Algorytmy i stuktury danych - Lista 7

Alicja Myśliwiec - gr. wtorek 7:30

In [2]:

```
import sys
from pythonds.graphs import PriorityQueue
from operator import itemgetter
```

Zad. 1 Zaimplementuj własną klasę **Graph** o własnościach podanych na wykładzie.

Zad. 2 Dodaj do powyższej klasy metodę generującą reprezentację grafu w języku **dot**. Użyj programu **graphviz** (lub jego wersji online: **http://www.webgraphviz.com/**) do przedstawienia wyniku na rysunku.

Zad. 3 Rozbuduj klasę o metody przeszukiwania w głąb i wszerz.

Zad. 4 Zmodyfikuj metodę przeszukiwania w głąb tak, aby sortowała ona graf topologicznie.

Zad. 5 Korzystając z przeszukiwania wszerz, stwórz algorytm wyliczający najkrótsze ścieżki od dowolnego węzła grafu do wszystkich pozostałych.

In [3]:

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)


class Vertex:
    def __init__(self, num):
        self.id = num
        self.connected_to = {}
        self.color = 'white'
        self.dist = sys.maxsize
        self.pred = None
        self.disc = 0
        self.fin = 0

    def add_neighbor(self, nbr, weight=0):
        self.connected_to[nbr] = weight

    def set_color(self, color):
        self.color = color

    def set_distance(self, d):
        self.dist = d

    def set_pred(self, p):
        self.pred = p

    def set_discovery(self, dtime):
        self.disc = dtime

    def set_finish(self, ftime):
        self.fin = ftime

    def get_finish(self):
        return self.fin

    def get_discovery(self):
        return self.disc

    def get_pred(self):
        return self.pred

    def get_distance(self):
        return self.dist

    def get_color(self):
        return self.color

    def get_connections(self):
        return self.connected_to.keys()

    def get_weight(self, nbr):
        return self.connected_to[nbr]

    def __str__(self):
        return str(self.id) + ":color " + self.color + ":disc " + str(self.disc) + ":fin " + str(
            self.fin) + ":dist " + str(self.dist) + ":pred \n\t[" + str(self.pred) + "]\n"

    def get_id(self):
        return self.id
```

In [69]:

```
class Graph:
    def __init__(self):
        self.vert_list = {}
        self.num_vertices = 0
        self.steps = 0

    def add_vertex(self, key):
        self.num_vertices += 1
        new_vertex = Vertex(key)
        self.vert_list[key] = new_vertex
        return new_vertex

    def delete_vertex(self, key):
        if key in self.get_vertices_list():
            self.num_vertices -= 1
            del self.vert_list[key]

    def get_vertex(self, n):
        if n in self.vert_list:
            return self.vert_list[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vert_list

    def add_edge(self, f, t, cost=0):
        if f not in self.vert_list:
            nv = self.add_vertex(f)
        if t not in self.vert_list:
            nv = self.add_vertex(t)
        self.vert_list[f].add_neighbor(self.vert_list[t], cost)

    def get_vertices(self):
        return self.vert_list.keys()

    def get_vertices_list(self):
        return list(self.get_vertices())

    def __iter__(self):
        return iter(self.vert_list.values())

    def dot_repr(self):
        dot_string = "digraph G {\n"
        for v in self:
            for w in v.get_connections():
                weight = v.connected_to[w]
                if weight != 0:
                    if isinstance(w.get_id(), (int, float)):
                        dot_string += '\t{} -> {} [ label = "{}" ];\n'.format(v.get_id(), w.get_id(), weight)
                    else:
                        dot_string += '\t"{}" -> "{}" [ label = "{}" ];\n'.format(v.get_id(), w.get_id(), weigh
                else:
                    if isinstance(w.get_id(), (int, float)):
                        dot_string += "\t{} -> {}\n".format(v.get_id(), w.get_id())
                    else:
                        dot_string += '\t"{}" -> "{}"\n'.format(v.get_id(), w.get_id())
        dot_string += "}"
        return dot_string

    def __str__(self):
        file = open("dot_repr.txt", 'w')
        file.writelines(self.dot_repr())
        file.close()
        return self.dot_repr()

