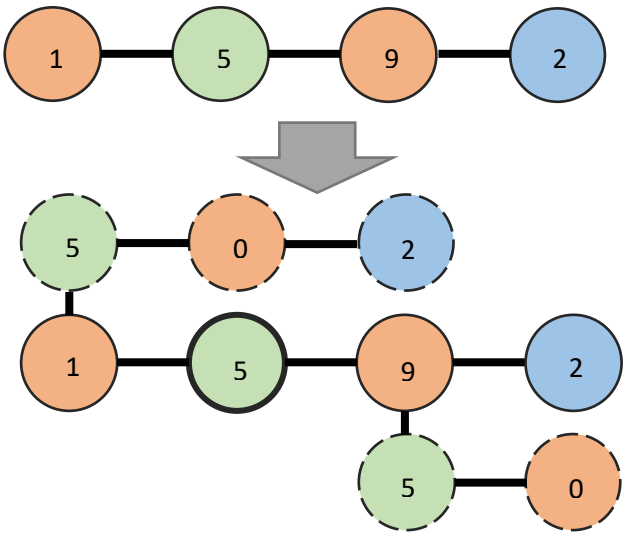
Antoni Charchuła 283713
Semestr 5 18Z Informatyka ISI wydział EiTi
Projekt z przedmiotu Analiza Algorytmów (AAL)

Zadanie AAL-2-LS podróż

Dane są miasta połączone siecią dróg. Dla każdej drogi znany jest czas przejazdu w godzinach. Wjazd do każdego miasta oraz na każdą drogę jest płatny (koszt ustalony oddzielnie dla każdego miasta i drogi). Miasta mogą należeć do grup partnerskich (maksymalnie do jednego partnerstwa) - wjazd do takiego miasta zwalnia z opłat za wjazd do pozostałych miast partnerskich. Każda godzina podróży ma ustalony koszt (niezależny od drogi). Należy znaleźć najtańszą opcję podróży pomiędzy dwoma wyróżnionymi miastami.

Rozwiązanie

Podany w treści problem najłatwiej przedstawić w postaci grafu ważonego. Miasta są wierzchołkami, a drogi krawędziami. Każdy wierzchołek zawiera dane o koszcie wjazdu do miasta, a krawędzie zawierają dane o koszcie wjazdu na drogę oraz jej długości. Graf przechowuje informacje o koszcie jednej godziny podróży, tak żeby była możliwość policzenia kosztu przejazdu przez drogę. Dodatkowo graf zawiera listę zawierającą listy reprezentujące grupy miast partnerskich.
Znalezienie najtańszej opcji podróży pomiędzy dwoma wyróżnionymi miastami sprowadza się do problemu najkrótszej ścieżki w grafie ważonym. Do znajdywania takich ścieżek służy między innymi algorytm Dijkstry, który został użyty w projekcie. Jednakże powstaje pewien problem, polegający na zmienności wag w trakcie przeszukiwania. Po napotkaniu jednego z miast partnerskich, koszty wjazdu do pozostałych miast zmieniają się na 0. Algorytm Dijkstry nie jest odporny na takie zmiany, dlatego należy dokonać pewnych modyfikacji grafu zanim go uruchomimy. W przypadku wejścia do jednego z miast partnerskich należy stworzyć kopie grafu dla każdego z jego sąsiadów, uwzględniając przy tym zmiany dotyczące kosztów wejścia do miast partnerskich. Najłatwiej to przedstawić na przykładzie (Przedstawione zostały dla uproszczenia tylko koszty wjazdu do miasta. Pomarańczowy kolor odpowiada miastom z grupy partnerskiej, zielony kolor to start, a niebieski koniec. W drugim grafie start oznaczony pogrubionym konturem, a kopie wierzchołków konturem przerywanym):



Po takiej modyfikacji grafu bez problemu można szukać najkrótszej ścieżki za pomocą algorytmu Dijkstry.

Generowanie losowego grafu zrealizowano kolejno dodając do grafu wierzchołki, losując przy tym losowego sąsiada. Na koniec losowo dodawana jest określona ilość krawędzi pomiędzy istniejącymi wierzchołkami. Ilość partnerstw i miast w partnerstwach także jest losowana. Losowanie odbywa się w przedziale narzuconym przez użytkownika.

Całość projektu została zrealizowana przy użyciu języka Python. Do wizualizacji grafu użyto biblioteki Numpy.

Opis struktur danych

Dane są reprezentowane za pomocą trzech klas zawartych w pliku Graph.py.

