Laboratorium grafowe 4

Projekt „Matematyka dla Ciekawych Świata”,
Tomasz Świerczewski, Jakub Jałowiec

24.05.2021

1 Klonowanie replit'a

Do wykonania zadań związanych z algorytmem Dijikstry potrzebny będzie jakiś ciekawy graf - np. znany nam już graf ścieżek w Tatrach. Graf ten znajdziesz w replit'cie replit'cie MDCS-Lab4. Słownik zawierający wszystkie wierzchołki tego grafu znajdziesz w pliku ***tatry.py***.

2 Grafy ważone

Do implementacji grafów ważonych nie wystarczy nam już sama lista sąsiedztwa, a potrzebujemy parę „wierzchołek docelowy, waga”. Z tego powodu zamiast ze słownika list (jak to robiliśmy do tej pory dla *grafów nieważonych*) skorzystamy ze słownika słowników, które umożliwią nam nadawanie wag krawędziom.

Dla przypomnienia - dotychczas używaliśmy następującej struktury, do implementacji grafów nieważonych:

```
graf = {
  "A" : ["B", "C"],
  "B" : ["A", "C"],
  "C" : ["A", "B"]
}
```

Teraz będziemy używać następującej struktury, aby móc przechowywać również wagi krawędzi:

```
graf = {
  "A" : {
    "B" : 1,
    "C" : 4
  },
  "B" : {
    "A" : 2,
    "C" : 3,
  },
  "C" : {
    "A" : 5,
    "B" : 3
  }
}
```

Od tego momentu będziemy ponadto zawsze zakładać, że grafy są skierowane. W tym przykładzie widać, że np. krawędź od `"A"` do `"B"` ma wagę 1, a od `"B"` do `"A"` ma wagę 2. W dalszej części skryptu będziemy pracować na przykładzie grafu ścieżek w Tatrach, w którym krawędzie łączące dwa wierzchołki mają różną wagę.

Zadanie 2.0.1

Napisz funkcję `dodaj_krawedz(graf, zrodlo, cel, waga)`, która doda do grafu `graf` krawędź od wierzchołka `zrodlo` do wierzchołka `cel` z wagą `waga`.