    def bfs(self, start):
        start.set_distance(0)
        start.set_pred(None)
        vert_queue = Queue()
        vert_queue.enqueue(start)
        while vert_queue.size() > 0:
            current_vert = vert_queue.dequeue()
            for nbr in current_vert.get_connections():
                if nbr.get_color() == 'white':
                    nbr.set_color('gray')
                    nbr.set_distance(current_vert.get_distance() + 1)
                    nbr.set_pred(current_vert)
                    vert_queue.enqueue(nbr)
            current_vert.set_color('black')

    def dfs(self):
        for a_vert in self:
            a_vert.set_color('white')
            a_vert.set_pred(-1)
        for a_vert in self:
            if a_vert.get_color() == 'white':
                self.dfsvisit(a_vert)

    def dfsvisit(self, start_vert):
        start_vert.set_color('gray')
        self.steps += 1
        start_vert.set_discovery(self.steps)
        for next_vert in start_vert.get_connections():
            if next_vert.get_color() == 'white':
                next_vert.set_pred(start_vert)
                self.dfsvisit(next_vert)
        start_vert.set_color('black')
        self.steps += 1
        start_vert.set_finish(self.steps)

    def topological_sorting(self):
        self.dfs()
        verts = []
        for i in self.vert_list.keys():
            end_of_process_time = self.vert_list[i].get_finish()
            verts.append((i, end_of_process_time))
        verts.sort(key=itemgetter(1))
        sort = [i[0] for i in verts]
        sort.reverse()

        for i in range(1, self.num_vertices):
            current = sort[i]
            connections = [vert.id for vert in self.vert_list[current].get_connections()]
            for j in sort[:i]:
                if j in connections:
                    raise ValueError('Cannot sort the graph this way')
        return sort

    def dijkstra(self, start):
        pq = PriorityQueue()
        start.set_distance(0)
        pq.buildHeap([(v.get_distance(), v) for v in self])
        while not pq.isEmpty():
            current_vert = pq.delMin()
            for next_vert in current_vert.get_connections():
                new_dist = current_vert.get_distance() + current_vert.get_weight(next_vert)
                if new_dist < next_vert.get_distance():
                    next_vert.set_distance(new_dist)
                    next_vert.set_pred(current_vert)
                    pq.decreaseKey(next_vert, new_dist)

    def traverse(self, vert):
        result = []
        x = vert
        while x.get_pred():
            result.append(x.get_id())
            x = x.get_pred()
        result.append(x.get_id())
        result.reverse()
        return tuple(result)

    def fastest_route(self, start, end=None):
        self.dijkstra(self.get_vertex(start))
        routes = {}
        for vert in self.get_vertices_list():
            if vert == start:
                routes[vert] = tuple([0])
            else:
                route = self.traverse(self.get_vertex(vert))
                if start in route:
                    routes[vert] = route
                else:
                    routes[vert] = None

        for vert in self:
            vert.set_distance(sys.maxsize)

        if end is not None:
            if end in routes.keys():
                return "{}: {}".format(end, routes[end])
            else:
                raise KeyError("No such value in the graph")
        return routes

    def route_length(self, start, end=None):
        data = self.fastest_route(start, end)
        length_dict = {}
        for key in self.get_vertices_list():
            length_dict[key] = len(data[key]) - 1
        return length_dict