Graf reprezentowany jest za pomocą klasy GraphOfTowns. Zawiera ona słownik z miastami (po podaniu id miasta otrzymujemy jego obiekt), koszt godziny podróży, listę list partnerstw, listę ścieżek do miasta docelowego oraz liczbę dróg .Zawiera między innymi takie funkcje jak: dodawanie nowego partnerstwa, dodawanie miasta do partnerstwa, dodawanie miasta, otrzymywanie miasta czy dodawanie dróg.

Miasto reprezentowane jest za pomocą klasy Town. Zawiera ona identyfikator miasta, drogi wychodzące z miasta, koszt wejścia do miasta, numer partnerstwa, do którego miasto należy, listę odwiedzonych partnerstw, flagę „alreadyExpanded" (opisana poniżej), dystans do miasta docelowego (na początku wszystkie miasta mają dystans równy nieskończoności) oraz wskaźnik na miasto „previous", służący do zapisywania najkrótszej ścieżki.

Droga reprezentowana jest za pomocą klasy Road. Zawiera ona koszt wjazdu na drogę, długość przejazdu przez drogę oraz oba identyfikatory miast, które łączy.

Opis algorytmu

Algorytm poszukujący najkrótszą ścieżkę w grafie opiera się na algorytmie Dijkstry z użyciem kopca (jako kopiec użyto heapq).

Na początku sprawdzane jest czy będą wykonywane kopie grafu – czyli czy graf posiada partnerstwa:

```
if len(graph.listOfTownPartnerships) != 0:
    cleanGraph = copy.deepcopy(graph)
```

Jeśli tak, wykonywana zostaje kopia niezmienionego grafu, której będziemy używali później.

Następnie pobierane zostaje pierwsze miasto i sprawdzane zostaje czy jest ono miastem partnerskim:

```
if startNode.partnershipNumber is not None:
    graph.discountOnPartnershipTownFee(startNode.partnershipNumber)
    startNode.visitedPartnerships.append(startNode.partnershipNumber)
    startNode.assignVisitedPartnerships(graph)
```

Jeśli jest, to wykonujemy dwie istotne operacje:

- *DiscountOnPartnershipTownFee* jest funkcją, która dla danego grafu i partnerstwa, o numerze id przekazanym w parametrze, zmienia *townEnterFee* na wartość równą 0.
- *AssignVisitedPartnerships* jest funkcją, która dla każdego partnerstwa, które nie zostało odwiedzone, przypisuje listę z partnerstwami już odwiedzonymi.

Te dwie operacje gwarantują nam, że koszt wejścia do miast z odwiedzonego partnerstwa będzie równy 0 oraz, że rozszerzając nasz graf, gdy dojdziemy do jakiegoś innego partnerstwa, będziemy mieli zapisane, to że konkretne partnerstwo zostało już odwiedzone.

Następnie „wrzucamy" nasze miasto do kopca:

```
heapq.heappush(queue, startNode)
```

Później wykonywana zostaje główna pętla. Najpierw pobierane zostaje miasto o najmniejszym dystansie:

```
current = heapq.heappop(queue)
```

Następnie sprawdzamy czy osiągnęliśmy miasto finałowe:

```
if current.id == endId:
    graph.finalTownList.append(current)
```

Jeśli tak, to zapisujemy je w liście ze ścieżkami. Później użyjemy go do znalezienie najkrótszej ścieżki.

Następnie wykonywane są operacje:

```
if current.partnershipNumber is not None and current.townEnterFee != 0 and
current.alreadyExpanded is not True:
    copied = copy.deepcopy(cleanGraph)
    current.addChangedGraphToAdjacent(copied)
```

Dany warunek sprawdza czy miasto, które pobraliśmy należy do partnerstwa oraz czy już nie zostało to partnerstwo odwiedzone. Pamiętajmy, że w przypadku odwiedzenia miasta z danego partnerstwa *townEnterFee* jest ustawiane na 0 u wszystkich pozostałych miast z partnerstwa. Odwiedzone miasto nadal posiada swój koszt, jednak ustawiamy na nim flagę *alreadyExpanded* sprawdzaną powyżej.

