AiSD - raport z zadania 3.

Antoni Szustakowski

Grudzień 2022

Spis treści

1 Treść zadania 3. - Gra w pomosty

Dane są dwie liczby naturalne: N i M, $N, M \leqslant 1000$. Należy utworzyć ciąg liczb naturalnych $(a_0, a_i, \ldots, a_k)$ taki, że:

- $a_0 = N$ i $a_k = M$
- dla $i \geqslant 0, a_{i+1} = a_i + 3$ lub $a_{i+1} = 3a_i$ lub $a_{i+1} = \frac{1}{2}a_i$ (o ile a_i jest liczbą parzystą).
- $a_i \leqslant 1000, i = 0, \ldots, k$.

Celem gry jest odpowiedź na pytanie, czy dla dowolnych liczb N i M istnieje taki ciąg, a jeśli tak $-$ podać ciąg o najmniejszej możliwej długości (jeśli jest kilka takich ciągów, podać dowolny z nich).

2 Wstęp

W ramach zadania projektowego 3. rozwiązano problem znajdowania najkrótszego ciągu liczb naturalnych, spełniającego warunki z treści zadania. W tym celu zaimplementowano grafowy algorytm Dijkstry w języku Python.

Zaimplementowane rozwiązanie porównano z biblioteką *networkx* zaimplementowaną w języku Python. Testy wykazały pełną poprawność programu, zarówno w znajdowaniu najkrótszego ciągu, jak i zwracaniu informacji o braku ciągu pomiędzy dwoma liczbami spełniającego warunki zadania.

3 Grafowe przedstawienie problemu

3.1 Opis

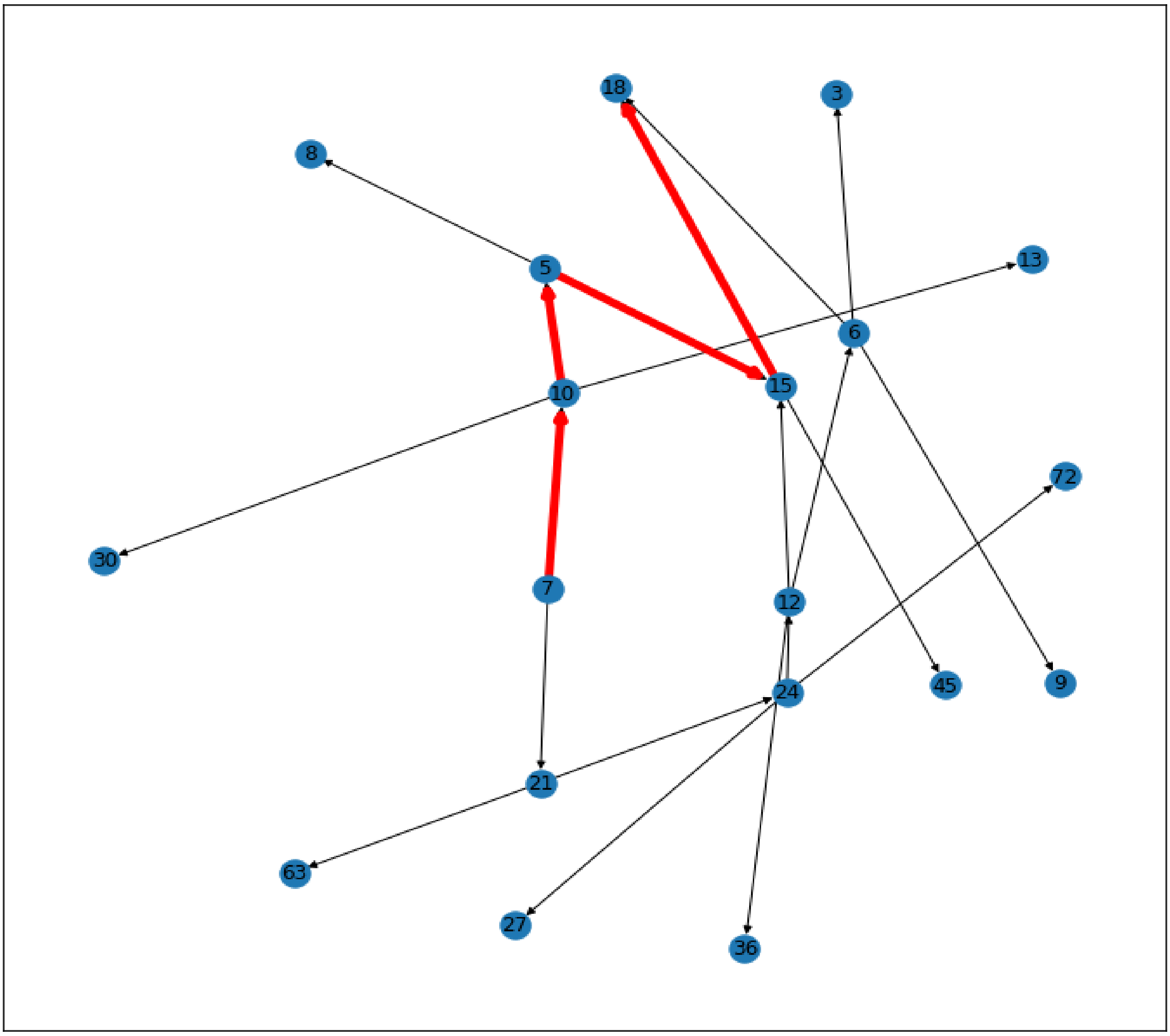
Poszukiwanie najkrótszego ciągu pomiędzy dwiema zadanymi liczbami naturalnymi jest równoważne z poszukiwaniem najkrótszej ścieżki w grafie o wierzchołkach będących liczbami naturalnymi, w którym:

