UCZENIE MASZYNOWE

ZADANIE 1 - SPRAWOZDANIE

Karolina Czerczak & Bartosz Lewandowski

30.03.2022

CZĘŚĆ A

W zadaniu mamy informację o konkursie organizowanym przez NASA, w którym zgłoszony zespół może otrzymać grant na współpracę. Zasady konkursu są następujące:

- każdy członek zgłoszonego zespołu musi mieć przynajmniej jedną publikację na Arxiv GR;
- nie może być w zespole naukowców, którzy mieli wspólną publikację

Kwota grantu jest zależna od ilości osób w zespole – wynosi 100.000$ za każdego członka zespołu.

Dane “Arxiv_GR_collab_network.txt” przedstawiają graf połączeń pomiędzy współpracującymi ze sobą naukowcami, przy czym wszyscy mają publikację na Arxiv GB, zatem pierwszy warunek otrzymania grantu przez zespół jest automatycznie zapewniony. Ponieważ wysokość grantu ściśle zależy od wielkości zespołu, więc problem sprowadza się do znalezienia jak największej liczby naukowców, z których każdych dwóch nigdy nie współpracowało ze sobą. Oznacza to, iż musimy znaleźć najdłuższy antyłańcuch w grafie przedstawionym w pliku z danymi.

Do rozwiązania tego problemu zastosowaliśmy metaheurystykę populacyjną. Algorytm bazuje na losowo wybranej populacji kilku możliwych zespołów naukowców. Podczas kolejnych iteracji algorytmu tworzone są nowe zespoły poprzez krzyżowanie zespołów z populacji, a następnie mutowanie tych nowopowstałych. Później wybierane są najlepsze zespoły, czyli te, które spełniają warunki otrzymania grantu oraz mają największą liczbę naukowców, i uaktualniana jest populacja. Algorytm kończy się po osiągnięciu warunku stopu i podaje najlepszy zespół, który udało mu się znaleźć.

Metody stosowane w algorytmie:

- Metody krzyżowania:
    - metoda *half_of_list()* – dwa zespoły (listy) są przecinane w połowie, a nowy zespół (lista) powstaje poprzez sklejenie pierwszej połowy jednego zespołu oraz drugiej połowy drugiego zespołu. Metoda ta ma jeden parametr – *crossed_number*, który określa ile najlepszych zespołów z populacji ma być skrzyżowanych;
    - metoda *random_cut_place()* – dwa zespoły (listy) są przecinane w losowym miejscu (obydwie w tym samym), a nowy zespół (lista) powstaje poprzez sklejenie początkowej części jednego zespołu oraz końcowej części drugiego zespołu. Metoda ta ma jeden parametr – crossed_number, który określa ile najlepszych zespołów z populacji ma być skrzyżowanych;

- Metody mutowania:
    - metoda *change_individual()* – lista członków danego zespołu jest przedstawiona za pomocą ciągu zer oraz jedynek – zero oznacza, że naukowiec na danej pozycji nie należy do zespołu, zaś jedynka oznacza, że taki naukowiec należy do zespołu. W każdym zespole po krzyżowaniu jest losowo wybierana odpowiednia liczba elementów z listy członków dla tego zespołu oraz zamieniana na wartość przeciwną: 0 jest zamieniane na 1, natomiast 1 jest zamieniane na 0. Metoda ta ma jeden parametr – *number_of_changes,* który odpowiada za ilość zmienianych elementów w liście członków dla każdego zespołu;
    - metoda *change_n_places()* – losujemy konkretną liczbę elementów list członków ze wszystkich zespołów po krzyżowaniu. Zamieniamy wszystkie wylosowane elementy na elementy przeciwne według powyżej opisanej zasady. Metoda ta ma jeden parametr – *number_of_changes*, który odpowiada za liczbę losowanych i zmienianych elementów list członków;
- Metoda selekcji (*selection_method()*) – wybiera najlepsze zespoły (ze względu na wysokość otrzymanego grantu) z tych powstałych po krzyżowaniu oraz mutacji. Liczba wybieranych zespołów jest równa rozmiarze populacji;
- Warunki stopu:
    - warunek *time_limit()* – określa, po jakim czasie algorytm powinien zostać przerwany (działa powyżej 20 sekund, ze względu na początkowe działania algorytmu niezbędne do wykonania części ulepszania i wybierania zespołów). Warunek ten ma jeden parametr – *stop_condition_value*, który definiuje ilość sekund, po upłynięciu których następuje zatrzymanie algorytmu;
    - warunek *iteration_limit()* – określa, po ilu wykonanych iteracjach algorytm powinien zostać przerwany. Warunek ten ma jeden parametr – *stop_condition_value*, który definiuje ilość iteracji, po których algorytm się kończy.