    def get_route_only(self, start, end):
        self.dijkstra(self.get_vertex(start))
        for vert in self.get_vertices_list():
            if vert == end:
                route = self.traverse(self.get_vertex(vert))
                return route
        raise KeyError("Cannot find the wanted key")
```

In [70]:

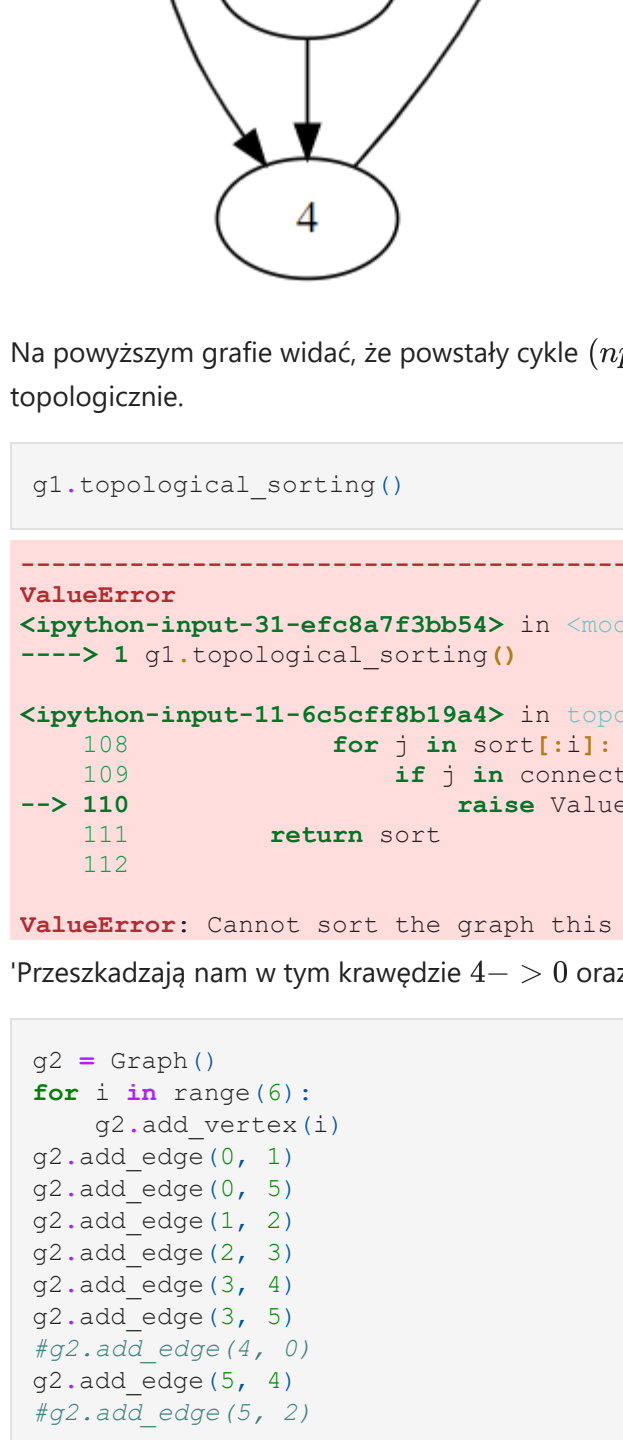
```
g1 = Graph()
for i in range(6):
    g1.add_vertex(i)
g1.add_edge(0, 1)
g1.add_edge(0, 5)
g1.add_edge(1, 2)
g1.add_edge(2, 3)
g1.add_edge(3, 4)
g1.add_edge(3, 5)
g1.add_edge(4, 0)
g1.add_edge(5, 4)
g1.add_edge(5, 2)
```

In [71]:

```
print(g1)
```

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

Wynik otrzymany na stronie http://www.webgraphviz.com/



Na powyższym grafie widać, że powstały cykle ($np. 2->3->5->2$). Sugeruje nam to, że nie bedzie dało się posortować grafu topologicznie.

In [31]:

```
g1.topological_sorting()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-efc8a7f3bb54> in <module>
----> 1 g1.topological_sorting()

<ipython-input-11-6c5cff8b19a4> in topological_sorting(self)
    108             for j in sort[:i]:
    109                 if j in connections:
--> 110                     raise ValueError('Cannot sort the graph this way')
    111         return sort
    112

ValueError: Cannot sort the graph this way
```

'Przeszkadzają nam w tym krawędzie $4->0$ oraz $5->2$.

In [32]:

```
g2 = Graph()
for i in range(6):
    g2.add_vertex(i)
g2.add_edge(0, 1)
g2.add_edge(0, 5)
g2.add_edge(1, 2)
g2.add_edge(2, 3)
g2.add_edge(3, 4)
g2.add_edge(3, 5)
#g2.add_edge(4, 0)
g2.add_edge(5, 4)
#g2.add_edge(5, 2)
```

In [33]:

```
print(g2)
```

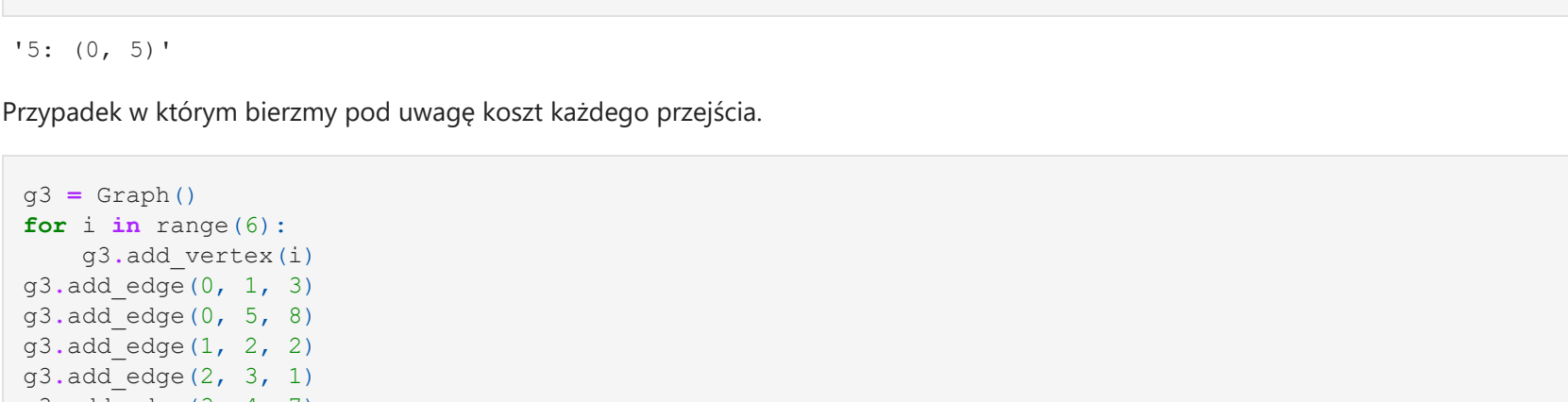
```
digraph G {
        0 -> 1
        0 -> 5
        1 -> 2
        2 -> 3
        3 -> 4
        3 -> 5
        5 -> 4
}
```

In [39]:

```
g2.topological_sorting()
```

Out[39]: `[0, 1, 2, 3, 5, 4]`

Została zwrócona kolejność w jakiej będzie posortowany graf.



Do zadania 5 wykorzystany został algorytm Dijkstry. Dzięki niemu, można znaleźć najkrótsze ścieżki, również zwracając uwagę na wagę danego przejścia. Wróćmu do pierwszego utworzonego grafu.

In [40]:

```
g1.route_length(0)
```

Out[40]: `{0: 0, 1: 1, 2: 2, 3: 3, 4: 2, 5: 1}`

Przejścia przez konkretne punkty:

In [41]:

```
g1.fastest_route(0)
```

Out[41]: `{0: (0,), 1: (0, 1), 2: (0, 1, 2), 3: (0, 1, 2, 3), 4: (0, 5, 4), 5: (0, 5)}`

Można również znaleźć najkrótsza trasę między konkretnymi punktami.

In [42]:

```
g1.fastest_route(0, 5)
```

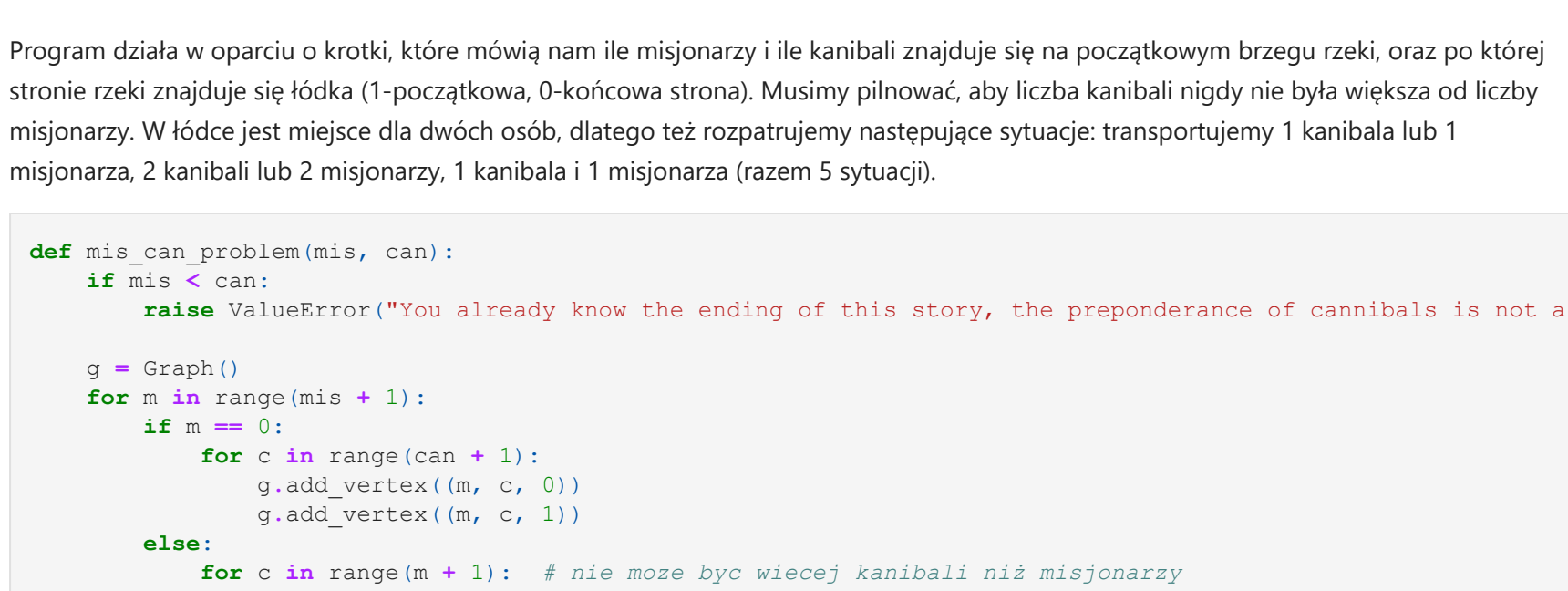
Out[42]: `'5: (0, 5)'`

Przypadek w którym bierzmy pod uwagę koszt każdego przejścia.

In [72]:

```
g3 = Graph()
for i in range(6):
    g3.add_vertex(i)
g3.add_edge(0, 1, 3)
g3.add_edge(0, 5, 8)
g3.add_edge(1, 2, 2)
g3.add_edge(2, 3, 1)
g3.add_edge(3, 4, 7)
g3.add_edge(3, 5, 1)
g3.add_edge(4, 0, 1)
g3.add_edge(5, 4, 1)
g3.add_edge(5, 2, 9)
print(g3)
```

```
digraph G {
        0 -> 1 [ label = "3" ];
        0 -> 5 [ label = "8" ];
        1 -> 2 [ label = "2" ];
        2 -> 3 [ label = "1" ];
        3 -> 4 [ label = "7" ];
        3 -> 5 [ label = "1" ];
        4 -> 0 [ label = "1" ];
        5 -> 4 [ label = "1" ];
        5 -> 2 [ label = "9" ];
}
```



Ilością kroków wydawałoby się, że naszybsza trasa $0->4$, to $0->5->4$. Jednak biorąc pod uwagę koszt przejść:

In [59]:

```
g3.fastest_route(0, 4)
```

Out[59]: `'4: (0, 1, 2, 3, 5, 4)'`

Zad. 6 Korzystając z grafów, napisz program rozwiązujący zagadnienie misjonarzy i kanibalów (**https://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem**).

Program działa w oparciu o krotki, które mówią nam ile misjonarzy i ile kanibali znajduje się na początkowym brzegu rzeki, oraz po której stronie rzeki znajduje się łódka (1-początkowa, 0-końcowa strona). Musimy pilnować, aby liczba kanibali nigdy nie była większa od liczby misjonarzy. W łódce jest miejsce dla dwóch osób, dlatego też rozpatrujemy następujące sytuacje: transportujemy 1 kanibala lub 1 misjonarza, 2 kanibali lub 2 misjonarzy, 1 kanibala i 1 misjonarza (razem 5 sytuacji).

In [77]:

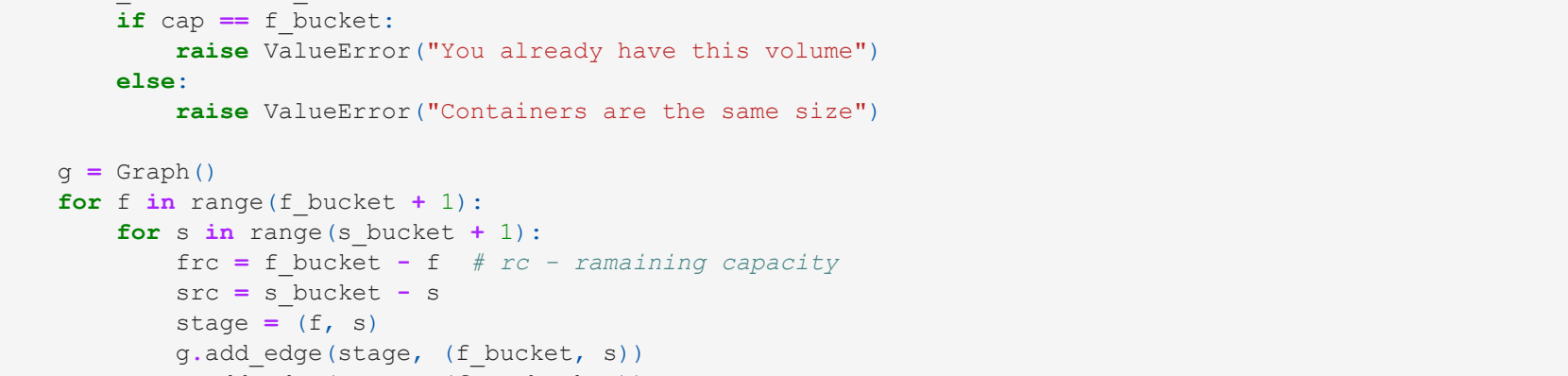
```
def mis_can_problem(mis, can):
    if mis < can:
        raise ValueError("You already know the ending of this story, the preponderance of cannibals is not a go

    g = Graph()
    for m in range(mis + 1):
        if m == 0:
            for c in range(can + 1):
                g.add_vertex((m, c, 0))
                g.add_vertex((m, c, 1))
        else:
            for c in range(m + 1):  # nie moze byc wiecej kanibali niż misjonarzy
                g.add_vertex((m, c, 0))
                g.add_vertex((m, c, 1))

    for vert in g.get_vertices_list():  # po drugiej stronie również
        current_mis, current_can = vert[0], vert[1]
        if current_mis not in [0, mis]:
            if current_mis >= current_can and mis - current_mis >= can - current_can:
                pass
            else:
                g.delete_vertex(vert)

    options = []
    for vert in g.get_vertices_list():
        river_bank_side = vert[2]
        current_mis, current_can = vert[0], vert[1]
        if river_bank_side == 1:   # przepłyniecie na drugą stronę
            options.append((current_mis - 1, current_can, 0))
            options.append((current_mis, current_can - 1, 0))
            options.append((current_mis - 1, current_can - 1, 0))
            options.append((current_mis - 2, current_can, 0))
            options.append((current_mis, current_can - 2, 0))
            for o in options:
                if o in g.get_vertices_list():
                    g.add_edge(vert, o)
        elif river_bank_side == 0:  # powrót
            options.append((current_mis + 1, current_can, 1))
            options.append((current_mis, current_can + 1, 1))
            options.append((current_mis + 1, current_can + 1, 1))
            options.append((current_mis + 2, current_can, 1))
            options.append((current_mis, current_can + 2, 1))
            for o in options:
                if o in g.get_vertices_list():
                    g.add_edge(vert, o)
        options = []

    route = g.get_route_only((mis, can, 1), (0, 0, 0))
    final_ret = ""
    for step in route:   # aby przejscie pokazywały stan w miejscu gdzie znajduje sie łódź
        if step[2] == 0:
            step = (mis - step[0], can - step[1], 0)
        final_ret += "-> {} ".format(step)
    return final_ret
```

Krotki w funkcji przedstawiają sytuację na lewym brzegu, stąd szukamy najkrótszej ścieżki do (0, 0, 0) (po drugiej stronie są już wszyscy, po lewej stronie 0 misjonarzy, 0 kanibali, łódka na prawym brzegu - 0). Otrzymany wynik jednak został zmieniony, aby pokazać sytuację tym na brzegu, przy którym aktualnie znajduje się łódka.

In [80]:

```
mis_can_problem(3, 3)
```

Out[80]: `'-> (3, 3, 1) -> (0, 2, 0) -> (3, 2, 1) -> (0, 3, 0) -> (3, 1, 1) -> (2, 2, 0) -> (2, 2, 1) -> (3, 1, 0) -> (0, 3, 1) -> (3, 2, 0) -> (1, 1, 1) -> (3, 3, 0) '`



In [79]:

```
mis_can_problem(7, 6)
```

Out[79]: `'-> (7, 6, 1) -> (1, 1, 0) -> (7, 5, 1) -> (0, 3, 0) -> (7, 4, 1) -> (2, 2, 0) -> (5, 5, 1) -> (3, 2, 0) -> (5, 4, 1) -> (3, 3, 0) -> (4, 4, 1) -> (4, 3, 0) -> (4, 3, 1) -> (4, 4, 0) -> (3, 3, 1) -> (5, 4, 0) -> (3, 2, 1) -> (5, 5, 0) -> (2, 2, 1) -> (6, 5, 0) -> (2, 1, 1) -> (6, 6, 0) -> (1, 1, 1) -> (7, 6, 0) '`

Zad. 7 Napisz program, który znajdzie sposób na odmierzenie dwóch litrów wody przy użyciu dwóch kanistrów o objętości 3l i 4l.

W tym zadaniu sytuacja ponownie będzie przedstawiona na krotkach, które oznaczają ilość wody w kanistrach. Dla każdych pojemności tworzymy sytuacje: dolewania, wylewania i przelewania. Na końcu szukamy najkrótszej ścieżki, która łączy warunki początkowe z sytuacją, gdzie otrzymujemy porządaną objętość.

In [73]:

```
def water_transfer(f_bucket, s_bucket, cap):
    if cap > f_bucket and cap > s_bucket:
        raise ValueError("Expected volume is too large")

    if f_bucket == s_bucket:
        if cap == f_bucket:
            raise ValueError("You already have this volume")
        else:
            raise ValueError("Containers are the same size")

    g = Graph()
    for f in range(f_bucket + 1):
        for s in range(s_bucket + 1):
            frc = f_bucket - f  # rc - ramaining capacity
            src = s_bucket - s
            stage = (f, s)
            g.add_edge(stage, (f_bucket, s))
            g.add_edge(stage, (f, s_bucket))
            g.add_edge(stage, (f, 0))
            g.add_edge(stage, (0, s))
            if f > src:
                g.add_edge(stage, (f - src, s + src))
            else:
                g.add_edge(stage, (0, s + f))
            if s > frc:
                g.add_edge(stage, (f + frc, s - frc))
            else:
                g.add_edge(stage, (s + f, 0))

    if f_bucket > cap and s_bucket > cap:
        route = min([g.get_route_only((0, 0), (cap, 0)), g.get_route_only((0, 0), (0, cap))], key=len)
    else:
        if f_bucket > cap:
            route = g.get_route_only((0, 0), (cap, 0))
        else:
            route = g.get_route_only((0, 0), (0, cap))
    final_ret = ""
    for val in route:
        final_ret += " -> {}".format(val)
    return final_ret
```

In [74]:

```
water_transfer(4, 3, 2)
```

Out[74]: `' -> (0, 0) -> (0, 3) -> (3, 0) -> (3, 3) -> (4, 2) -> (0, 2)'`

Kolejne przejścia:

- najpierw nalewamy wodę do kanistra o pojemności 3l
- przelewamy zawartość do drugiego kanistra (poj. 4l)
- znów uzupełniamy pierwszy (poj. 3l)
- ponownie przelewamy, do drugiego kanistra możemy przelać jedynie litr (4 - 3 = 1), przez co w pierwszym pozostają 2 litry (3 - 1 = 2), których szukamy
- krok niewymagany, czyli wylanie wody z drugiego kanistra.

In [75]:

```
water_transfer(7, 16, 3)
```

Out[75]: `' -> (0, 0) -> (7, 0) -> (0, 7) -> (7, 7) -> (0, 14) -> (7, 14) -> (5, 16) -> (5, 0) -> (0, 5) -> (7, 5) -> (0, 12) -> (7, 12) -> (3, 16) -> (3, 0)'`

In [76]:

```
water_transfer(5, 3, 4)
```

Out[76]: `' -> (0, 0) -> (5, 0) -> (2, 3) -> (2, 0) -> (0, 2) -> (5, 2) -> (4, 3) -> (4, 0)'`

LINK GITHUB

https://github.com/AlutkaMalutka/Programowanie_python/tree/main/semestr_3/Lista_7-3s-