SPRAWOZDANIE - LISTA 7

Małgorzata Kowalczyk

Kamil Kowalski

01.02.2022

Zadanie 1

W tym zadaniu naszym celem było stworzenie klasy `class Graph` . Skorzystaliśmy z klasy podanej na wykładzie, odpowiednio ją modyfikując. Dodaliśmy metodę `def get_edges(self)` , która zwraca nam listę wszystkich krawędzi w grafie.

In [2]:
```
g1 = Graph()
for i in range(6):
    g1.add_vertex(i)
g1.add_edge(0,1,5)
g1.add_edge(0,5,2)
g1.add_edge(1,2,4)
g1.add_edge(2,3,9)
g1.add_edge(3,4,7)
g1.add_edge(3,5,3)
g1.add_edge(4,0,1)
g1.add_edge(5,4,8)
g1.add_edge(5,2,1)
g1.get_edges()
```

Out[2]:
```
[(0, 1), (0, 5), (1, 2), (2, 3), (3, 4), (3, 5), (4, 0), (5, 4), (5, 2)]
```

In [3]:
```
g2 = Graph()
for i in range(9):
    g2.add_vertex(i)
g2.add_edge(0, 3, 5)
g2.add_edge(1, 3, 2)
g2.add_edge(2, 3, 4)
g2.add_edge(3, 4, 9)
g2.add_edge(3, 6, 7)
g2.add_edge(4, 8, 7)
g2.add_edge(5, 6, 3)
g2.add_edge(6, 7, 1)
g2.add_edge(7, 8, 1)
g2.add_edge(7, 8, 1)
g2.get_edges()
```

Out[3]:
```
[(0, 3), (1, 3), (2, 3), (3, 4), (3, 6), (4, 8), (5, 6), (6, 7), (7, 8)]
```

Zadanie 2

Następnie dodaliśmy do powyższej klasy metodę generującą reprezentację grafu w języku **dot**. Do przedstawienia wyniku na rysunku posłużymy się stroną internetową www.webgraphviz.com. Wystarczy, że to, co otrzymamy dzięki naszej funkcji `def generate_digraph(self)` przekopiujemy na wspomnianą wcześniej stronę.

Metoda `def generate_digraph(self)` przechodzi po kolejnych wierzchołkach grafu i dołącza do stringa str wszystkie połączenia.

In [4]:
```
print(g1.generate_digraph())
```

```
digraph G { "0" -> "1"; "0" -> "5"; "1" -> "2"; "2" -> "3"; "3" -> "4"; "3" -> "5"; "4" -> "0"; "5" -> "4";
"5" -> "2"; }
```

WebGraphviz is Graphviz in the Browser

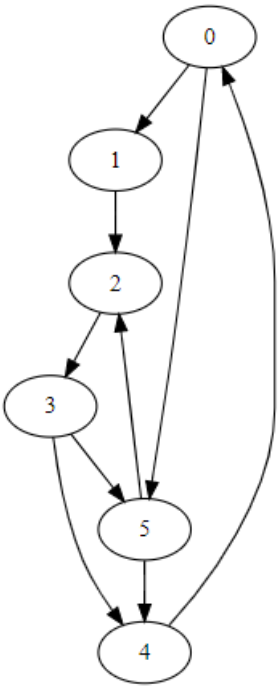
Enter your graphviz data into the Text Area:

(Your Graphviz data is private and never harvested)

Sample 1 | Sample 2 | Sample 3 | Sample 4 | Sample 5

```
digraph G { "0" -> "1"; "0" -> "5"; "1" -> "2"; "2" -> "3";
"3" -> "4"; "3" -> "5"; "4" -> "0"; "5" -> "4"; "5" -> "2"; }
```

Generate Graph!



In [5]:
```
print(g2.generate_digraph())
```

```
digraph G { "0" -> "3"; "1" -> "3"; "2" -> "3"; "3" -> "4"; "3" -> "6"; "4" -> "8"; "5" -> "6"; "6" -> "7";
"7" -> "8"; }
```

WebGraphviz is Graphviz in the Browser

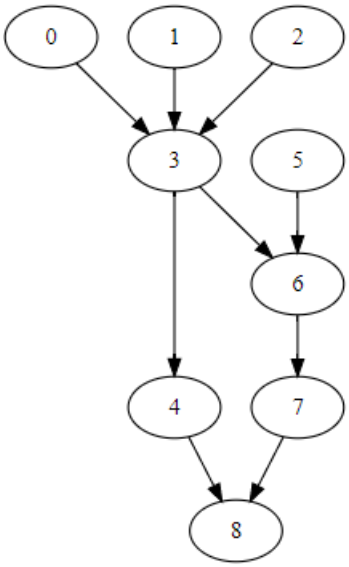
Enter your graphviz data into the Text Area:

(Your Graphviz data is private and never harvested)

Sample 1 | Sample 2 | Sample 3 | Sample 4 | Sample 5

```
digraph G { "0" -> "3"; "1" -> "3"; "2" -> "3"; "3" -> "4";
"3" -> "6"; "4" -> "8"; "5" -> "6"; "6" -> "7"; "7" -> "8"; }
```

Generate Graph!



Zadanie 3

Kolejnym naszym zadaniem było rozbudowanie klasy o metody przeszukiwania w głąb i wszerz.

Za algorytm przeszukiwania w głąb odpowiada `def dfs(self)` oraz `def dfsvisit(self, start_vertex)` , natomiast za algorytm przeszukiwania wszerz `def bfs(self, start)` . Były one prezentowane na wykładzie. [1] [2]

Przeszukiwanie wszerz

Do realizacji algorytmu wykorzystujemy kolejkę. Przeszukiwanie grafu wszerz zaczynamy od wierzchołka początkowego. Odwiedzamy wszystkie wierzchołki grafu sąsiadujące z nim, a następnie wszystkie wierzchołki z nich osiągalne (sąsiedzi sąsiadów) i tak dalej.

Postępowanie:

- mamy trzy kolory wierzchołków: biały (nieodwiedzony), szary (odwiedzony, ale niesprawdzony) i czarny (sprawdzony);
- kolorujemy wszystkie węzły na biało i rozpoczynamy przeszukiwanie od zadanego wierzchołka;
- dla każdego wierzchołka w jego liście sąsiedztwa zmieniamy kolor na szary;
- jeżeli wierzchołek startowy nie będzie miał już białych sąsiadów, zmieniamy jego kolor na czarny;
- powtarzamy całą procedurę dla każdego szarego sąsiada.

Gdy algorytm natrafi na biały węzeł, to zmieniamy jego kolor na szary, aktualny węzeł staje się jego poprzednikiem, odległość zwiększamy o 1 względem poprzednika oraz węzeł trafia na koniec kolejki kandydatów do przetworzenia.

Przeszukiwanie w głąb

Przeszukiwanie grafu w głąb polega na przeszukiwaniu wszystkich krawędzi grafu wychodzących z danego wierzchołka. Przechodzimy krawędź najdalej jak to możliwe, aż zostanie osiągnięty wierzchołek, który nie posiada nieodwiedzonych sąsiadów. Jeżeli dana ścieżka nie doprowadziła nas do wierzchołka końcowego, wówczas cofamy się do momentu, z którego możemy pójść kolejną dostępną krawędzią.