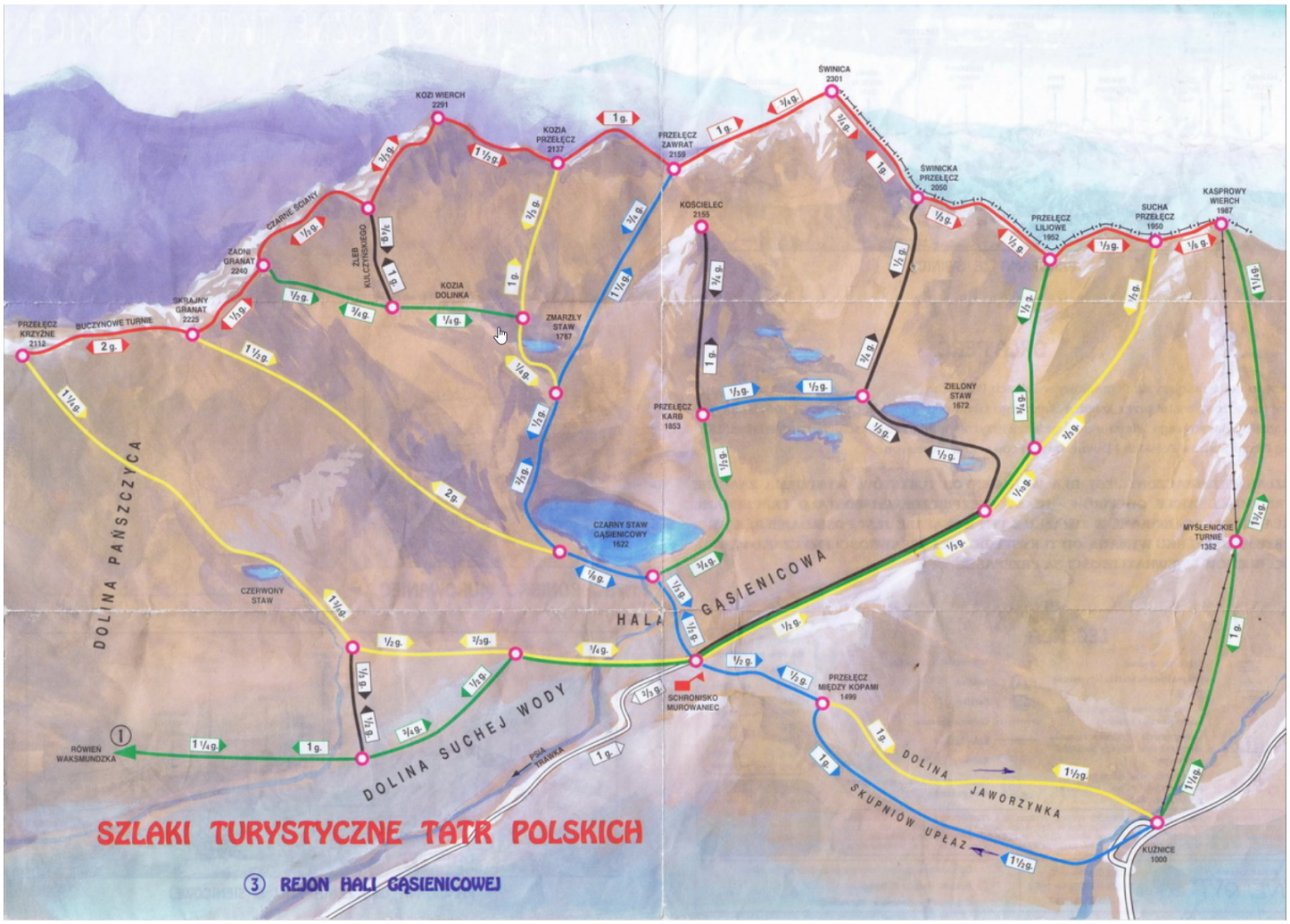
Zadanie 2.0.2

Napisz funkcję `znajdz_najwieksza_waga(graf)`, która znajdzie w grafie krawędź z największą wagą oraz zwróć trójkę `zrodlo, cel, waga`.

3 Znajdowanie najkrótszych ścieżek w grafie

Na wykładzie 6 „Wszerz czy w głąb" mieliście okazję wirtualnie pochodzić po górach. Na przykładzie mapy szlaków turystycznych Tatr Polskich w rejonie Hali Gąsienicowej poznawaliście grafy ważone. Postaramy się wykorzystać ten przykład i stworzyć na jego bazie graf ważony. Mapa tych szlaków jest na Rysunku 1.

