Programování 2

12. cvičení, 07-05-2024

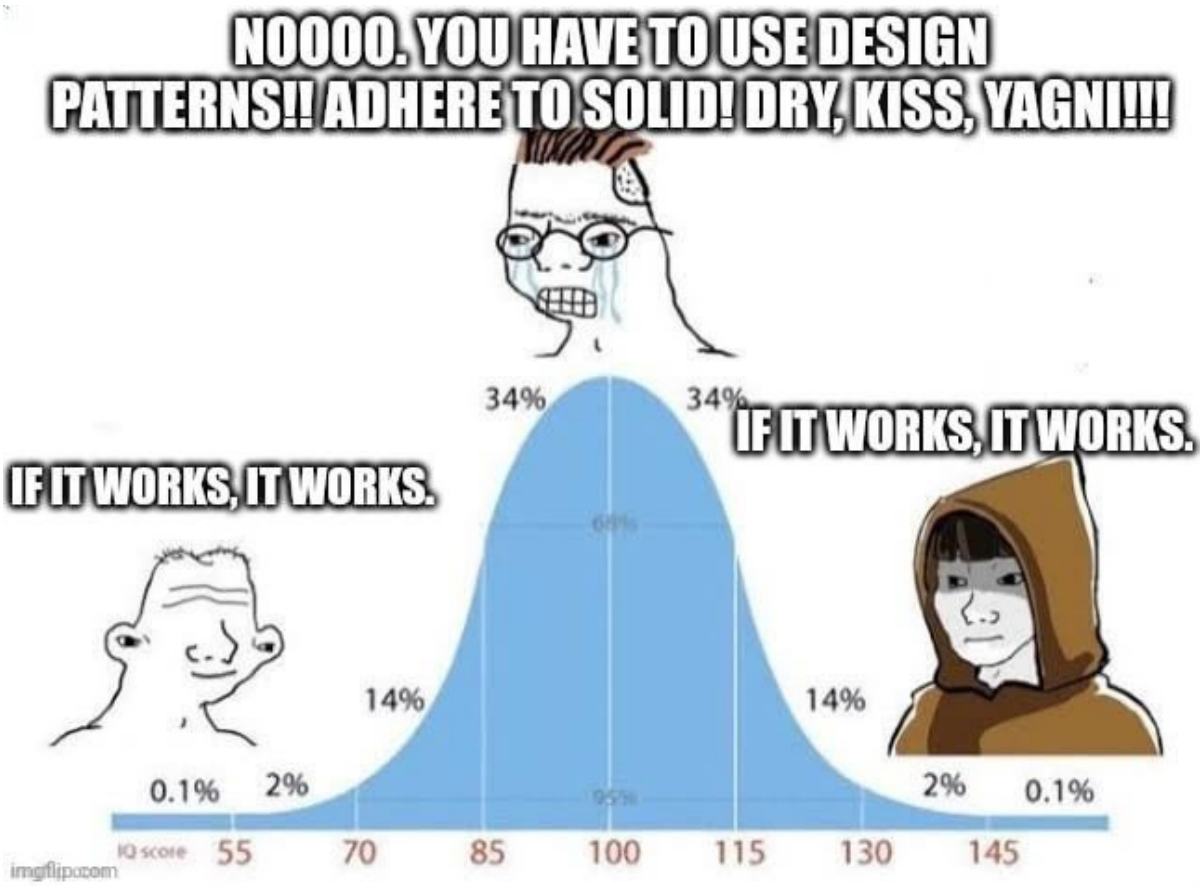
Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáří cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**
   - Cesta věže - lehká úloha na prohledávání stromu
   - Domino - velice typická úloha na prohledávání do hloubky.
   - Kružnice v grafu - prohledávání grafu
3. **Zápočtový program**:

   Většina z vás má už domluvené téma zápočtového programu. Příklad, jak může váš zápočtový program vypadat. naleznete na https://github.com/kolik95/pexeso
4. **Průběh semestru**:
   - Toto je **poslední** praktické cvičení, příští týden cvičení nebude (rektorský den)
   - **Dostanete poslední domácí úkoly** v bonusovém režřimu: počty bodů se nebudou započítavat do maximálního počtu bodů, takže se nebude zvyšovat počet bodů nutných k získání zápočtu. Kdo by chtěl dodatečné příklady, aby si zlepšil bodovou bilanci, napište mi.
   - 21. května bude zápočtový test:
     - Přijdete na cvičení v obvyklém termínu 15:40, resp. 17:20
     - Dostanete jedinou programovací úlohu, kterou vyřešíte přímo na cvičení ve vymezeném čase 75 minut.
     - Řešení nahrajete do ReCodExu a tam najdete i hodnocení.
   - Zápočet za teoretické a praktické cvičení dostanete ode mne. Podmínky:
     - schválení od cvičícího na teoretickém cvičení
     - domácí úkoly
     - zápočtový test
     - zápočtový program
   - Opravné prostředky:
     - Umíme dát do pořádku mírná selhání v některých disciplínách - domácí úkoly, zápočtový test a třeba i zápočtový program.

Dnešní program:

- Piškvorky
- Domácí úkoly
- Prohledávání stavového prostoru: 8 dam
- Grafy a grafové algoritmy



Měli byste hlavně psát čistý a robustní kód.

Práce v souborovém systému: `pathlib`

Třída `Path`: adresa objektu v souborovém systému.

```
from pathlib import Path

Path("main.py").exists()
Out[3]: True

Path("img").mkdir()         # Nový adresář

Path("img").mkdir()         # Už existuje - chyba
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-
packages\IPython\core\interactiveshell.py", line 3369, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-5-c6fb86db84a7>", line 1, in <cell line: 1>
    Path("img").mkdir()
  File "C:\ProgramData\Anaconda3\lib\pathlib.py", line 1323, in mkdir
    self._accessor.mkdir(self, mode)
FileExistsError: [WinError 183] Cannot create a file when that file already
exists: 'img'

Path("img").mkdir(exist_ok=True) # nedojde k chybě
```

Soubory můžeme také přesouvat. `pathlib` rozumí, v jakém operačním sytému pracuje:

```
file = Path("sk.py").replace("img/sk.py")
file
Out[8]: WindowsPath('img/sk.py')
```

`.parent`, `.name` a další:

```
Path("img").parent
Out[10]: WindowsPath('.')
Path("img").parent.parent
Out[11]: WindowsPath('.')
```

V jakém adresáři běží můj skript?

```
from pathlib import Path

folder = Path("__file__").parent
print(folder)
```

Domácí úkoly

Cesta věže

Toto je malý problém, takže není vůbec potřebné moc se zamýšlet. Prozkoumáme všechny možné tahy věže, a na každém navštíveném poli necháme informaci o tom, kolika nejméně tahy se k němu lze dostat.