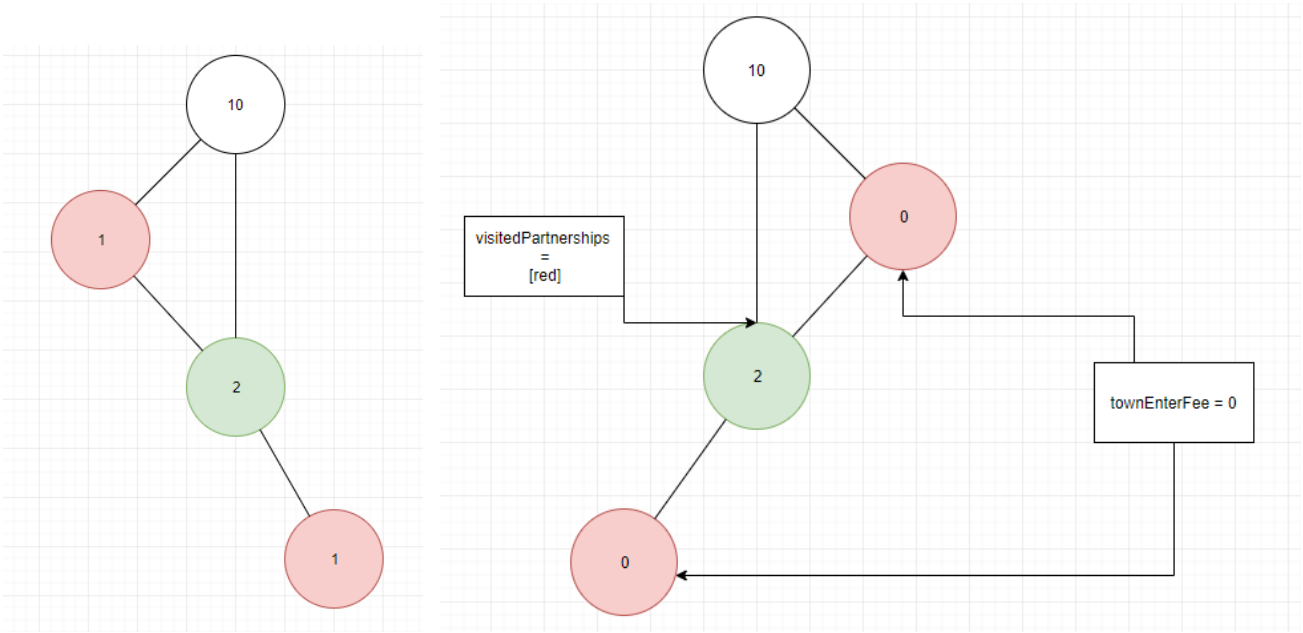
Jeśli warunek jest spełniony to następuje kopia czystego grafu oraz jej zmiana w funkcji *addChangedGraphToAdjacent:*

```
townToSwap = graph.getTown(self.id)

for road in townToSwap.adjacent:
    self.adjacent.append(road)

    if road.firstTown.id == self.id:
        road.firstTown = self
    else:
        road.secondTown = self

self.visitedPartnerships.append(self.partnershipNumber)

for x in self.visitedPartnerships:
    graph.discountOnPartnershipTownFee(x)

graph.setTown(self.id, self)
self.assignVisitedPartnerships(graph)
self.alreadyExpanded = True
```

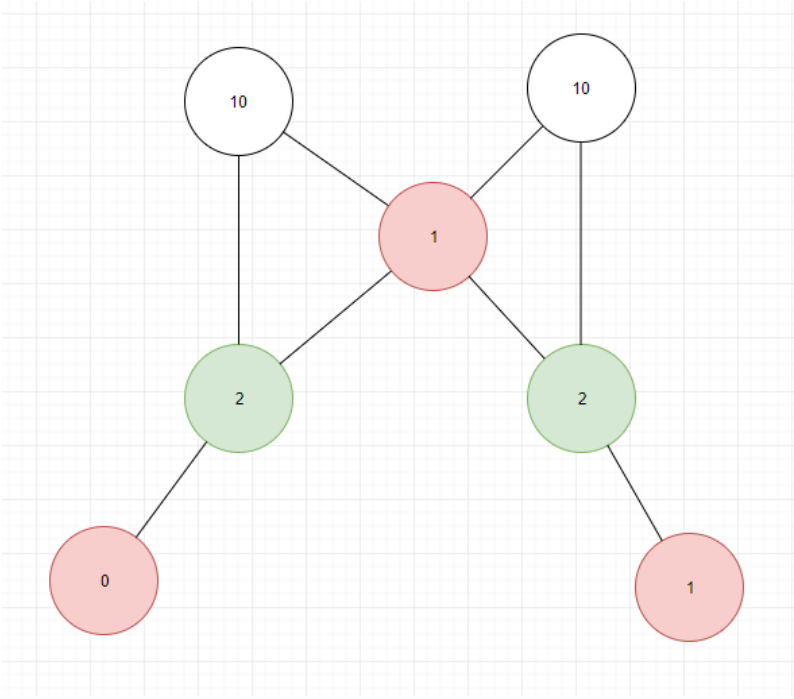
Funkcja ta ma za zadanie, znaleźć to samo miasto w skopiowanym grafie, dodać drogi zawarte w tym skopiowanym mieście do dróg miasta w oryginalnym grafie, zmienić *townEnterFee* na 0 w skopiowanym grafie we wszystkich miastach z list *visitedPartneships* znajdującej się w klasie odwiedzonego miasta. Następnie w skopiowanym grafie usunąć to samo miasto i wstawić w jej miejsce aktualnie odwiedzone miasto. Ustawia temu miastu flagę *alreadyExpanded = true* oraz na koniec za pomocą funkcji