Do naszej klasy `class Graph` dodaliśmy nowy atrybut `time`, który przechowuje liczbę kroków. Metoda `def dfs(self)` odpowiada za budowanie lasu drzew DFS. Metoda `def dfsvisit(self, start_vertex)` odpowiada za odwiedzanie węzłów w gałęzi.

Zadanie 4

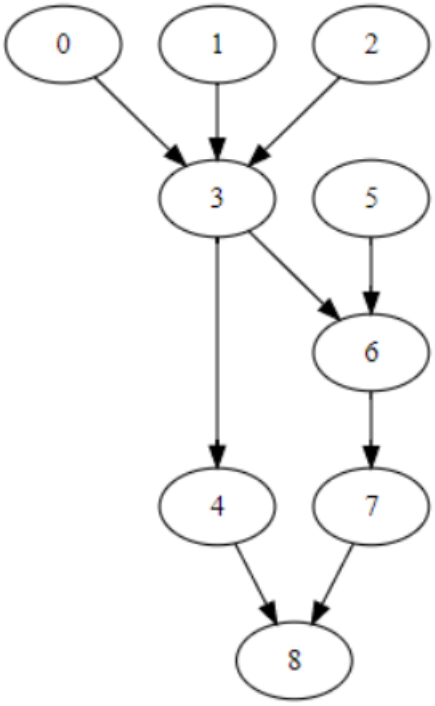
W tym zadaniu naszym celem było zaimplementowanie sortowania topologicznego. Stworzyliśmy metodę `def sort_topological(self)`, która przy użyciu przeszukiwania w głąb sortuje nasz graf. Dzięki niemu jesteśmy w stanie wyznaczyć czas odwiedzin oraz zakończenia przetwarzania węzła. Następnie wystarczy uporządkować węzły malejąco według czasów wykonania. Robimy to przy pomocy funkcji `lambda`.
Przy sortowaniu topologicznym należy pamiętać, by sprawdzić, czy graf jest acykliczny, ponieważ w przeciwnym wypadku nie jesteśmy w stanie posortować go topologicznie. Weryfikujemy to, przechodząc po uporządkowanych wierzchołkach.
Poniżej prezentujemy przykładowe sortowania topologiczne.

In [6]:
```
g = Graph()
for i in range(9):
        g.add_vertex(i)
g.add_edge(0, 3, 5)
g.add_edge(1, 3, 2)
g.add_edge(2, 3, 4)
g.add_edge(3, 4, 9)
g.add_edge(3, 6, 7)
g.add_edge(4, 8, 7)
g.add_edge(5, 6, 3)
g.add_edge(6, 7, 1)
g.add_edge(7, 8, 1)
g.add_edge(7, 8, 1)

print(g.sort_topological())
```

```
[5, 2, 1, 0, 3, 6, 7, 4, 8]
```

In [7]:
```
print(g.generate_digraph())
```

```
digraph G { "0" -> "3"; "1" -> "3"; "2" -> "3"; "3" -> "4"; "3" -> "6"; "4" -> "8"; "5" -> "6"; "6" -> "7";
"7" -> "8"; }
```



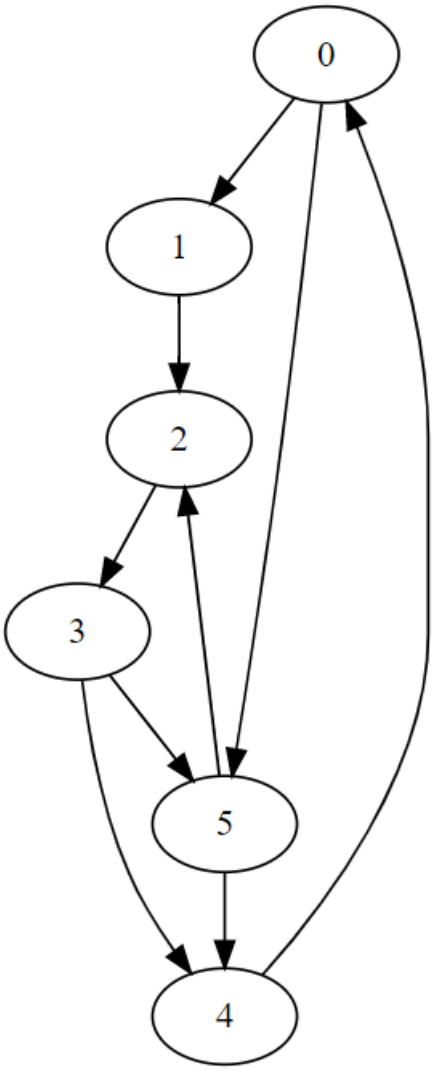
In [8]:
```
g = Graph()
for i in range(6):
    g.add_vertex(i)

g.add_edge(0, 1, 5)
g.add_edge(0, 5, 2)
g.add_edge(1, 2, 4)
g.add_edge(2, 3, 9)
g.add_edge(3, 4, 7)
g.add_edge(3, 5, 3)
g.add_edge(4, 0, 1)
g.add_edge(5, 4, 8)
g.add_edge(5, 2, 1)
print(g.sort_topological())
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-80c242a0f91a> in <module>
     12 g.add_edge(5, 4, 8)
     13 g.add_edge(5, 2, 1)
---> 14 print(g.sort_topological())

<ipython-input-1-5e6ca7055383> in sort_topological(self)
    316             return result
    317         else:
--> 318             raise ValueError('Graph is not linear!')
    319
    320 if __name__ == "__main__":

ValueError: Graph is not linear!
```

Graf nie jest liniowy, co możemy zobaczyć na rysunku.

In [9]:
```
print(g.generate_digraph())
```

```
digraph G { "0" -> "1"; "0" -> "5"; "1" -> "2"; "2" -> "3"; "3" -> "4"; "3" -> "5"; "4" -> "0"; "5" -> "4";
"5" -> "2"; }
```



Zadanie 5

W tym zadaniu musieliśmy stworzyć algorytm wyliczający najkrótszą ścieżkę od dowolnego węzła grafu do pozostałych. Dzięki algorytmowi Dijkstra zaprezentowanemu na wykładzie rozszerzymy działanie algorytmu i będziemy wyliczać najmniejszy koszt ścieżki. W przypadku gdy wagi ścieżek nie są podane, nadajemy im koszt 1 i w ten sposób znajdujemy najkrótszą drogę.

Metoda `def dijkstra(self, start)` wykorzystuje kopiec binarny (jego pierwotną wersję implementowaliśmy na poprzedniej liście zadań). Dodaliśmy w nim atrybut `atributes` , który przechowuje wagi. Dzięki niemu znamy najmniejszy koszt i trasę dotarcia do każdego z węzłów.