Rysunek 1: Mapa szlaków turystycznych Tatr Polskich w rejonie Hali Gąsienicowej



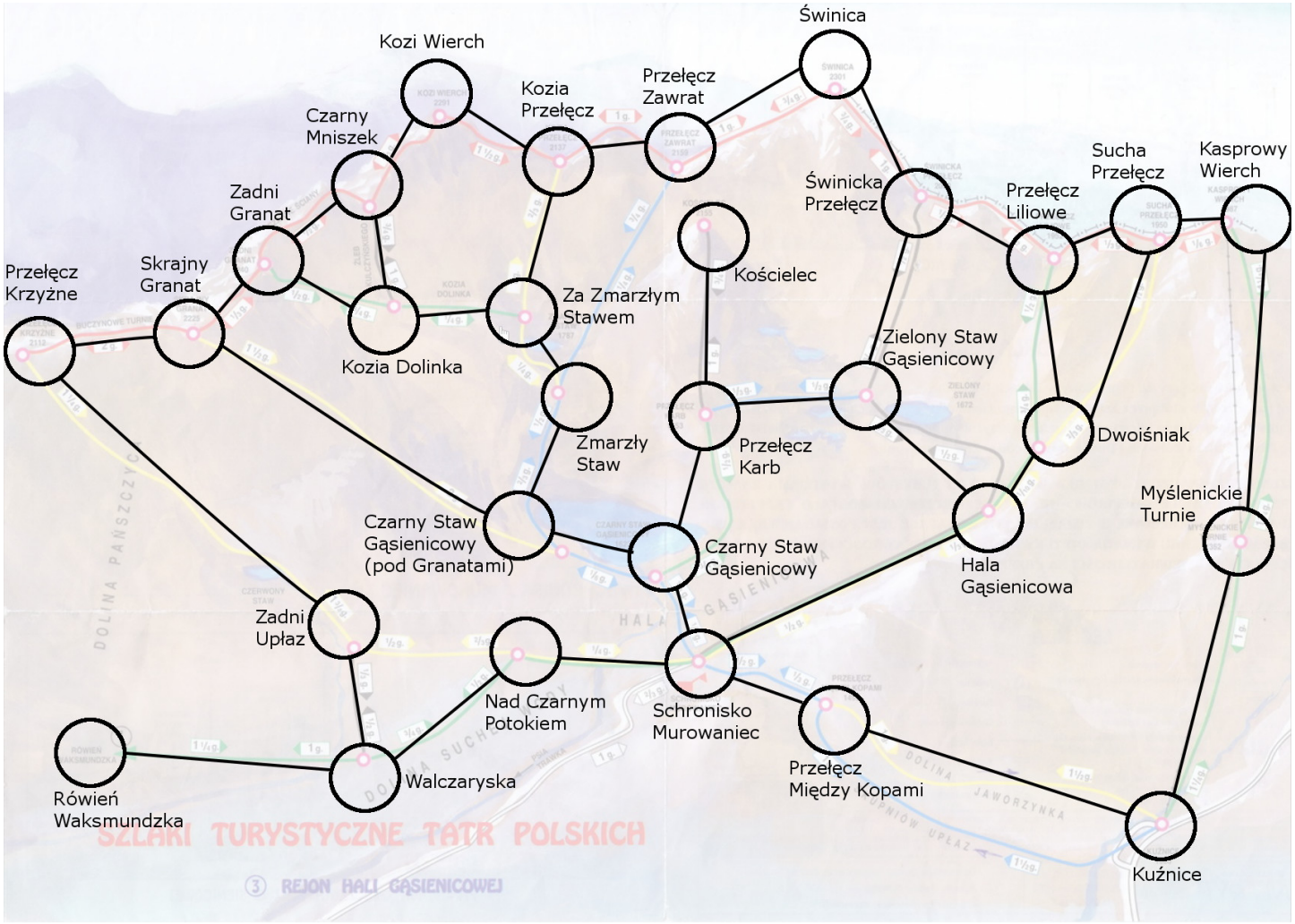
Jest to bardzo dobry przykład na to, że krawędź łącząca oba wierzchołki może mieć różne wagi. Szlaki w obu strony zazwyczaj biegną w tym samym miejscu, a czas wejścia do góry różni się od schodzenia z góry do dołu. Tak na przykład dotarcie z Kuźnic do Kasprowego Wierchu przez Myślenickie Turnie zajmie nam łącznie 3 godziny, a zejście już tylko 2 i 1/4 godziny.

Do Kasprowego Wierchu według tej mapy możemy też dojść na drugi sposób — przez Schronisko Murowaniec. Musimy tym razem wykorzystać kilka szlaków. Przeanalizujmy każdy jego etap. Od Kuźnic do Przełęczy Między Kopami będziemy podchodzić 1 i 1/2 godziny. Od Przełęczy Między Kopami do

Schroniska Murowaniec będziemy schodzić w 1/3 godziny. Od Schroniska do Hali Gąsienicowej koło dolnej stacji wyciągu narciarskiego będziemy podchodzić 1/2 godziny. Następnie już do Dwoiśniaka będziemy podchodzić 1/10 godziny, a do Suchej Przełęczy 2/3 godziny. Ostatnia prosta do Kasprowego Wierchu to już tylko 1/6 godziny. Łącznie daje nam to 3 godziny i 16 minut, o 16 minut dłuższa trasa według tej mapy niż przez Myślenickie Turnie.