$$(v, w) \in E \iff w = v + 3 \vee w = 3v \vee w = \frac{1}{2}v, v, w \in \mathbb{N}$$

3.2 Przykład

Zilustrujmy problem na przykładzie dwóch wierzchołków podanych w treści zadania: 7 oraz 18. Ścieżki spełniające warunki zadania to m.in.:

- $[7, 10, 5, 15, 18]$
- $[7, 21, 24, 12, 6, 18]$



Rysunek 1: Wykres prezentujący fragment grafu, wraz z zaznaczoną ścieżką od 7 do 18.

Celem programu będzie wyznaczanie ścieżki oznaczonej kolorem czerwonym pomiędzy dowolnymi dwoma wierzchołkami v, w (o ile $v, w \leqslant 1000$) lub wskazywanie, że danej ściezki nie ma.

4 Rozwiązanie problemu

4.1 Tworzenie grafu

Rozwiązanie zadania zaczynamy od stworzenia funkcji odległości, która będzie odpowiadała połączeniu wierzchołków w grafie.

$$l(v,w) = \begin{cases} 1, & w = v+3 \lor w = 3v \lor w = \frac{1}{2}v \\ \infty, & wpp. \end{cases}, v,w \leqslant 1000, v,w \in \mathbb{N}$$

4.2 Uwagi do konstrukcji grafu

Zauważmy, że tak skonstruowany digraf nie jest spójny. Chociażby z wierzchołka 999 nie możemy pójść dalej w ścieżce, z uwagi na ograniczenie górne przez liczbę 1000. Natomiast istnieje też wiele innych par wierzchołków, które nie mogą być połączone ścieżką w tym grafie.

4.3 Rozwiązanie poprzez algorytm Dijkstry

Do rozwiązania problemu zaimplementowano algorytm Dijkstry, służący znajdowaniu najkrótszej ścieżki w zadanym grafie ważonym. W przypadku rozważanego zadania, wszystkie krawędzie mają tę samą wagę równą 1. Zacytujmy ideę algorytmu z wykładu (1):

1. *Zbiór wierzchołków V dzielimy na: zbiór wierzchołków odwiedzonych S i zbiór wierzchołków jeszcze nieodwiedzonych NS. Początkowo $S = \{s\}$.*
2. *algorytm przebiega w $n-1$ fazach, $|V| = n$. W każdej fazie wyznaczamy wierzchołek $w \in NS$, którego odległość od s, spośród wszystkich wierzchołków w NS, jest minimalna. Taka najkrótsza ścieżka od s do w prowadzi przez wierzchołki tylko ze zbioru S (tzw. ścieżka specjalna).*
3. $S := S \cup \{w\}$.