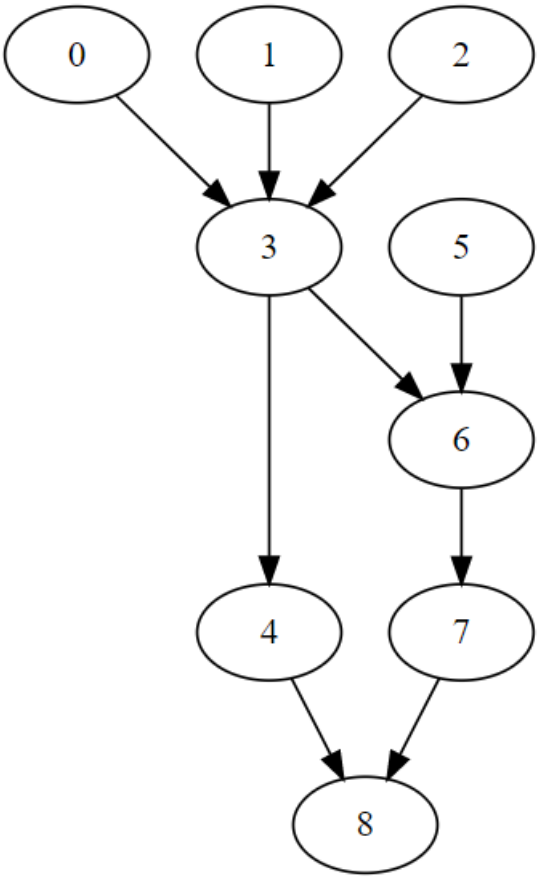
Metoda `def find_fastest(self, start)` zaczyna od wywołania algorytmu Dijkstra dla zadanego węzła *start*, a następnie przy wykorzystaniu metody `def traverse(self, vert)` , która zwraca trasę dotarcia do węzła, buduje słownik, w którym kluczem jest id węzła, a wartością krotka *((droga_do_węzła), koszt_trasy)*. Jeśli dotarcie do jakiegoś wierzchołka nie jest możliwe, wartość w słowniku wynosi *None*. Na końcu ustawiamy wartości `set_distance` oraz `set_pred` na domyślne - dzięki temu będziemy mogli wywołać funkcję ponownie, dla innego wierzchołka grafu.

Poniżej prezentujemy przykładowe wywołania.

In [10]:
```
g = Graph()
for i in range(9):
    g.add_vertex(i)
g.add_edge(0, 3, 5)
g.add_edge(1, 3, 2)
g.add_edge(2, 3, 4)
g.add_edge(3, 4, 9)
g.add_edge(3, 6, 7)
g.add_edge(4, 8, 7)
g.add_edge(5, 6, 3)
g.add_edge(6, 7, 1)
g.add_edge(7, 8, 1)
g.add_edge(7, 8, 1)

dictionary = g.find_fastest(3)
```

```
for x in dictionary:
    print (x,':', dictionary[x])
```

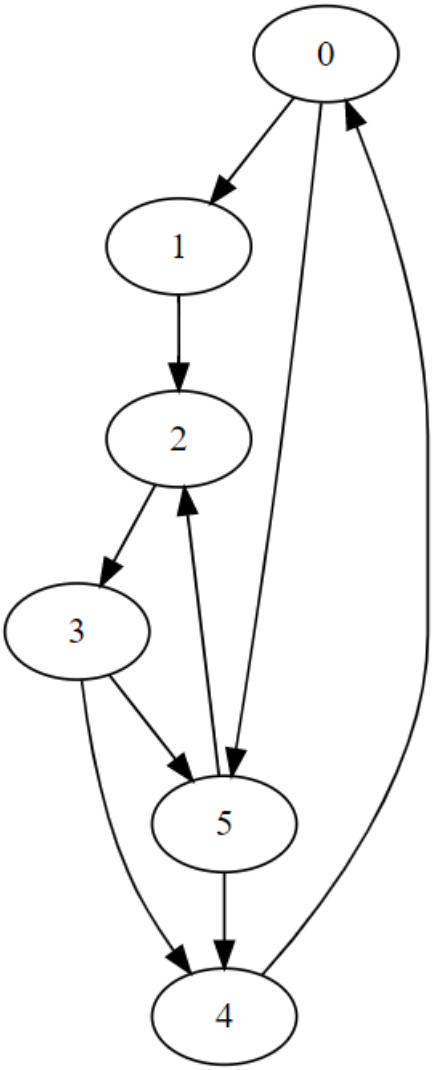
```
0 : None
1 : None
2 : None
3 : ((3,), 0)
4 : ((3, 4), 9)
5 : None
6 : ((3, 6), 7)
7 : ((3, 6, 7), 8)
8 : ((3, 6, 7, 8), 9)
```



In [11]:

```
g = Graph()
for i in range(6):
    g.add_vertex(i)
g.add_edge(0, 1, 5)
g.add_edge(0, 5, 2)
g.add_edge(1, 2, 4)
g.add_edge(2, 3, 9)
g.add_edge(3, 4, 7)
g.add_edge(3, 5, 3)
g.add_edge(4, 0, 1)
g.add_edge(5, 4, 8)
g.add_edge(5, 2, 1)

dictionary = g.find_fastest(3)
for x in dictionary:
    print (x,':', dictionary[x])
```

```
0 : ((3, 4, 0), 8)
1 : ((3, 4, 0, 1), 13)
2 : ((3, 5, 2), 4)
3 : ((3,), 0)
4 : ((3, 4), 7)
5 : ((3, 5), 3)
```