W ten sposób intuicyjnie dzięki wykorzystaniu mapy przeanalizowaliśmy najkrótszą drogę. Czasem chcielibyśmy, aby to algorytm wybrał np. najkrótszą drogę. Wielokrotnie my też jesteśmy w stanie znaleźć tę najkrótszą dzięki naszemu mózgowi i oczom, lecz czasami przerasta nas skala problemu. Np. gdybyśmy chcieli drogą lądową dojechać z Warszawy do Singapuru. A co jeśli nie chcielibyśmy najkrótszej, a najtańszej trasy z uwzględnieniem kosztu winiet, średnich kosztów paliwa w danym kraju? Taki problem optymalizacyjny może nas przerosnąć i dlatego czasem warto użyć do tego algorytmów i komputera.

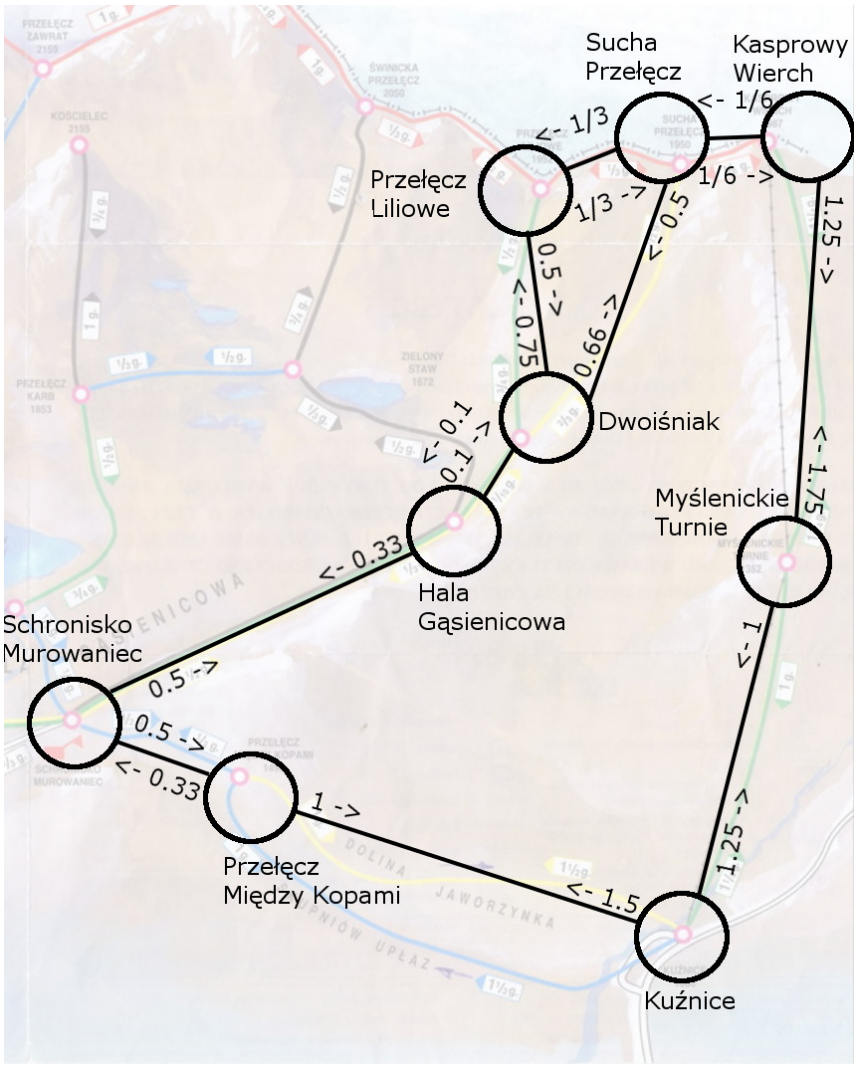
Rysunek 2: Graf powstały przez uproszczenie Rysunku 1. Wagi krawędzi nie zostały zaznaczone, aby nie zaciemniać rysunku - znajdziesz je przygotowane jako Pythonowy słownik w replit'cie MDCS-Lab4.