Przetestujemy efektywność algorytmu przy zastosowaniu różnych metod krzyżowania i mutacji, przy różnych warunkach stopu oraz przy różnych wartościach wymienionych wyżej parametrów. Efektywność będziemy badać na podstawie próbek 5 wyników otrzymanych przy takich samych ustawieniach algorytmu.

Krzyżowanie	Mutacja	Warunek stopu	Wynik
half_of_lists(5)	change_individual(50)	time_limit(600)	2002000
half_of_lists(5)	change_n_places(500)	time_limit(600)	2008000
half_of_lists(5)	change_individual(50)	time_limit(300)	2010000
half_of_lists(5)	change_n_places(500)	time_limit(300)	2009000
half_of_lists(10)	change_individual(50)	time_limit(600)	2007000
half_of_lists(10)	change_n_places(500)	time_limit(600)	2008000

half_of_lists(10)	change_individual(50)	time_limit(300)	2007000
half_of_lists(10)	change_n_places(500)	time_limit(300)	2013000
half_of_lists(10)	change_individual(50)	iteration_limit(1000)	2008000
half_of_lists(10)	change_n_places(500)	iteration_limit(1000)	1998000
half_of_lists(10)	change_individual(50)	iteration_limit(500)	2008000
half_of_lists(10)	change_n_places(500)	iteration_limit(500)	2009000
random_cut_place(10)	change_individual(50)	time_limit(600)	2008000
random_cut_place(10)	change_n_places(500)	time_limit(600)	2010000
random_cut_place(10)	change_individual(50)	time_limit(300)	2009000
random_cut_place(10)	change_n_places(500)	time_limit(300)	2004000
random_cut_place(10)	change_individual(50)	iteration_limit(1000)	2007000
random_cut_place(10)	change_n_places(500)	iteration_limit(1000)	2009000
random_cut_place(10)	change_individual(50)	iteration_limit(500)	2005000
random_cut_place(10)	change_n_places(500)	iteration_limit(500)	2001000
half_of_lists(20)	change_individual(1000)	time_limit(1000)	2015000
random_cut_place(5)	change_n_places(100)	time_limit(120)	1991000

Widać, że wyniki są podobne po zmianie metod krzyżowania i mutacji oraz warunku stopu, zatem wybrane metody niewiele się od siebie różnią. Jednak zauważalny jest wynik 2015000, który otrzymaliśmy po zwiększeniu liczby krzyżowanych zespołów, ilości mutacji w każdym zespole oraz czasu działania algorytmu. Widać również, że przy drastycznym zmniejszeniu tych parametrów, wynik jest wyraźnie gorszy, bo tylko 1991000. Wnioskujemy, iż algorytm daje podobne wyniki w każdym z testów, gdyż algorytm startuje od dość dobrego zestawu zespołów, wybierając dużą liczbę naukowców przy zachowaniu warunku, że nikt ze sobą nie współpracował.

CZĘŚĆ B

W części B zadania 1, mamy informację o zabójcy z Long Island, który został zlokalizowany w pewnym miejscu na terenie Stanów Zjednoczonych. Następnie Policja ograniczyła teren, na którym należy przeprowadzić obławę. Polecenie brzmiało, aby rozstawić policjantów tak, by każdą ulicę obserwował co najmniej jeden stróż prawa, przy czym osoba stojąca na skrzyżowaniu jest w stanie obserwować wszystkie ulice do niej dojeżdżające. Z uwagi na ogromne koszta operacji, wyzwanie w tym zadaniu polega na wykorzystaniu jak najmniejszej ilości policjantów. Będzie to zagadnieni typu NP - zupełnego.