4.4 Rozwiązania techniczne

Program używa kilku zmiennych pomocniczych: dist - aktualnie obliczona minimalna odległość od źródła, prev - poprzedni wierzchołek na ścieżce, path - ścieżka, j - liczba kroków, mindist - minimalny dystans od źródła spośród wszystkich wierzchołków nieodwiedzonych. Ideę list dist oraz prev ponownie zacytujmy z wykładu (1):

Niech w i-tym kroku wyznaczony został wierzchołek $w \in NS$. Teraz dla wszystkich pozostałych wierzchołków v ze zbioru NS należy (być może) uaktualnić ich minimalną odległość od źródła. Może okazać się, że poprzednio określona ścieżka od s do v jest dłuższa niż ścieżka od s do w i od w do v (tj. ścieżka wiodąca przez wyznaczony wierzchołek w. Jeśli tak jest, to dla wierzchołka v modyfikujemy jego aktualnie minimalną odległość od źródla (pole dist), a wierzchołek w staje się poprzednikiem na tej minimalnej ścieżce (pole prev).

4.5 Algorytm

Niech s oznacza wierzchołek początkowy (start), natomiast e oznacza wierzchołek końcowy (end).

1. **begin**
2.   dist:=[], prev:=[0 **for** $i \in V$], path:=[];
3.   **forall** $w \in V - S$ **do**
4.   **if** $dist[w] < \infty$ **then** $prev[w] := s$; **fi**;
5.   **od**;
6.   **forall** $v \in V$ **do** $dist[v] := l(s, v)$; **od**;
7. $S := \{s\}$
8.   **for** $j := 1$ **to** 999 **do**
9. $mindist := \infty$;
10.   **forall** $w \in V - S$ **do**

11. **if** $dist[w] < mindist$ then $v := w$; $mindist := dist[w]$; **fi**;

12. **od**;

13. **if** $v = e$ **then break**; **fi**;

14. $S := S \cup \{v\}$

15. **forall** $w \in V - S$ **do**

16. **if** $dist[w] > mindist + l(v, w)$ **then**

17. $dist[w] := mindist + l(v, w); prev[w] := v$;

18. **fi**;

19. **od**;

20. **od**;

21. Insert(path, e);

22. **while** $prev[e] \neq 0$ **do**

23. $e := prev[e]$; Insert(path, e); **od**;

24. **if** $s \in path$ **then return** path;

25. **else**

26. **return** *Brak ścieżki od* s *do* e; **fi**;

27. **end**

4.6 Opis algorytmu

Linijka 2. tworzy zmienne pomocnicze: dist, prev oraz path. Lista prev, która odpowiada za poprzedników danego wierzchołka na ścieżce, ma w sobie same zera, w celu ułatwienia dodawania wierzchołków do ostatecznej listy path.

W linijce 3. oraz 4. startujemy od 1. iteracji. Wszystkie wierzchołki, do których możemy pójść z s mają poprzednika s.

Linijki 6-20 stanowią główny komponent algorytmu Dijkstry opisanego w podrozdziale 4.3. Jedyną różnicę stanowi linijka 8.: z uwagi na niespójność zadanego digrafu pętlę **while** $S \neq V$ **do** należy zamienić na pętlę **for** uwzględniającą maksymalną liczbę kroków w ścieżce, czyli 999.