3.1 Algorytm Dijkstry

Jednym z najprostszych algorytmów jest algorytm Dijkstry, napisany w 1956 i pokazany światu w 1959. Jest on przykładem algorytmu zachłannego. Algorytm zachłanny to taki algorytm, który wybiera w danym momencie rozwiązanie, które najlepiej rokuje. Mimo wszystko takie rozwiązanie może nie być rozwiązaniem optymalnym, gdy spojrzymy na to globalnie, ale jest pewną drogą optymalizacyjną.

Rysunek 3: Wycinek grafu Tatr służący do zobrazowania *Algorytmu Dijkstry*.



Rysunek 3 przedstawia wycinek Tatr, który posłuży nam do zobrazowania algorytmu Dijkstry. Przećwiczmy ten algorytm na jego przykładzie. Algorytm ten oblicza najkrótsze ścieżki do wszystkich wierzchołków od wierzchołka startowego. Skupmy się ponownie na drogach z Kuźnic do Kasprowego Wierchu. Ograniczmy się ponadto do tych wierzchołków, widocznych na Rysunku 3:

```
"Kuźnice"
"Przełęcz Między Kopami"
"Myślenickie Turnie"
"Schronisko Murowaniec"
"Hala Gąsienicowa"
"Dwoiśniak"
"Przełęcz Liliowe"
"Sucha Przełęcz"
"Kasprowy Wierch"
```

Poniżej przedstawiony jest pseudokod algorytmu Dijkstry.

```
Funkcja algorytm_dijkstry(graf, start):
  Dla każdego wierzchołka v w grafie:
    Oznacz odległość do tego wierzchołka d[v]  = nieskończoność.

  Dla wierzchołka startowego oznacz odległość równą zero.

  Dla każdego wierzchołka w grafie:
    Dodaj go do kolejki priorytetowej Q, którą sortujemy po aktualnej
    odległości wierzchołka od wierzchołka startowego.

  Dopóki kolejka Q nie jest pusta:
    Pobierz i usuń z kolejki wierzchołek u o najniższym priorytecie
    (czyli najmniejszej odległości).
    Dla każdego sąsiada v wierzchołka u:
      Jeśli odległość do v przez wierzchołek u jest mniejsza niż
      dotychczasowa, to zastąp ją. Inaczej pisząc jeśli d[u] + w(u, v) < d[v]
      to d[v] = d[u] + w(u, v), gdzie w(u, v) to waga krawędzi od u do v.
```

Tabela 1 pokazuje nam jak przebiegały kolejne kroki algorytmu z wierzchołka startowego "Kuźnice" dla grafu z Rysunku 3. Analizowany w danym kroku algorytmu wierzchołek jest zaznaczony na czerwono. W jednym kroku algorytm oblicza minimalną długość ścieżki z wierzchołka początkowego do bieżącego wierzchołka. Algorytm Dijkstry ma tę cechę. że raz przetworzony wierzchołek nie jest nigdy później ponownie przetwarzany (bo i tak nie da się dojść szybciej). Na szaro zaznaczono te wartości, które już zostały usunięte z kolejki i nie ma do nich szybszej drogi.

Algorytm działa w ten sposób, że kolejno w pewnym sensie sprawdza jak daleko możemy dotrzeć. W pierwszym kroku dowiadujemy się, że do Myślenickich Turni możemy dojść w 1 godzinę i 15 minut, a do Przełęczy Między Kopami w godzinę i 30 minut. Algorytm widzi, że poniżej godziny i 15 minut poza Turniami nigdzie nie dotrzemy, więc analizujemy teraz Myślenickie Turnie. Do nich też już szybciej nie dotrzemy, do inna droga mogłaby tylko prowadzić przez Przełęcz Między Kopami, a tam dojdziemy w dłuższym czasie.