Problem polega na minimalnym pokryciu wierzchołkowym grafu, gdzie:

- skrzyżowania są wierzchołkami,
- ulice są krawędziami,
- zbiór policjantów, jest zbiorem wierzchołków pokrycia.

Do dyspozycji otrzymaliśmy dwa pliki (warianty):

- “roadNet_USRoads.txt” - wariant łatwy (129164 wierzchołków i 165435 krawędzi),
- “roadNet-PA.txt” - wariant trudny (1088092 wierzchołków i 3083796 krawędzi).

Do rozwiązania tego zadania zastosowaliśmy metaheurystykę bazującą na przeszukiwaniu lokalnym, tj. Tabu search. Dobór metody był ściśle uwarunkowany możliwością występowania wielu minimów lokalnych, gdzie algorytm TS powinien sprawdzić się najlepiej. Ustalonym wariantem przeszukiwania jest (uproszczona) propozycja z pracy zatytułowanej “Multi-start iterated tabu search for the minimum weight vertex cover problem”. Oczywiście naszym celem optymalizacyjnym będzie dążenie do minimalnej ilości policjantów potrzebnych do obserwacji wszystkich ulic.

Kroki algorytmu to:

1. wygenerowanie losowego rozwiązania (niekoniecznie minimalnego),
2. przeprowadzenie wyszukiwania Tabu, na wygenerowanym rozwiązaniu,
3. porównanie wyników,
4. perturbacja najlepszego wyniku i ponowne jego przeszukanie TS.

Powyższe kroki są wyrażone poniższym pseudokodem:

Algorithm 1 Pseudo-code of MS-ITS for the MWVCP

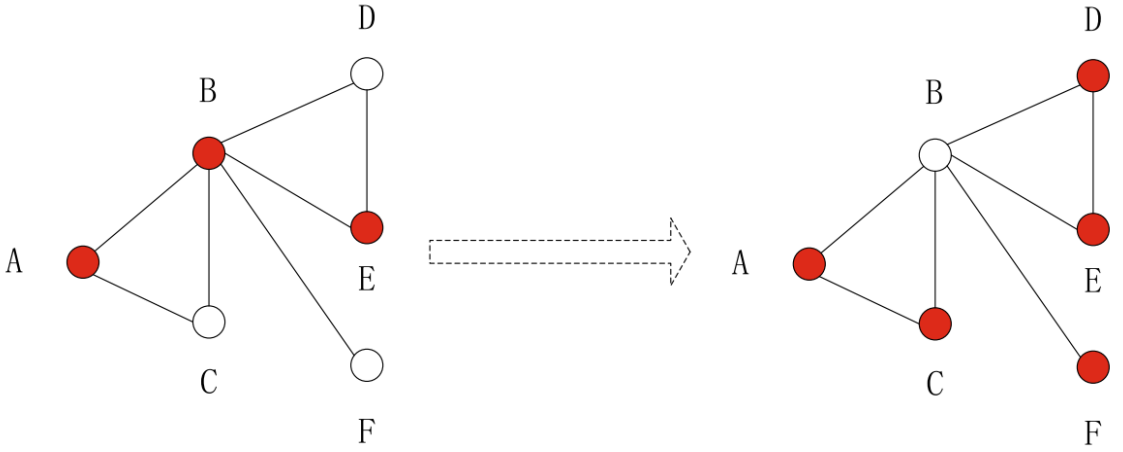
Input: Undirected graph $G(V, E)$, N_{start}, α
Output: The best solution V'_{gbest} found so far
$iter \leftarrow 0$
while $iter < N_{start}$ **do**
  Generate initial solution V' /* See Section 3.2 */
  $no\_improve \leftarrow 0$
  while $no\_improve < \alpha$ **do**
    $V'_{lbest} \leftarrow TabuSearch(V')$ /* See Section 3.3* /
    if V'_{lbest} is better than V'_{gbest} **then**
      $V'_{gbest} \leftarrow V'_{lbest}$
      $no\_improve \leftarrow 0$
    else
      $no\_improve \leftarrow no\_improve + 1$
    end if
    $V' \leftarrow Perturbation(V'_{lbest})$ /* See Section 3.4 */
  end while
  $iter \leftarrow iter + 1$
end while