assignVisitedPartnerships, zapisuje informacje o odwiedzonych partnerstwach w miastach z partnerstw jeszcze nieodwiedzonych.

Całą operację można zobaczyć na poniższym obrazku:



(Po lewej oryginał, po prawej kopia ze zmianami, a poniżej wynik końcowy)



Gdy ta operacja zostanie już wykonana (lub warunek nie został spełniony) zostają wykonane czynności typowe dla algorytmu Dijkstry:

```
for road in current.adjacent:

    if road.secondTown.id != current.id:
        nextNode = road.secondTown
    else:
        nextNode = road.firstTown

    newDistance = current.distance + road.hoursOfDriving * graph.costOfOneHourTrip + road.roadEnterFee + nextNode.townEnterFee

    if newDistance < nextNode.distance:
        nextNode.previous = current
        nextNode.distance = newDistance

        if nextNode in queue:
            heapq.heapify(queue)
        else:
            heapq.heappush(queue, nextNode)
```

Czyli dla każdej drogi w odwiedzonym mieście A sprawdzane jest czy istnieje droga do innego miasta B, która z sumowanym dystansem dojścia do miasta A z kosztem dojścia do miasta B jest mniejsza niż dystans zapisany w mieście B. Jeśli dla któregoś miasta zajdzie taka zależność, nadawany jest nowy dystans miastu B i jeśli znajdował się on na stosie, to sortujemy stos, a jeśli nie to dodajemy go do stosu.

Po zakończeniu działania programu rezultat możemy uzyskać za pomocą funkcji *getShortestPath*, która wypisze wszystkie ścieżki o najkrótszej długości pomiędzy zadanymi miastami. Używa ona do tego listy *finalTownList* zawartej w zmiennych grafu. Posługując się wskaźnikami *previous*, zaczynając od końcowego miasta, znajduje całą ścieżkę kończąc na mieście początkowym.

Złożoność obliczeniowa

W zaimplementowanym algorytmie, główny wpływ na czas wykonania, ma ilość kopii grafu, z którą jest bezpośrednio związana ilość partnerstw i miast do nich należących. Dodatkowo trzeba uwzględnić proces działania algorytmu Dijkstry.

Jako, że korzystamy z kopca, znalezienie kolejnego wierzchołka o najmniejszym dystansie ma złożoność równą *logV*, a wykonujemy tych czynności dokładnie *V* razy. Dodatkowo, po każdej zmianie dystansu trzeba przebudować kopiec, co również odbywa się w czasie *logV*, a wykonujemy tą czynność *E* razy.

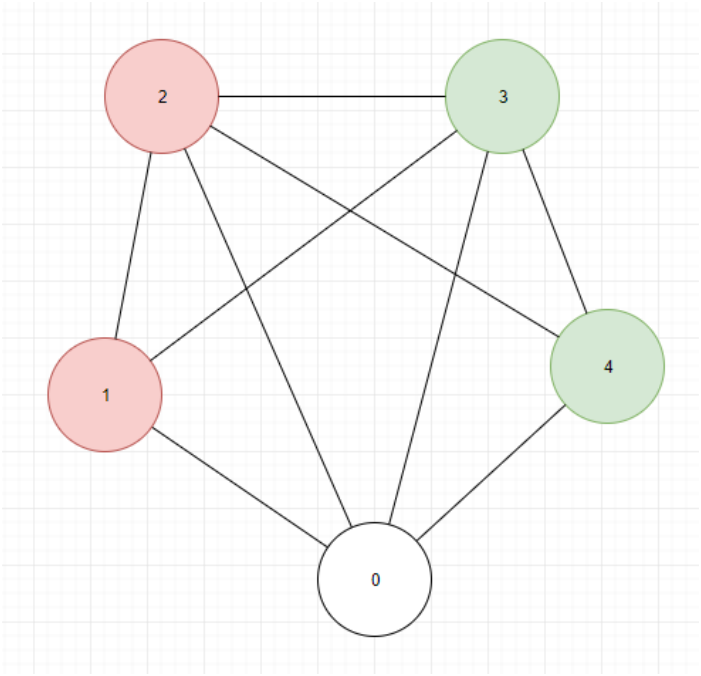
Niestety, ustalenie liczby kopii jest trudniejsze do ustalenia ze względu na różne konfiguracje położeń miast partnerskich. Wobec tego rozpatruję tutaj najgorszą możliwość, czyli graf pełny , oraz że każde partnerstwo posiada tyle samo miast partnerskich.