- Používáme prioritní frontu: nejdřív se zabýváme poli, které vidíme poprvé.

```
# Find the shortest path of the rook on chessboard
from itertools import product
import heapq


SIZE = 8
INF = 10_000


@lambda cls: cls()  # Create class instance immediately
class Chessboard:
    def __init__(self):
        """Just create chessboard"""
        self.chessboard = dict([((i,j),INF) for i, j in product(range(SIZE),
range(SIZE))])
        self.start = None
        self.end = None

    def is_in_range(self, k, l):
        return (k,l) in self.chessboard.keys()

    def set_obstacle(self, i, j):
        self.chessboard[(i,j)] = -1
```

```
    def set_start(self, i, j):
        self.start = (i, j)
        self.chessboard[(i,j)] = 0

    def set_end(self, i, j):
        self.end = (i, j)

    def get_steps(self, i, j):
        return self.chessboard[(i,j)]

    def set_steps(self, i, j, steps):
        self.chessboard[(i,j)] = steps

    def rook_fields(self, i, j):
        """Return a list of fields controlled by a queen at (i, j)"""
        steps = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        fields = []
        for s, t in steps:
            k = i + s
            l = j + t
            while self.is_in_range(k, l) and self.chessboard[k,l] != -1:
                fields.append((k, l))
                k = k + s
                l = l + t
        return fields

    def print(self):
        chart = [["_" for _ in range(SIZE)] for _ in range(SIZE)]
        for pos, steps in self.chessboard.items():
            i, j = pos
            if steps == -1:
                chart[i][j] = "x"
            elif steps == INF:
                chart[i][j] = "?"
            else:
                chart[i][j] = str(steps)
        for i in range(SIZE):
            print(*chart[i])
        print()


def read_chessboard():
    global Chessboard
    for i in range(SIZE):
        row = input().strip()
        for j in range(SIZE):
            if row[j] == ".":
                continue
            elif row[j] == "x":
                Chessboard.set_obstacle(i, j)
            elif row[j] == "v":
                Chessboard.set_start(i, j)
            elif row[j] == "c":
                Chessboard.set_end(i, j)
    return
```

```
def grade_chessboard():
    global Chessboard
    stack = []
    heapq.heappush(stack, (0, *Chessboard.start))
    while stack:
        steps, row, col = heapq.heappop(stack)
        for field in Chessboard.rook_fields(row, col):
            field_steps = Chessboard.get_steps(*field)
            if field_steps <= steps + 1:
                continue
            else:
                Chessboard.set_steps(*field, steps+1)
                heapq.heappush(stack, (steps+1, *field))
        Chessboard.print()
    return


def main():
    global Chessboard
    read_chessboard()
    grade_chessboard()
    steps_to_end = Chessboard.get_steps(*Chessboard.end)
    if steps_to_end == INF:
        print(-1)
    else:
        print(steps_to_end)
    return


if __name__ == '__main__':
    main()

---
........
........
.v..x...
...x.x..
..x...xx
.x......
x....c..
........
```

Domino

Toto je typická úloha na backtracking (vzpomeňte na Sudoku). Proto nebudu mluvit o řešení, ale o dvou technických detailech:

- "obojakosti" kostek domina: kostka AB je zároveň také kostkou BA.
- vícenásobných výskytech některých kostek

Tady to chce dobré designové rozhodnutí, jedna z možností je ukládat kostky domina do matice 7 x 7 (0-6 teček). Pokud se rozhodnete špatně, váš kód bude úpět.

Cykly v grafu

Ve zkratce: jezdíme po grafu, dokud nenatrefíme na konec nebo na uzel, kde jsme již byli.

Piškvorky

Toto je jednoduchá hra, a chceme najít optimální strategii. Kam dát následující kroužek?

Min-max strategie:

Data: seznam znaků x, o, . o délce 9 (nechceme 2D pole)

Hodnocení: Pokud se mřížce nachází trojice xxx, plus nekonečno. Pokud se v mřížce nachází trojice ooo, mínus nekonečno. Jinak 0.

Detekce: Pro každý znak najdeme všechna místa, kde se nachází, a porovnáme se seznamem 8 možných trojic:

```
INFINITY = 1
MINUS_INFINITY = - INFINITY

empty_grid = ["."] * 9
triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8}, {0,
4, 8}, {2, 4, 6}]


def find_triple(grid, sign):
    positions = {i for i in range(9) if grid[i] == sign}
    result = [t for t in triples if t.issubset(positions)]
    return result


def grade(grid) -> int:
    if find_triple(grid, "x"):
        return INFINITY
    elif find_triple(grid, "o"):
        return MINUS_INFINITY
```

```
    else:
        return 0


def get_sign(player: bool) -> str:
    return "o" if player else "x"
```

Tisk mřížky:

```
def print_grid(grid) -> None:
    print()
    for i in range(3):
        for j in range(3):
            print(grid[3*i + j], end = " ")
        print()
    print(grade(grid))
    print()
```

Strom:

```
class Node:
    def __init__(self, grid):
        self.grid = grid
        self.df = self.grid.count(".")
        self.player = (9 - self.df) % 2
        self.score = grade(self.grid)
        self.children = []
```