Funkcje stosowane w algorytmie:

- *GenerateInitialSolution()* – poruszając się po wskazanych krawędziach, w sposób losowy, wybierany jest wierzchołek z “nieodwiedzonej” krawędzi. W ten sposób mamy pewność iż każda z krawędzi jest incydenta do pewnego wierzchołka.
- *MW()* – mamy do dyspozycji dwa zbiory:
  - V’ - zbiór pokrytych wierzchołków,

- V_r(v) - zbiór sąsiadów wierzchołka v, niebędących w zbiorze V'.

Przeprowadzamy "sąsiedzki ruch". Polega on na wskazaniu wierzchołka z V', usunięcie go ze zbioru V', a jego sąsiadów niebędących w V', tj ze zbioru V_r(v), przeniesienie do zbioru V'.



Neighborhood move $MV(B)$

- *TabuSearch()* – wykonujemy kolejne ruchy sąsiedzkie metodą brute force po wierzchołkach danego rozwiązania. Dodatkowo wprowadzamy dwie tabele Tabu:
  - T1 - przechowuje wierzchołki, na których został zastosowany ruch sąsiedzki,
  - T2 - przechowuje zbiór sąsiadów V_r(v) przeniesionych do V'.

  Wykorzystanie tabu polega na wprowadzenie ograniczenia na możliwość wykonania kolejnych sąsiedzkich ruchów, tj. jeśli v w V' oraz liczba V_r(v) będących już w V' jest większa niż n, to jest to ruch tabu - nie wykonujemy.

  Przykład:
  Jeśli zastosujemy przesunięcie MV(v), wierzchołek v zostanie zapisany na pierwszej liście tabu T1, a wierzchołki V_r(v) zostaną zapisane na drugiej liście tabu T2. Wówczas ruch sąsiedni MV(u) zostałby uznany za tabu, jeśli wierzchołek u znajduje się na pierwszej liście tabu T1 i więcej niż n wierzchołków ze zbioru V_r(u) znajduje się na drugiej liście tabu T2.
- *Perturbation()* – usuwamy z zadanego rozwiązania p wierzchołków, a następnie losowo uzupełniamy braki w wierzchołkach.
- *MS_ITS()* – szkielet rozwiązania. Zaimplementowanie algorytmu w oparciu o zaproponowany pseudokod.

Przetestujemy efektywność algorytmu przy zastosowaniu różnych ustawieniach parametrów, gdzie:

➔ graph: dict - graf w postaci słownika
➔ edges: list - lista krawędzi w grafie, w postaci par wierzchołków,
➔ Nstart: int - liczba iteracji przeprowadzania pełnego obiegu,
➔ alpha: int - liczba iteracji przeprowadzania Tabu search wraz z perturbacją,
➔ gamma: int - maksymalna liczba wierzchołków z V_r(v) będących już w V' (warunek T2),

➔ Tau: float - ułamek wierzchołków jakie powinny zostać usunięte z rozwiązania w ramach perturbacji.