W takiej konfiguracji ilość wykonanych kopii grafu wynosi :

$T(n) = P * n * (T(n-1) + 1)$ z czego $T(1) = P$

Gdzie n to ilość partnerstw, a P to ilość miast w partnerstwie.

Można ten wzór wytłumaczyć używając do pomocy rysunku:



Miasta czerwone i zielone są miastami z różnych partnerstw, a miasto białe nie należy do żadnego. Zaczynamy w mieście 0.

Aby wykonała się kopia grafu musimy losować z pośród partnerstw, a następnie z pośród miast danego partnerstwa, czyli w naszym przypadku 2 * 2 – ogólnie zapisując n * P. Po przejściu do danego wierzchołka wykonuje się kopia grafu (tutaj pojawia się + 1 we wzorze), a następnie nasz problem zmniejsza się do grafu z liczbą partnerstw mniejszą o jeden. Według tej analogii można ustalić wyżej przedstawiony wzór.

Obliczając dalej złożoność musimy znać złożoność funkcji *deepCopy* użytej do kopiowania grafu. Funkcja ta wykonuje dwie rzeczy:

1. Przechodzi przez swój słownik skopiowanych obiektów i sprawdza czy sprawdzany obiekt był już skopiowany
2. Jeśli nie, to kopiuje obiekt

Szukanie obiektu w najgorszym przypadku ma złożoność „n", jednak zdarza się to bardzo rzadko, dlatego przyjmuje tutaj złożoność najczęstszą, czyli 1.

Kopiowanie obiektów ma złożoność równą 1, dlatego dla całego grafu będzie miało taką ile jest obiektów, czyli: *1 (sam obiekt grafu) + E (liczba dróg) + V (liczba miast)*.

Tworząc ostateczny wzór na złożoność obliczeniową algorytmu, należy pamiętać, że oprócz działania algorytmu Dijkstry w oryginalnym grafie, będzie on działać w każdym innym skopiowanym grafie.

Dodatkowo, jeśli istnieje jakiekolwiek partnerstwo, na początku algorytmu tworzona jest kopia grafu niezmienionego. Ją także trzeba dodać do wzoru.

Wszystkie te oszacowania dają nam ostateczny wzór podany poniżej:

$$T(n) = (P * n * (T(n-1) + 1) + 1)(1 + E + V + E*logV + V*logV) + E*logV + V*logV$$

Problemy implementacyjne i wnioski

Jak widać, głównym czynnikiem zwiększającym czas wykonania algorytmu jest wykonywanie kopii grafu. Możliwe, że istnieje metoda, która tej kopii nie wymaga, a tylko operuje na wierzchołkach oryginalnego grafu, jednak to rozwiązanie łatwo zapewnia nam cofanie się w grafie oraz późniejsze uruchomienie na nim algorytmu Dijkstry.

Kolejnym pomysłem na przyspieszenie działania programu, jest stworzenie osobnej funkcji tworzącej graf specjalnie dedykowanej trybowi trzeciemu, gdzie tworzony jest graf pełny. Aktualnie tworzenie takiego grafu jest wolne, ponieważ przy dodawaniu dodatkowych dróg do grafu, dwa miasta są losowane z pośród wszystkim miast. Wobec tego pod koniec tworzenia grafu, bardzo często zdarza się sytuacja, taka że wylosowane zostają miasta, które już mają między sobą połączenie, co powoduje powtórzenie całej operacji losowania.

Do implementacji użyto języka Python, z powodu chęci samorozwoju (nigdy nie pisałem w tym języku – z tego powodu możliwe niedociągnięcia). Po ukończeniu projektu uważam, że jest to język bardzo prosty do użytku jednak w miarę wolny. Szczególnie w przypadku testów w trybie trzecim, gdzie wymagana jest duża ilość iteracji w trakcie tworzenia grafu. Dodatkowo, pojawiają się problemy z „rozgrzaniem" maszyny, co jest widoczne na początkowych etapach testów. Możliwe, że następnym razem wybrałbym inny język np. C++, który by na pewno znacząco przyspieszył działanie programu, jednak wtedy pojawiłby się problem z wizualizacją grafu, która w Pythonie jest bardzo prosta.