Stavíme strom:

Musíme dát pozor na kombinatoriku. Mnohé pozice můžeme dosáhnout několika způsoby, takže pro pozici, kterou jsme již viděli, použijeme existující uzel stromu:

```
def build_tree(start_grid:list[int] = empty_grid) -> Node:
    node_dict = {}
    root = Node(start_grid)
    queue = deque([root])
    node_dict[tuple(start_grid)] = root
    n_nodes = 1
    while queue:
        node = queue.popleft()
        if node.score != 0:
            continue
        sign = get_sign(node.player)
        for pos in range(9):
            if node.grid[pos] == ".":
                new_grid = node.grid.copy()
                new_grid[pos] = sign
                if tuple(new_grid) in node_dict:
                    new_node = node_dict[tuple(new_grid)]
                else:
                    new_node = Node(new_grid)
                    node_dict[tuple(new_grid)] = new_node
```

```
                    queue.append(new_node)
                    n_nodes += 1
                node.children.append(new_node)
    print(n_nodes)
    return root
```

A konečně min-max:

```
class Choice:
    def __init__(self, choice, value):
        self.choice = choice
        self.value = value


def minmax(node):
    if not node.children:
        return Choice("end", node.score)

    choices = [minmax(c) for c in node.children]
    if node.player == 0:
        max_result = max(c.value for c in choices)
        max_choices = [i for i in range(len(node.children)) if choices[i].value
== max_result]
        return Choice(max_choices, max_result)
    else:
        min_result = min(c.value for c in choices)
        min_choices = [i for i in range(len(node.children)) if choices[i].value
== min_result]
        return Choice(min_choices, min_result)


def play(start_grid = empty_grid):
    tree = build_tree(start_grid)
    current_node = tree
    while True:
        print_grid(current_node.grid)
        choice = minmax(current_node)
        if choice.choice == "end":
            print("Game finished")
            break
        select = random.choice(choice.choice)
        current_node = current_node.children[select]
```

Výsledný program:

```
from collections import deque
import random

INFINITY = 1
MINUS_INFINITY = - INFINITY

empty_grid = ["."] * 9
```

```
triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8}, {0,
4, 8}, {2, 4, 6}]


def find_triple(grid, sign):
    positions = {i for i in range(9) if grid[i] == sign}
    result = [t for t in triples if t.issubset(positions)]
    return result


def grade(grid) -> int:
    if find_triple(grid, "x"):
        return INFINITY
    elif find_triple(grid, "o"):
        return MINUS_INFINITY
    else:
        return 0


def get_sign(player: bool) -> str:
    return "o" if player else "x"


def print_grid(grid) -> None:
    print()
    for i in range(3):
        for j in range(3):
            print(grid[3*i + j], end = " ")
        print()
    print(grade(grid))
    print()


class Node:
    def __init__(self, grid):
        self.grid = grid
        self.df = self.grid.count(".")
        self.player = (9 - self.df) % 2
        self.score = grade(self.grid)
        self.children = []


def build_tree(start_grid:list[int] = empty_grid) -> Node:
    node_dict = {}
    root = Node(start_grid)
    queue = deque([root])
    node_dict[tuple(start_grid)] = root
    n_nodes = 1
    while queue:
        node = queue.popleft()
        if node.score != 0:
            continue
        sign = get_sign(node.player)
        for pos in range(9):
            if node.grid[pos] == ".":
                new_grid = node.grid.copy()
```

```
                new_grid[pos] = sign
                if tuple(new_grid) in node_dict:
                    new_node = node_dict[tuple(new_grid)]
                else:
                    new_node = Node(new_grid)
                    node_dict[tuple(new_grid)] = new_node
                    queue.append(new_node)
                    n_nodes += 1
                node.children.append(new_node)
    print(n_nodes)
    return root


class Choice:
    def __init__(self, choice, value):
        self.choice = choice
        self.value = value

    def __str__(self):
        return f"Choosing {self.choice} to reach {self.value}"


def minmax(node):
    if not node.children:
        return Choice("end", node.score)

    choices = [minmax(c) for c in node.children]
    if node.player == 0:
        max_result = max(c.value for c in choices)
        max_choices = [i for i in range(len(node.children)) if choices[i].value
== max_result]
        return Choice(max_choices, max_result)
    else:
        min_result = min(c.value for c in choices)
        min_choices = [i for i in range(len(node.children)) if choices[i].value
== min_result]
        return Choice(min_choices, min_result)


def play(start_grid = empty_grid):
    tree = build_tree(start_grid)
    current_node = tree
    while True:
        print_grid(current_node.grid)
        choice = minmax(current_node)
        if choice.choice == "end":
            print("Game finished")
            break
        select = random.choice(choice.choice)
        current_node = current_node.children[select]


def main() -> None:
    start_grid = input().split()
    play(start_grid)
```

```
if __name__ == "__main__":
    main()
```