	Krok algorytmu								
Węzeł	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Kuźnice	0	0	0	0	0	0	0	0	0
Myślenickie Turnie	∞	1 1/4	1 1/4	1 1/4	1 1/4	1 1/4	1 1/4	1 1/4	1 1/4
Przełęcz Między Kopami	∞	1 1/2	1 1/2	1 1/2	1 1/2	1 1/2	1 1/2	1 1/2	1 1/2
Schronisko Murowaniec	∞	∞	∞	1 5/6	1 5/6	1 5/6	1 5/6	1 5/6	1 5/6
Hala Gąsienicowa	∞	∞	∞	∞	2 1/3	2 1/3	2 1/3	2 1/3	2 1/3
Dwoiśniak	∞	∞	∞	∞	∞	2 13/30	2 13/30	2 13/30	2 13/30
Kasprowy Wierch	∞	∞	3	3	3	3	3	3	3
Sucha Przełęcz	∞	∞	∞	∞	∞	∞	3 1/10	3 1/10	3 1/10
Przełęcz Liliowe	∞	∞	∞	∞	∞	∞	3 11/60	3 11/60	3 11/60

Tabela 1: Kroki algorytmu Dijkstry. W poszczególnych komórkach tabeli znajdują się obliczone w danym kroku algorytmu długości najkrótszych ścieżek dla danego wierzchołka.

Algorytm w każdym kroku wybiera najmniejszą wartość, która jest w kolejce priorytetowej. Dojście do niej na tym etapie dowolnie inną drogą musi zająć więcej czasu. To co jest nadal w kolejce i ma większy czas ma nadal szansę na zmniejszenie swojej wartości — na przykład gdyby okazało się, że do Kasprowego Wierchu przez Schronisko Murowaniec byłoby 2h 50 minut zamiast 3 godzin. Dopiero gdy np. te 3 godziny

są najmniejszą wartością w kolejce, to wiemy, że dowolna inna droga wymagałaby już więcej niż te 3 godziny.

Jednak w pierwszej kolejności przećwiczymy coś, co nam się przyda w tym algorytmie. Ta funkcja wykorzystuje kolejkę priorytetową. Dotychczas jej nie implementowaliśmy, a implementacja dostępna w Pythonie nie pozwala na wygodną zmianę priorytetu w tej kolejce. Powyżej był znany powszechnie algorytm Dijkstry. Na naszych zajęciach w celu prostszej implementacji trochę go zmodyfikujemy:

```
Funkcja algorytm_dijkstry_mdcs(graf, start):
  Stwórz słownik S taki, że jako klucze zawiera wierzchołki grafu, a jako
  wartości obecny koszt drogi do nich. Dla wierzchołka startowego dystans
  jest równy 0, dla reszty nieskończoność.

  Dopóki słownik S nie jest pusty:
    Pobierz i usuń klucz i wartość ze słownika taką, że jest najmniejsza
    wartość (czyli najmniejszej odległości).
    Dodaj ten klucz i wartość do wyniku (nie da się mniejszym kosztem
    dotrzeć do tego wierzchołka)
    Dla każdego sąsiada v wierzchołka u:
      Jeśli nie jest sąsiad v w wyniku:
        Jeśli odległość do v przez wierzchołek u jest mniejsza niż
        dotychczasowa, to zastąp ją. Inaczej pisząc jeśli d[u] + w(u, v) < d[v]
        to d[v] = d[u] + w(u, v), gdzie w(u, v) to waga krawędzi od u do v.
```

Pomocne może się okazać najpierw rozwiązanie poniższego zadania. Dzięki temu przećwiczycie usuwanie odpowiednich rzeczy ze słownika. W celu usunięciu rekordu ze słownika zastosujcie „`del slownik[klucz]`”. W celu reprezentacji nieskończoności możecie zastosować „`float("inf")`”.

Zadanie 3.1.1

Napisz funkcję `pobierz_minimum(slownik)`, która usunie parę klucz i wartość ze słownika dla takiego klucza, dla którego wartość jest najmniejsza w słowniku. Jeśli jest kilka kluczy, dla których jest osiągnięta minimalna wartość, niech usunie jedną z nich.