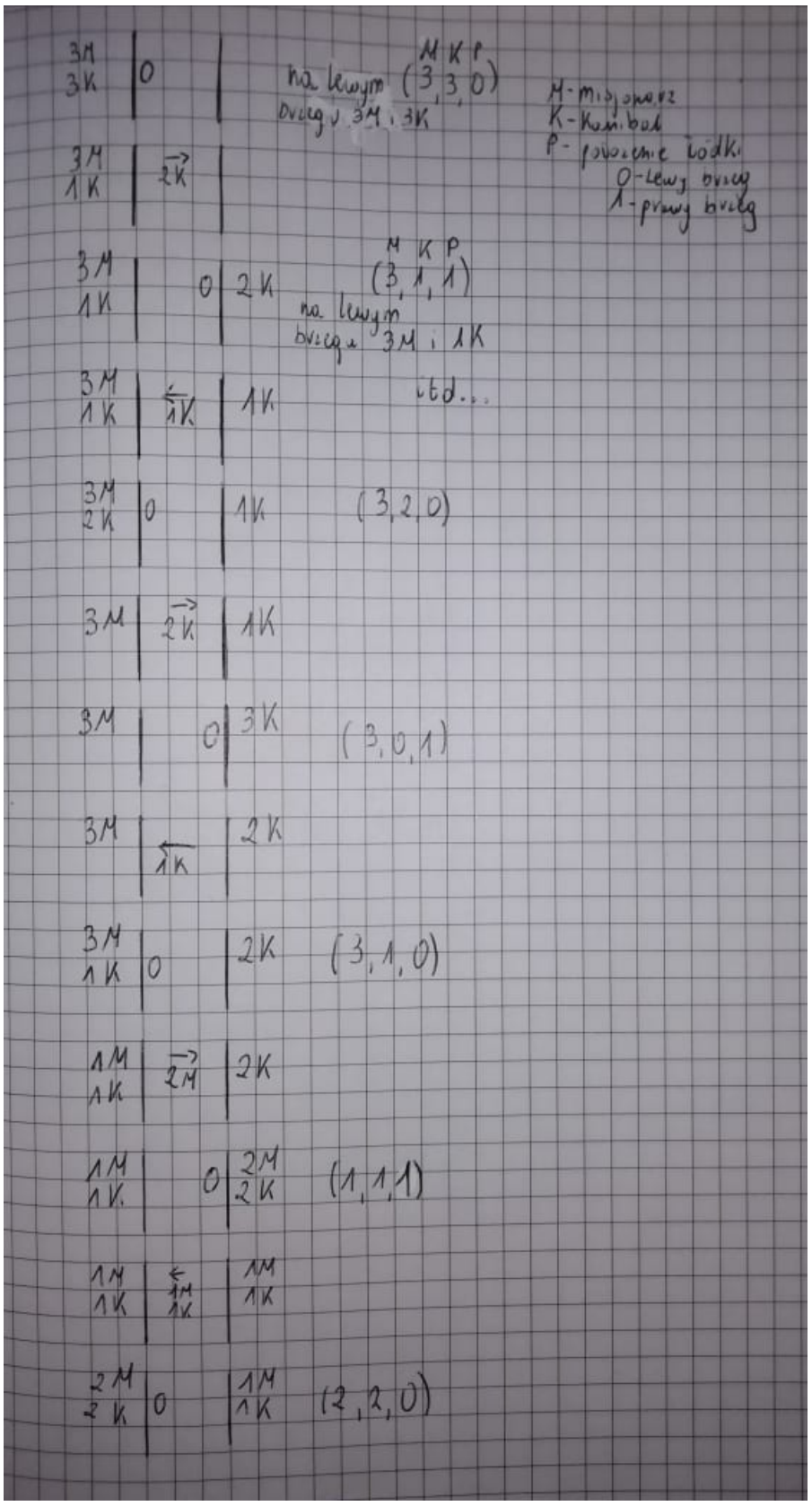
Zadanie 6

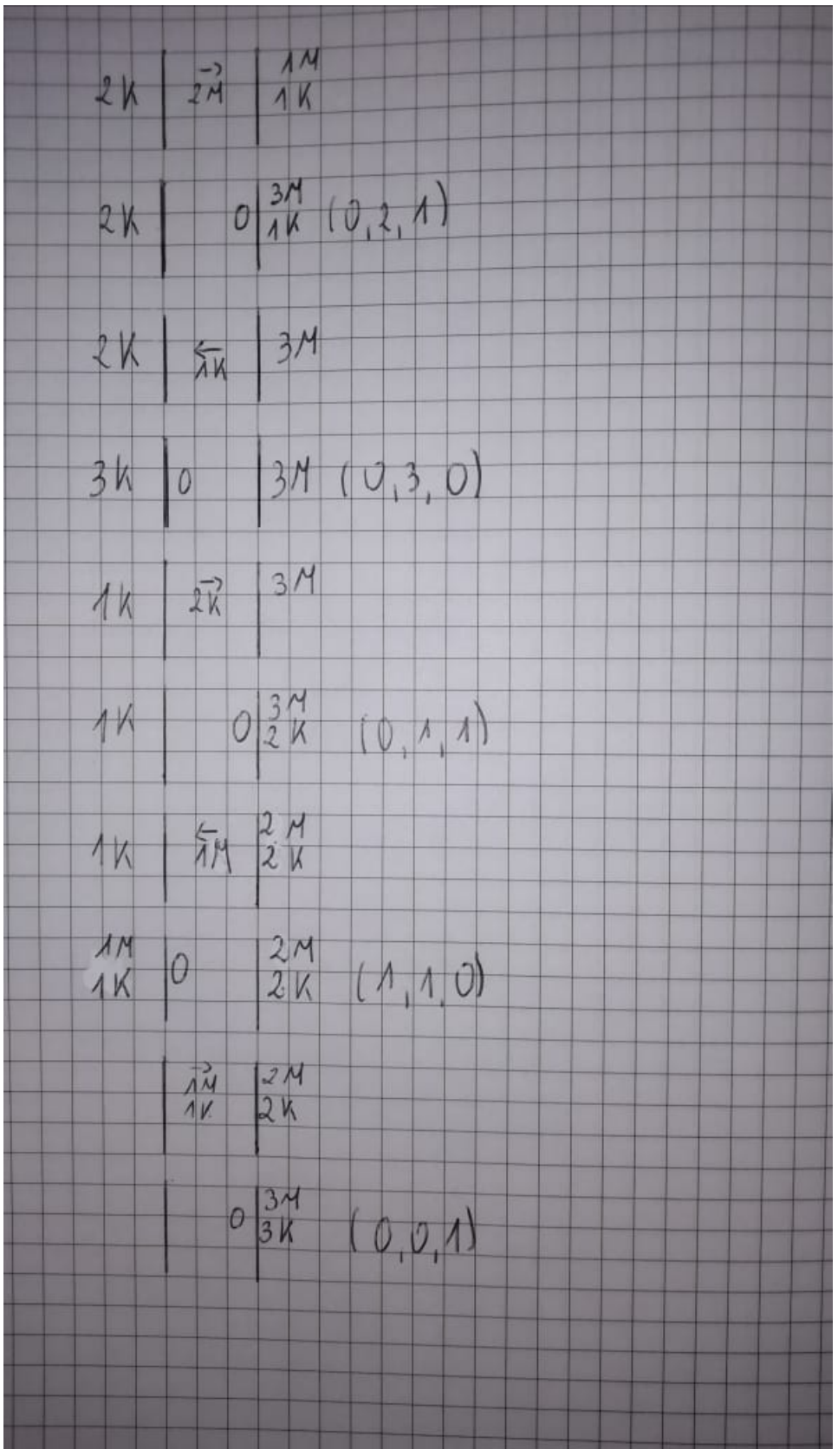
Zadanie 6 to rozważenie zagadnienia misjonarzy i kanibalów. Znajdują się oni na lewym brzegu rzeki i musimy ich przeprawić na drugi brzeg przy użyciu dwuosobowej łódki. Na jednym brzegu nie może znajdować się więcej kanibali niż misjonarzy, ponieważ wtedy zostaną oni zjedzeni. Program przedstawia kolejne ruchy, jakie należy podjąć, by rozwiązać łamigłówkę. Stworzyliśmy następujące funkcje:

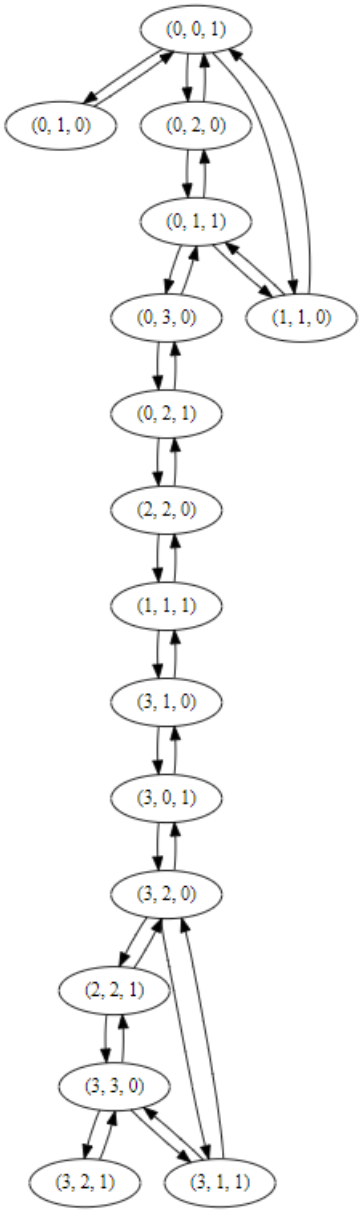
- `def game_over(location, missionaries, cannibals)` - Funkcja sprawdza, czy dla danego ułożenia (location) zachodzi koniec gry tzn. czy kanibali jest więcej niż misjonarzy. Zwraca True, jeśli następuje koniec gry. W przeciwnym wypadku - False.
- `def calculate_position(location, move)` - Funkcja oblicza i zwraca nowe rozłożenie misjonarzy, kanibali i łódki.
- `def move_legal(start, end)` - Funkcja sprawdza, czy możliwe jest przejście z rozłożenia start do rozłożenia end. Jeśli tak - zwraca True, a w przeciwnym wypadku False.
- `def create_graph(missionaries, cannibals)` - Główna funkcja tworząca graf. Na początku tworzy wszystkie możliwe wierzchołki grafu, a następnie tworzy między nimi połączenia. Zwraca stworzony graf.
- `def get_path(graph, start, finish)` - Funkcja znajduje i zwraca najlepszą ścieżkę na dotarcie z punktu start do finish.
- `def solve(missionaries, cannibals)` - Funkcja sterująca, wywołująca inne funkcje.
  Poniżej zamieszczamy przykładowe rozwiązania problemu misjonarzy i kanibalów. Liczba w krotce oznacza kolejno liczbę misjonarzy na lewym brzegu, kanibalów na lewym brzegu, położenie łódki (0 jeśli jest na lewym brzegu, 1 jeśli na prawym).

In [13]:
```
missionaries = 3
cannibals = 3
solve(missionaries, cannibals)
```

```
(((3, 3, 0), (3, 1, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1,
1), (1, 1, 0), (0, 0, 1)), 11)
```


M K P
3M 3K | 0 | na lewym brzegu 3M i 3K (3,3,0)
3M 1K | → 2K |
M K P
3M 1K | 0 | 2K (3,1,1)
na lewym brzegu 3M i 1K
3M 1K | ← 1K | 1K itd...
3M 2K | 0 | 1K (3,2,0)
3M | → 2K | 1K
3M | 0 | 3K (3,0,1)
3M | ← 1K | 2K
3M 1K | 0 | 2K (3,1,0)
1M 1K | → 2M | 2K
1M 1K | 0 | 2M 2K (1,1,1)
1M 1K | ← 1M 1K | 1M 1K
2M 2K | 0 | 1M 1K (2,2,0)
M - misjonarz
K - Kanibal
P - położenie łódki
0 - lewy brzeg
1 - prawy brzeg


2K | 2M → | 1M 1K
2K | 0 | 3M 1K (0,2,1)
2K | ← 1K | 3M
3K | 0 | 3M (0,3,0)
1K | 2K → | 3M
1K | 0 | 3M 2K (0,1,1)
1K | ← 1M | 2M 2K
1M 1K | 0 | 2M 2K (1,1,0)
| 1M → 1K | 2M 2K
| 0 | 3M 3K (0,0,1)

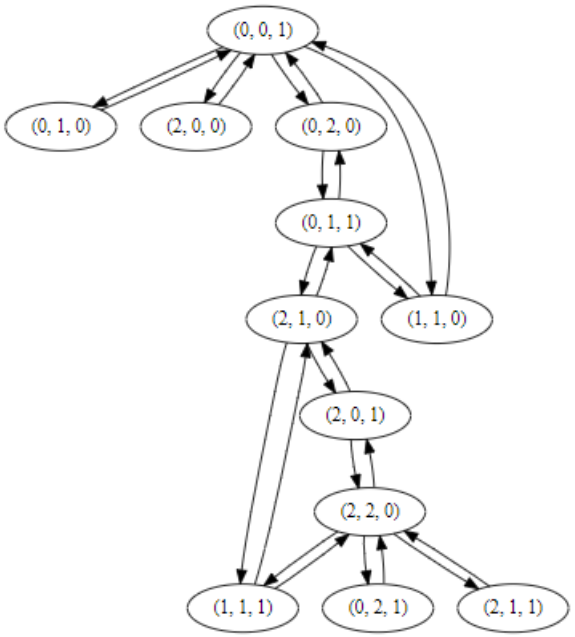


In [14]:
```
missionaries = 2
cannibals = 3
solve(missionaries, cannibals)
```

```
Missionaries were eaten.
```

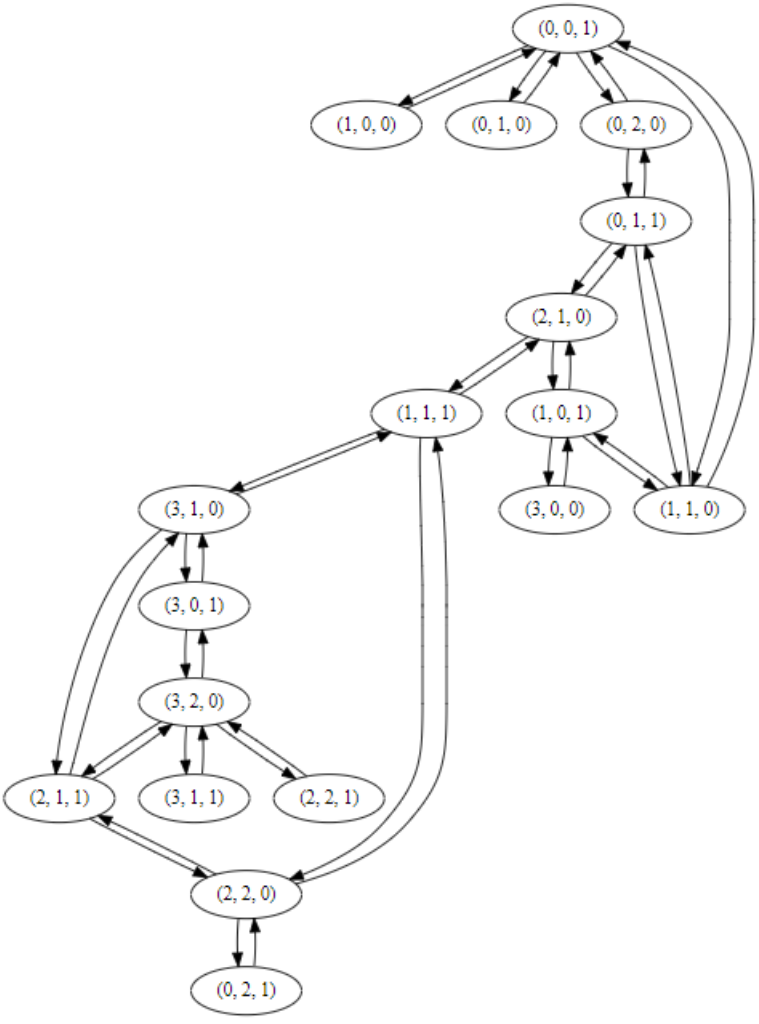
In [15]:
```
missionaries = 2
cannibals = 2
solve(missionaries, cannibals)
```

```
(((2, 2, 0), (1, 1, 1), (2, 1, 0), (0, 1, 1), (1, 1, 0), (0, 0, 1)), 5)
```



