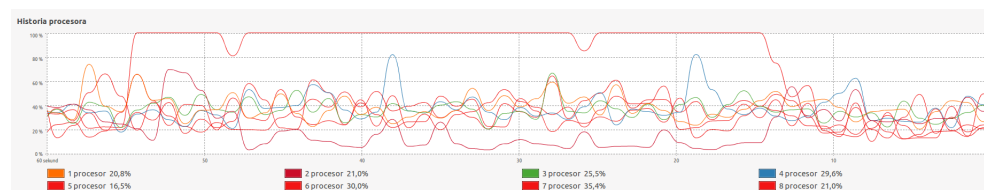
END: 4->0->1->2->3

PAth LEN: 50

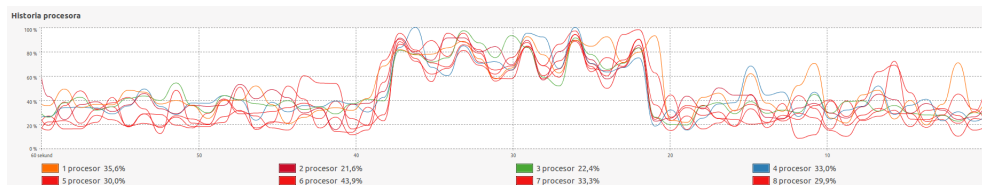
```

5.2 Testy efektywności zrównoleglenia

Kolejnym etapem testów było przetestowanie efektywności zrównoleglenia kodu. Wykorzystanie poszczególnych rdzeni procesora zostało przedstawione na rysunkach poniżej. Jeden z nich przedstawia wykonanie szeregowe programu, drugi zaś wykonanie zrównoległone.



Rysunek 3: Diagram przedstawiający zużycie wątków przy wykonaniu sekwencyjnym



Rysunek 4: Diagram przedstawiający zużycie wątków przy wykonaniu zrównoleglonym

6 Wnioski

W celu zbadania najbardziej pracochłonnych części programu dokonano profilowania aplikacji przy użyciu programu *callgrid*. Operacją, która trwa najdłużej jest krzyżowanie osobników. Dyrektywy *OpenMP* w celu zrównoleglenia pętli zostały dodane w następujących miejscach:

- operacja krzyżowania,
- operacja mutacji,
- obliczanie funkcji przystosowania dla osobników populacji,
- obliczanie prawdopodobieństwa osobnika do mechanizmu selekcji ruletkowej,
- konstruktor populacji - następuje w tym miejscu inicjalne tworzenie listy osobników na podstawie mapy miast na początku działania algorytmu.

Zrównoleglenie algorytmu za pomocą dyrektyw OpenMP powoduje około trzykrotne zmniejszenie czasu wykonania programu. Pliki wynikowe *outSequence.txt* i *outParallel.txt* przedstawiają Wyjście algorytmu w formie sekwencyjnej i zrównoleglonej. Ze względu na ich długość poniżej zostały przedstawione niewielkie, lecz najważniejsze wycinki.

Parallel: `whole_time` = 40832015 microseconds
 Sequential: `whole_time` = 16281865 microseconds

Następują niewielkie wahania współczynnika przyspieszenia, ale generalnie utrzymuje się on na poziomie około 3 (trzykrotne przyspieszenie po użyciu dyrektyw OpenMP)