Uwaga. *NXfunction()* jest wykorzystaniem wbudowanej funkcji *min_weighted_vertex_cover()* z biblioteki NetworkX do porównania wyników.

```
deg = list(map(lambda x: len(streets[0][x]), streets[0].keys()))
print(f"Średnia ilość sąsiadów wierzchołka wynosi: {sum(deg)/len(deg)}.")
```

```
Średnia ilość sąsiadów wierzchołka wynosi: 2.5616270787525934.
```

```
print(f"Wynik do pobicia: {NXfunction(streets[0])}.")
# parametry
graph = streets[0]
edges = streets[1]
Nstart = 1
alpha = 1
gamma = 2
Tau = 1/3
start = timer()
wynik = MS_ITS(graph, edges, Nstart, alpha, gamma, Tau)
end = timer()
print(f"Otrzymany wynik, to: {len(wynik)}.")
print(f"Czas działania wyniósł {(end-start)/60} minut.")
```

```
Wynik do pobicia: 114156.
Otrzymany wynik, to: 88959.
Czas działania wyniósł 52.50301348333333 minut.
```

```
print(f"Wynik do pobicia: {NXfunction(streets[0])}.")
# parametry
graph = streets[0]
edges = streets[1]
Nstart = 1
alpha = 1
gamma = 3
Tau = 1/3
start = timer()
wynik = MS_ITS(graph, edges, Nstart, alpha, gamma, Tau)
end = timer()
print(f"Otrzymany wynik, to: {len(wynik)}.")
print(f"Czas działania wyniósł {(end-start)/60} minut.")
```

```
Wynik do pobicia: 114156.
Otrzymany wynik, to: 89009.
Czas działania wyniósł 87.22036667166661 minut.
```

```
print(f"Wynik do pobicia: {NXfunction(streets[0])}.")
# parametry
graph = streets[0]
edges = streets[1]
Nstart = 1
alpha = 1
gamma = 2
Tau = 1/2
start = timer()
wynik = MS_ITS(graph, edges, Nstart, alpha, gamma, Tau)
end = timer()
print(f"Otrzymany wynik, to: {len(wynik)}.")
print(f"Czas działania wyniósł {(end-start)/60} minut.")
```

```
Wynik do pobicia: 114156.
Otrzymany wynik, to: 88956.
Czas działania wyniósł 87.29127429833333 minut.
```

```
print(f"Wynik do pobicia: {NXfunction(streets[0])}.")
# parametry
graph = streets[0]
edges = streets[1]
Nstart = 1
alpha = 1
gamma = 3
Tau = 1/2
start = timer()
wynik = MS_ITS(graph, edges, Nstart, alpha, gamma, Tau)
end = timer()
print(f"Otrzymany wynik, to: {len(wynik)}.")
print(f"Czas działania wyniósł {(end-start)/60} minut.")
```

```
Wynik do pobicia: 114156.
Otrzymany wynik, to: 88983.
Czas działania wyniósł 87.12232259166667 minut.
```

```
print(f"Wynik do pobicia: {NXfunction(streets[0])}.")
# parametry
graph = streets[0]
edges = streets[1]
Nstart = 1
alpha = 10
gamma = 2
Tau = 1/3
start = timer()
wynik = MS_ITS(graph, edges, Nstart, alpha, gamma, Tau)
end = timer()
print(f"Otrzymany wynik, to: {len(wynik)}.")
print(f"Czas działania wyniósł {(end-start)/60} minut.")
```

```
Wynik do pobicia: 114156.
Otrzymany wynik, to: 88844.
Czas działania wyniósł 508.917165775 minut.
```

Problem pojawia się przy zwiększeniu parametrów Nstart oraz alpha - zwiększenie ilości przeprowadzanych pętli. Uzyskany rezultat dla alpha = 10, znacząco nie odbiega od poprzednich rozwiązań. Zauważalnie wzrasta czas wykonania algorytmu. Dzieje się tak za sprawą przeprowadzania wielu operacji na wierzchołkach i tym samym niemałego zwiększenia liczby przeprowadzanych iteracji. Z uwagi na niepełne zaimplementowanie propozycji zawartej w wykorzystanej pracy naukowej, wynik oraz czas działania algorytmu można dalej optymalizować/minimalizować.

Finalnie przyjęty rezultat, to 88959 patrolujących skrzyżowania policjantów. Uzyskany wynik można dalej minimalizować, natomiast już w tym momencie jest on lepszy od wyniku wbudowanej funkcji o 25197 policjantów.