W linijce 21. dodajemy końcowy wierzchołek do path. Natomiast kolejne wierzchołki dodajemy do ścieżki z wykorzystaniem początkowego zadania listy prev. prev[s] nie zostało nigdy zaktualizowane, zatem dopóki nie osiągniemy 0 wśród listy prev, dodajemy wierzchołki do ścieżki path, podmieniając wierzchołek końcowy na jego poprzednik na ścieżce.

Linijki 24-26 sprawdzają, czy w ścieżce znalazł się wierzchołek startowy, jeśli tak - zwracana jest lista path, jeśli nie - oznacza to, że nie ma ścieżki od s do e i taka informacja jest zwracana.

4.7 Złożoność algorytmu

4.7.1 Złożoność obliczeniowa

W celu wyznaczenia złożoności obliczeniowej algorytmu skorzystam z obliczeń z wykładu (1), niech $n := |V|$:

- W linijce 2. stworzenie listy prev wymaga dokładnie n kroków, koszt stworzenia pozostałych list to $2C$, łącznie $Cn + 2C$.
- Pętla z linijek 3-5 obraca się dokładnie $n - 1$ razy, łącznie ze sprawdzeniem warunku **if** koszt to $C(n - 1)$.
- Koszt linijek 6-20 został wyznaczony na wykładzie i wynosi $C(n - 1)^2$.
- Linijka 21. to stały koszt operacji.
- Koszt pętli z linijek 22-23 wynosi maksymalnie $2C(n - 1)$ - długość maksymalnej ścieżki to $n - 1$, natomiast minimalnie $2C$ - dla ścieżki długości 1.
- Operacje z linijek 24-26 mają koszt stały.

Zatem ostatecznie:

$$T(n) \leqslant Cn+2C+C(n-1)+C(n-1)^2+C+2C(n-1) = (n-1)^2+4Cn \leqslant Cn^2+4Cn^2 = C_1n^2$$

$$T(n) \geqslant Cn + 2C + C(n - 1) + C(n - 1)^2 + C = 2Cn + C + Cn^2 - 2Cn + C = Cn^2 + 2C \geqslant Cn^2$$

Czyli:

$$T(n) = \Theta(n^2)$$

4.7.2 Złożoność pamięciowa

Program korzysta z 3 list: dist, prev oraz path i zapamiętuje liczbę kroków j. Zatem w pamięci przechowujemy obiekty o maksymalnej łącznej długości $3n + 1$, natomiast minimalnej $2n + 2$, z uwagi na różną możliwą długość listy path (od 1 do 1000). Zatem złożoność pamięciowa jest liniowa względem danych.

4.8 Obsługa błędów

Jeżeli podane liczby N, M nie spełnią warunków zadania, tzn. nie będą liczbami naturalnymi mniejszymi równymi 1000, program wskaże to błędem z odpowiedzią, na czym polega dany błąd.

4.9 Uwagi do programu wykonywalnego

Program wykonywalny nie ma żadnej interakcji z użytkownikiem, liczby N, M są losowane oraz automatycznie zwracana jest ścieżka pomiędzy N i M bądź informacja o braku ścieżki. Proces powtarzany jest dopóki użytkownik nie zakończy działania programu.

5 Testy

Poprawność programu sprawdzono z wynikami funkcji *shortest_path* z pakietu *networkx*. W niektórych przypadkach istnieje więcej niż 1 ścieżka o minimalnej długości, zatem testy przeprowadzono pod kątem długości ścieżki oraz zwracania informacji dotyczącej braku ścieżki pomiędzy zadanymi wierzchołkami. Funkcja *shortest_path* zwraca błąd w przypadku braku ścieżki, zatem testy przeprowadzono w pętli działającej dopóki nie zostanie zwrócony błąd. Liczby N, M były losowane zgodnie z warunkami zadania. Poniżej prezentujemy niektóre z wyników.