In [16]:
```
missionaries = 3
cannibals = 2
solve(missionaries, cannibals)
```

```
(((3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 1, 0), (0, 1, 1), (1, 1, 0), (0, 0, 1)), 7)
```



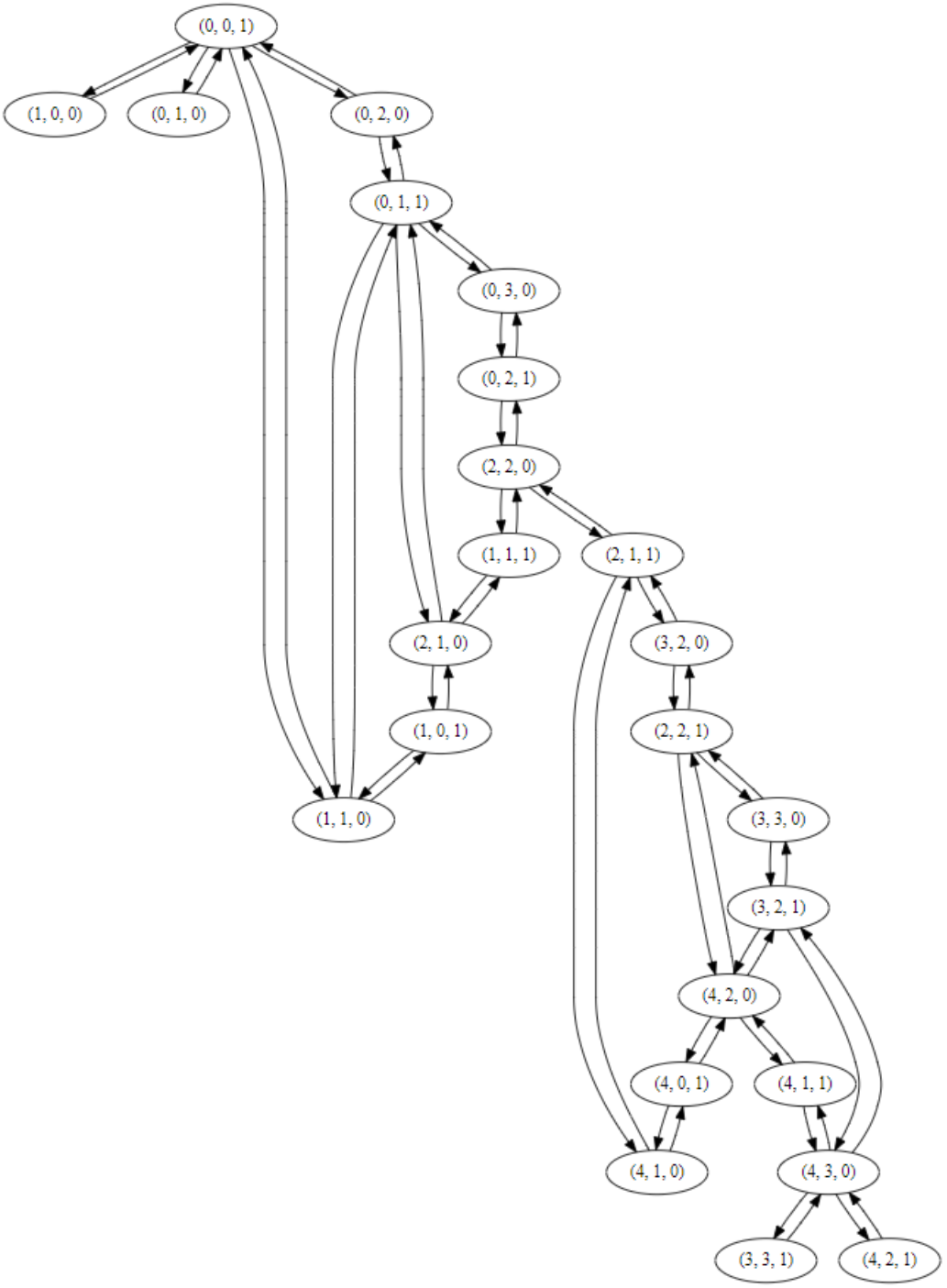
W przypadku gdy łódź może zabrać maksymalnie 2 osoby, a naszych par jest 4 lub więcej, zagadki nie da się rozwiązać.

In [17]:
```
missionaries = 4
cannibals = 4
solve(missionaries, cannibals)
```

```
Can not solve.
```

In [18]:
```
missionaries = 4
cannibals = 3
solve(missionaries, cannibals)
```

```
(((4, 3, 0), (4, 1, 1), (4, 2, 0), (2, 2, 1), (3, 2, 0), (2, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1,
1), (1, 1, 0), (0, 0, 1)), 11)
```



Zadanie 7

W ostatnim zadaniu naszym celem było napisanie programu do odmierzania konkretnej ilości wody, przy użyciu kanistrów o zadanej objętości. Zakładamy, że kanistry możemy dopełniać do maksymalnej objętości, wylewać z nich całą wodę oraz możemy dopełniać jeden kanister wodą znajdującą się w drugim kanistrze. Nasz program składa się z trzech funkcji:

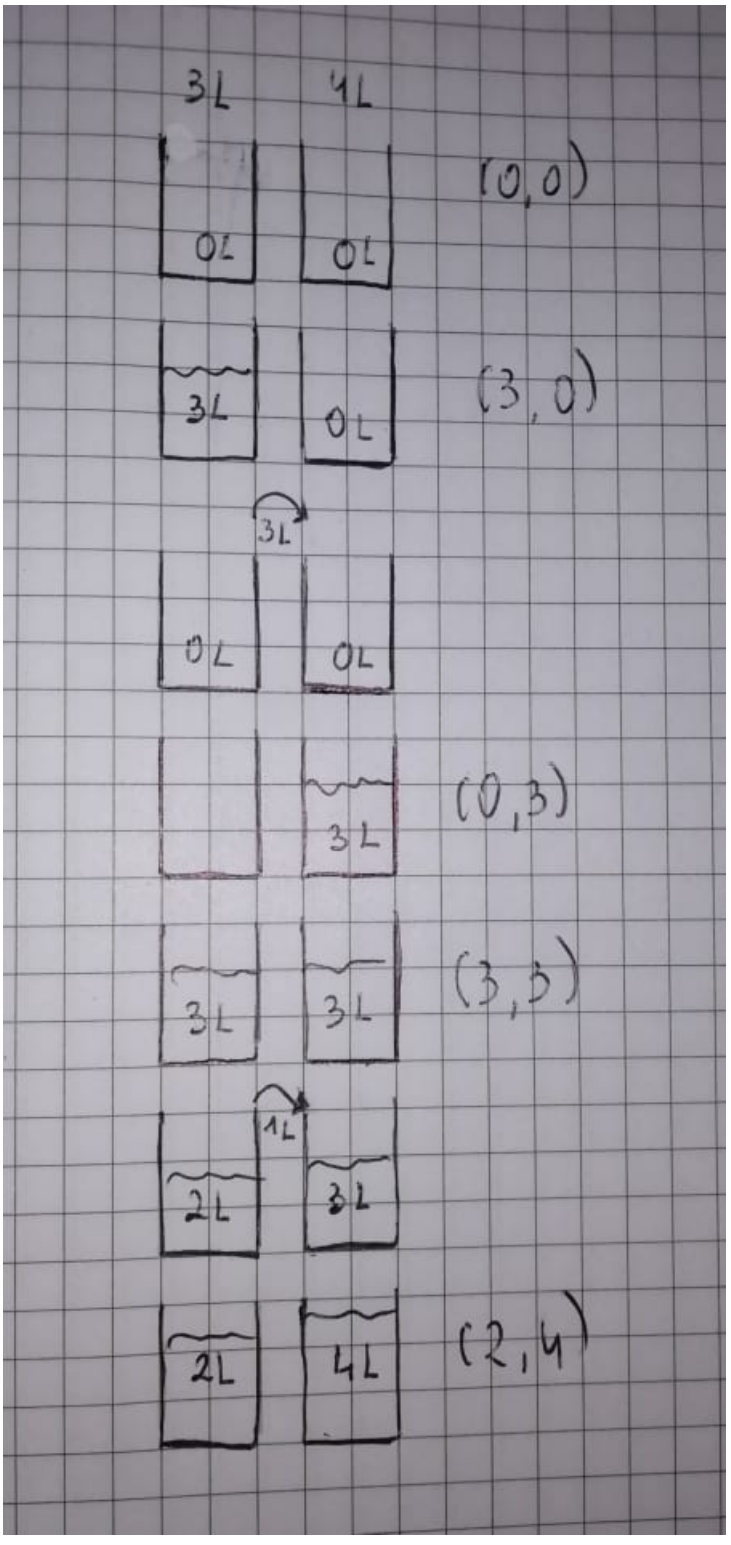
- `def build_graph(bucket1, bucket2)` - Główna funkcja tworząca graf. Na początku tworzy wierzchołki grafu dla liczb od 0 do objętości kanistrów. Następnie przechodzi po wszystkich stworzonych wierzchołkach i w zależności od sytuacji, tworzy możliwe powiązania między wierzchołkami.
- `def solve(x, y, goal)` - Funkcja na początku ustala, że pierwszy kanister to ten o mniejszej objętości, a następnie buduje graf, wywołując funkcję `build_graph` . Później dzięki `get_path` funkcja znajduje wszystkie możliwe ścieżki prowadzące od stanu początkowego (0,0) do wierzchołka w którym pojawia się szukana przez nas objętość. Ostatni krok to wybranie najkrótszej ścieżki. Jest to możliwe dzięki przejściu przez każdą ze ścieżek i porównaniu kosztów trasy.

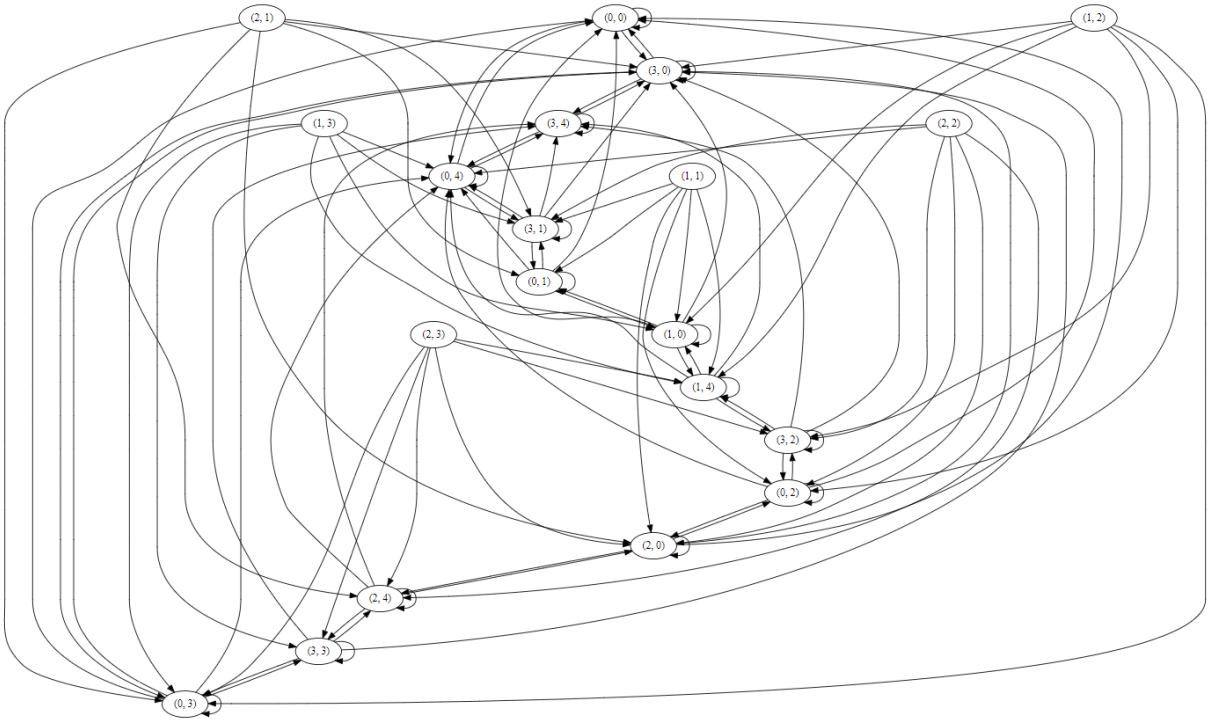
- `def get_path(graph, start, finish)` - Funkcja znajduje i zwraca najlepszą ścieżkę na dotarcie z punktu start do finish.

Poniżej przykładowe wywołania. Pierwsza liczba w krotce oznacza ilość wody w pierwszym kanistrze, odpowiednio druga liczba. Ostatnia liczba to ilość ruchów potrzebna, by rozwiązać zagadkę. Należy pamiętać, że drugi kanister to zawsze ten o większej objętości (nawet jeśli użytkownik podał inaczej).