Zadanie 3.1.2

Napisz algorytm Dijkstry. Przekazujemy do niego graf oraz wierzchołek startowy, a chcemy uzyskać jako wynik słownik z wartościami najkrótszych dróg do każdego z wierzchołków.

Zadanie 3.1.3 Dodatkowe

Napisz algorytm Kruskala, który oblicza minimalne drzewo rozpinające.
Link do opisu algorytmu: https://pl.wikipedia.org/wiki/Algorytm_Kruskala

4 Praca domowa

Zadanie 4.0.1 — 1 punkt

Napisz funkcję `znajdz_wage(graf, waga)`, która znajdzie w grafie krawędź z daną wagą. Jeśli występuje więcej krawędzi z daną wagą to ma zwrócić jedną z nich, jeśli nie ma żadnej, to ma wypisać odpowiednie powiadomienie na konsolę i zwrócić wartości (-1, -1).

Przykład:

```
> graf = {
>   "a": {"b": 1, "c": 2},
>   "b": {"c": 2},
>   "c": {"a": 3}
> }
> print(znajdz_wage(graf, 2))
('a', 'c')
> print(znajdz_wage(graf, 4))
(-1, -1)
```

Zadanie 4.0.2 — 1 punkt

Napisz funkcję `suma_wag_w_grafie(graf, waga)`, która zsumuje wszystkie wag w grafie.

Przykład:

```
> graf = {
>   "a": {"b": 1, "c": 2},
>   "b": {"c": 2},
>   "c": {"a": 3}
> }
> print(suma_wag_w_grafie(graf))
8
```

Zadanie 4.0.3 — 1 punkt

Napisz funkcję `znajdz_najwieksza_sume_dwoch_krawedzi(graf)`, która znajdzie w grafie dwukrawędziową ścieżkę, mającą największą możliwą wagę. Funkcja ma zwracać czwórkę `zrodlo, posredni, cel, waga`, gdzie:

- pierwsza krawędź wychodzi z wierzchołka `zrodlo` i wchodzi do wierzchołka `posredni`,
- druga krawędź wychodzi z wierzchołka `posredni` i wchodzi do wierzchołka `cel`,
- waga to największa suma wag, znaleziona dla wszystkich możliwych krawędzi `(zrodlo, posredni)` oraz `(posredni, cel)`.

Przykład:

```
> graf = {
>  "a": {"b": 1, "e": 4},
>  "b": {"c": 2, "e": 1},
>  "c": {"d": 3},
>  "d": {},
>  "e": {}
> }
> print(znajdz_najwieksza_sume_dwoch_krawedzi(graf))
('b', 'c', 'd', 5)
```

Zadanie 4.0.4 — 1 punkt

Napisz funkcję, która obliczy czas przejścia Orlej Perci od Przełęczy Zawrat do przełęczy Krzyżne. Posłuż się grafem Tatr dostępnym na *repl.it'cie*.
Przykład:

```
> tatry = {
>   ...
> }
> print(najkrotsza_sciezka(tatry, "Przełęcz Zawrat", "Przełęcz Krzyżne"))
('b', 'c', 'd', 5)
```

Zadanie 4.0.5 — 2 punkty

Napisz funkcję, która sprawdzi, czy graf ma ujemną pętlę.

Zadanie 4.0.6 — 2 punkty

Napisz kolejną wersję algorytmu Dijkstry, ale taką, która zwróci nam najmniejszą odległość dla każdej pary wierzchołków w grafie.

Zadanie 4.0.7 — 3 punkty

Załóżmy, że jesteśmy turystą, który chce spędzić jak najwięcej czasu w górach. Napisz funkcję, która znajdzie taką ścieżkę startującą z Kuźnic, że zajmie jak najwięcej czasu i wrócisz w miejsce startu, ale nie odwiedzając żadnego miejsca więcej niż raz (nie licząc Kuźnic, do których musisz wrócić). Nie jest to ani cykl Eulera, ani Hamiltona, ponieważ nie musisz odwiedzać każdego miejsca.