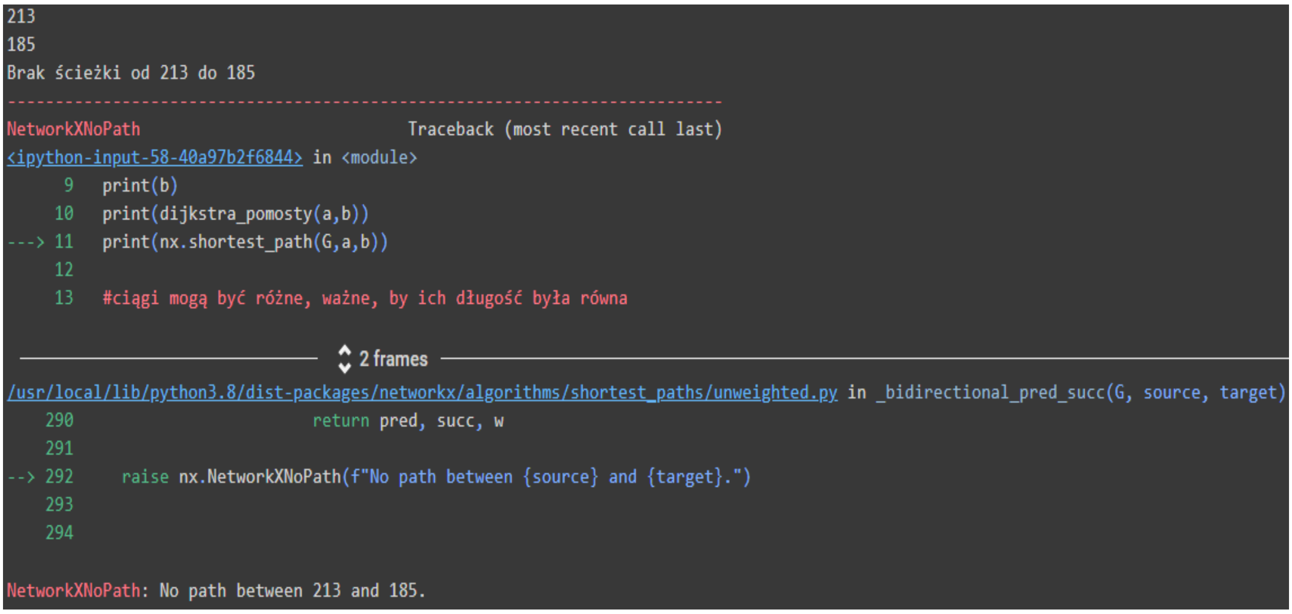
```
[7, 10, 5, 15, 18]
[7, 10, 5, 15, 18]
5
5
```

Rysunek 2: Porównanie programu z funkcją *shortest_path* dla wierzchołków 7 i 18.

```
590
947
[590, 593, 596, 599, 602, 605, 608, 611, 614, 617, 620, 623, 626, 629, 632, 635, 638, 641,
[590, 593, 596, 599, 602, 605, 608, 611, 614, 617, 620, 623, 626, 629, 632, 635, 638, 641,
True
877
807
[877, 880, 440, 220, 110, 55, 58, 29, 87, 261, 264, 267, 801, 804, 807]
[877, 880, 440, 220, 110, 55, 58, 174, 522, 261, 264, 267, 801, 804, 807]
True
826
447
[826, 413, 416, 208, 104, 52, 26, 13, 16, 48, 144, 147, 441, 444, 447]
[826, 413, 416, 208, 104, 52, 26, 13, 16, 48, 144, 147, 441, 444, 447]
True
646
40
[646, 323, 326, 163, 166, 83, 86, 43, 46, 23, 26, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40]
[646, 323, 326, 163, 166, 83, 86, 43, 46, 23, 26, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40]
True
949
558
[949, 952, 476, 238, 119, 122, 61, 183, 186, 558]
[949, 952, 476, 238, 119, 122, 366, 183, 186, 558]
True
391
994
[391, 394, 397, 400, 403, 406, 409, 412, 415, 418, 421, 424, 427, 430, 433, 436, 439, 442,
[391, 394, 397, 400, 403, 406, 409, 412, 415, 418, 421, 424, 427, 430, 433, 436, 439, 442,
True
```

Rysunek 3: Porównanie programu z funkcją *shortest_path* dla losowych wierzchołków, pomiędzy którymi istnieje ścieżka.