In [20]:
```
bucket1 = 3
bucket2 = 4
goal = 2
solve(bucket1, bucket2, goal)
```

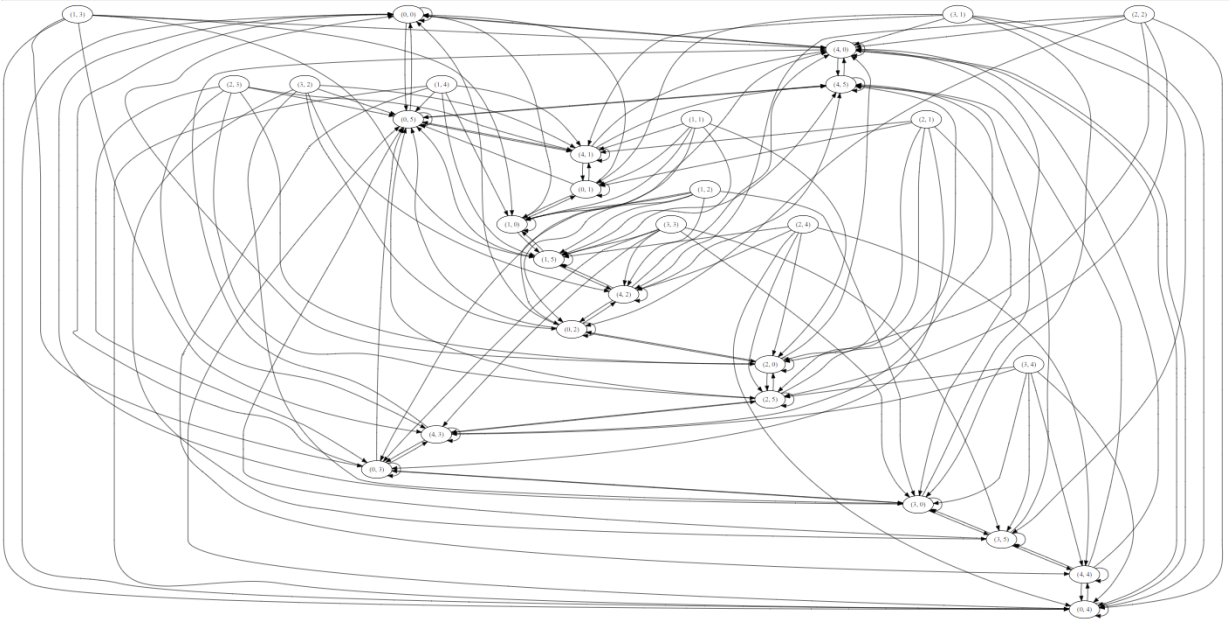
```
(((0, 0), (3, 0), (0, 3), (3, 3), (2, 4)), 4)
```





In [21]:
```
bucket1 = 5
bucket2 = 4
goal = 3
solve(bucket1, bucket2, goal)
```

```
(((0, 0), (4, 0), (0, 4), (4, 4), (3, 5)), 4)
```



In [22]:
```
bucket1 = 4
bucket2 = 4
goal = 3
solve(bucket1, bucket2, goal)
```

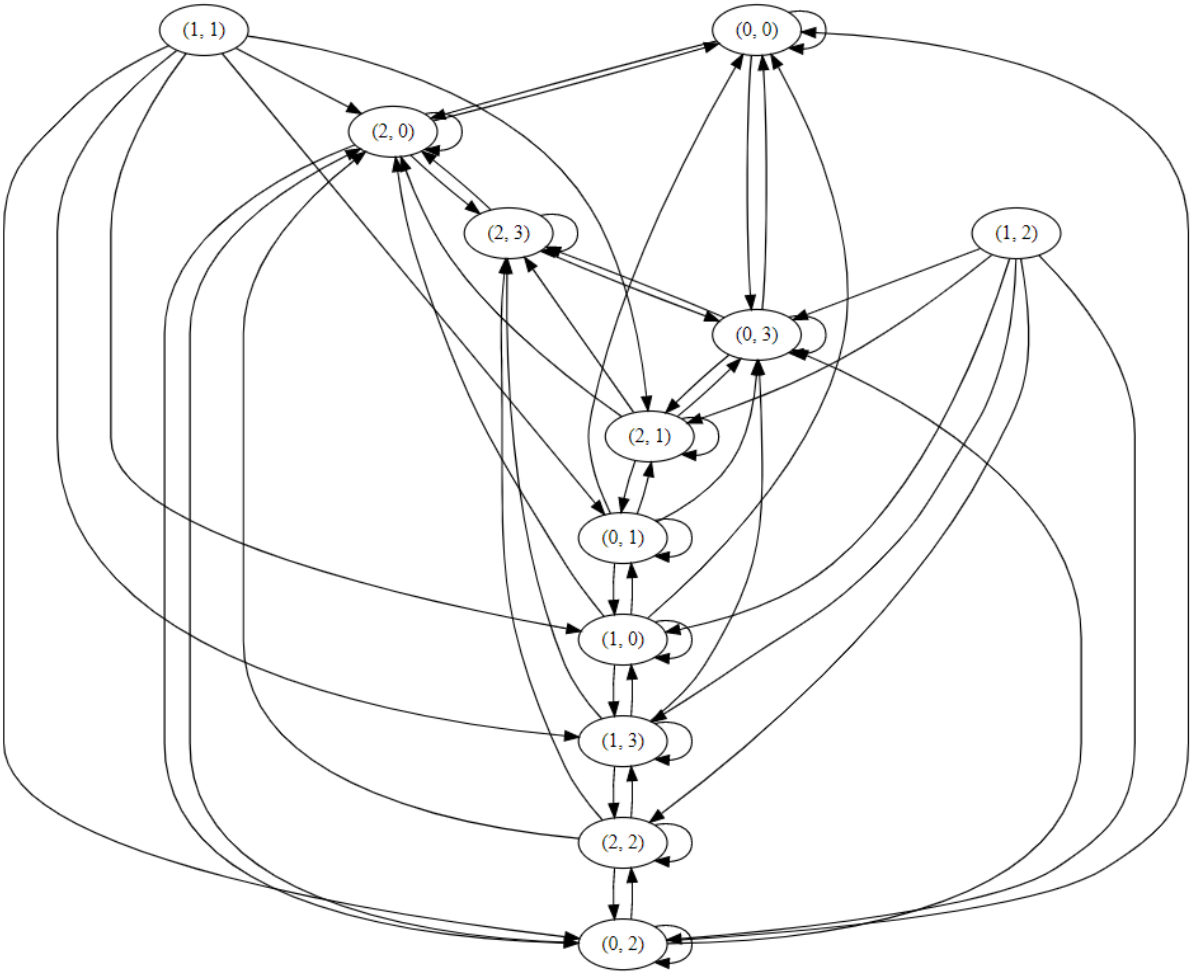
```
Can not solve.
```

In [23]:
```
bucket1 = 1
bucket2 = 2
goal = 3
solve(bucket1, bucket2, goal)
```

```
Can not solve.
```

In [24]:
```
bucket1 = 3
bucket2 = 2
goal = 1
solve(bucket1, bucket2, goal)
```

```
(((0, 0), (0, 3), (2, 1)), 2)
```



Linki

https://github.com/github-kamilk/AiSD/tree/main/Lista_7

Źródła

[1] J. Szwabiński, Wykład 10 - Grafy i podstawowe algorytmy grafowe, Algorytmy i struktury danych, str 12-13. Link: http://prac.im.pwr.wroc.pl/~szwabin/assets/algo/lectures/10.pdf.

[2] J. Szwabiński, Wykład 11 - Grafy i podstawowe algorytmy grafowe (ciąg dalszy), Algorytmy i struktury danych, str. 9. Link: http://prac.im.pwr.wroc.pl/~szwabin/assets/algo/lectures/11.pdf