Zauważmy, że w przypadku par wierzchołków 877, 807 oraz 949, 558 zwracane są istotnie różne ścieżki, natomiast ich długość jest taka sama. Wynika to najprawdopodobniej z rozpatrywania przez program wierzchołków w kolejności rosnącej, natomiast funkcja *shortest_path* jest zapewne inaczej zaimplementowana.



```
213
185
Brak ścieżki od 213 do 185
---------------------------------------------------------------------------
NetworkXNoPath                            Traceback (most recent call last)
<ipython-input-58-40a97b2f6844> in <module>
      9 print(b)
     10 print(dijkstra_pomosty(a,b))
---> 11 print(nx.shortest_path(G,a,b))
     12 
     13 #ciągi mogą być różne, ważne, by ich długość była równa

2 frames
/usr/local/lib/python3.8/dist-packages/networkx/algorithms/shortest_paths/unweighted.py in _bidirectional_pred_succ(G, source, target)
    290                     return pred, succ, w
    291 
--> 292     raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
    293 
    294 

NetworkXNoPath: No path between 213 and 185.
```

Rysunek 4: Porównanie programu z funkcją *shortest_path* dla losowych wierzchołków, pomiędzy którymi nie istnieje ścieżka.

Dla losowo wybranych wierzchołków, między którymi nie istnieje ścieżka, program zwraca poprawną informację, zgodną z funkcją *shortest_path*. Jedyna różnica polega na sposobie zwracania informacji; program zwraca tekst, natomiast funkcja *shortest_path* - błąd.

```
#obsługa błędów wartości
print(dijkstra_pomosty(1.6,17))

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-0bfac8d8b4cb> in <module>
      1 #obsługa błędów wartości
----> 2 print(dijkstra_pomosty(1.6,170))

<ipython-input-3-bf7a72cbf715> in dijkstra_pomosty(s, e)
     10
     11     if type(s)!=int or type(e)!=int:
---> 12         raise ValueError("Należy podać liczby naturalne mniejsze od 1000.")
     13
     14     if s>=1000 or e>1000 or s<0 or e<0:

ValueError: Należy podać liczby naturalne mniejsze od 1000.
```

Rysunek 5: Wynik programu, w przypadku podania danych niespełniających warunków zadania.

```
#obsługa błędów wartości
print(dijkstra_pomosty(3,-17))

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-63-2f6dd17cbe25> in <module>
      1 #obsługa błędów wartości
----> 2 print(dijkstra_pomosty(3,-17))

<ipython-input-51-e68d9f256d56> in dijkstra_pomosty(s, e)
     13
     14     if s>=1000 or e>1000 or s<0 or e<0:
---> 15         raise ValueError("Należy podać liczby naturalne mniejsze od 1000.")
     16
     17

ValueError: Należy podać liczby naturalne mniejsze od 1000.
```

Rysunek 6: Wynik programu, w przypadku podania danych niespełniających warunków zadania.

6 Wnioski

Program oparty na algorytmie Dijkstry zwraca poprawny wynik dla dowolnej pary liczb naturalnych mniejszych od 1001. W przypadku istnienia ścieżki zwracana jest najkrótsza, natomiast w przypadku braku ścieżki między zadanymi wierzchołkami, tak informacja jest podawana. Poprawność potwierdzono z funkcją *shortest_path* z pakietu *networtkx*. Program radzi sobie również z obsługą błędów wartości, zwracając odpowiednią informację użytkownikowi. Złożoność pamięciowa jest linowa, natomiast złożoność obliczeniowa wynosi $\Theta(n^2)$ - czyli potrzebne modyfikacje podstawowego algorytmu Dijkstry nie zmieniły złożoności obliczeniowej.

Literatura

[1] A. M. Radzikowska, *Wykład 4. z przedmiotu Algorytmy i Struktury Danych.*