Kód se chová v některých situacích divně:

```
. * * o . . . . .
200

. * *
o . .
. . .
0


o * *
o . .
. . .
0


o * *
o * .
. . .
0


o * *
o * .
o . .
-1

Game finished

Process finished with exit code 0
```

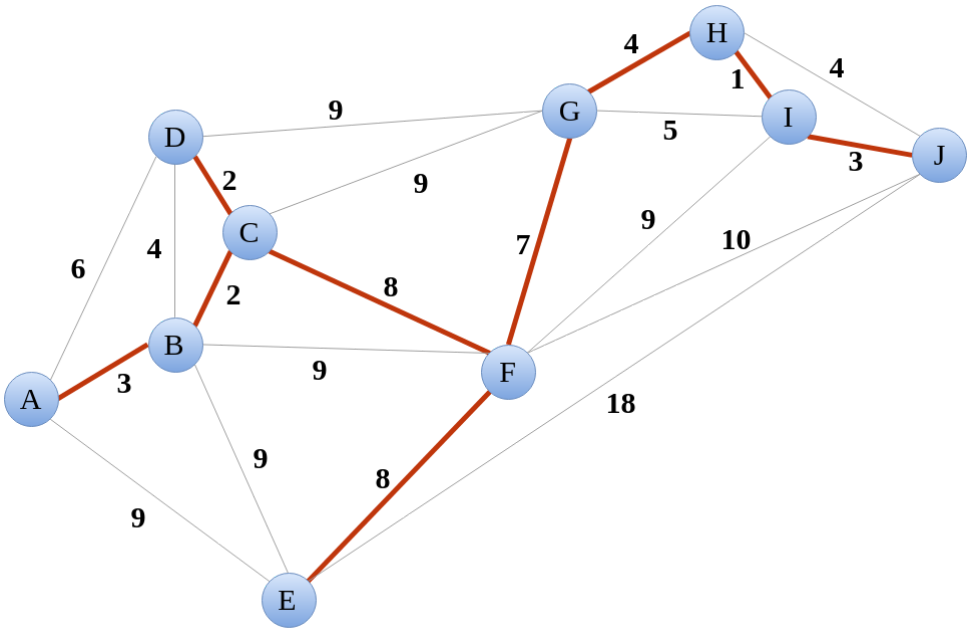
To je způsobeno tím, že pozice, ze které vycházíme, je pro hráče * prohrávající a tedy všechno, co udělá, je stejně účinné. Abychom situaci trošku vylepšili, můžeme upřednostnit agresivní řešení - tedy cesty, které vedou k výhře rychleji:

```
def minmax(node):
    aggresivity = 0.8
    if not node.children:
        return Choice("end", node.score)

    choices = [minmax(c) for c in node.children]
    if node.player == 0:
        max_result = max(c.value for c in choices)
        max_choices = [i for i in range(len(node.children)) if choices[i].value
== max_result]
        return Choice(max_choices, aggresivity * max_result)
    else:
```

```
        min_result = min(c.value for c in choices)
        min_choices = [i for i in range(len(node.children)) if choices[i].value
== min_result]
        return Choice(min_choices, aggresivity * min_result)
```

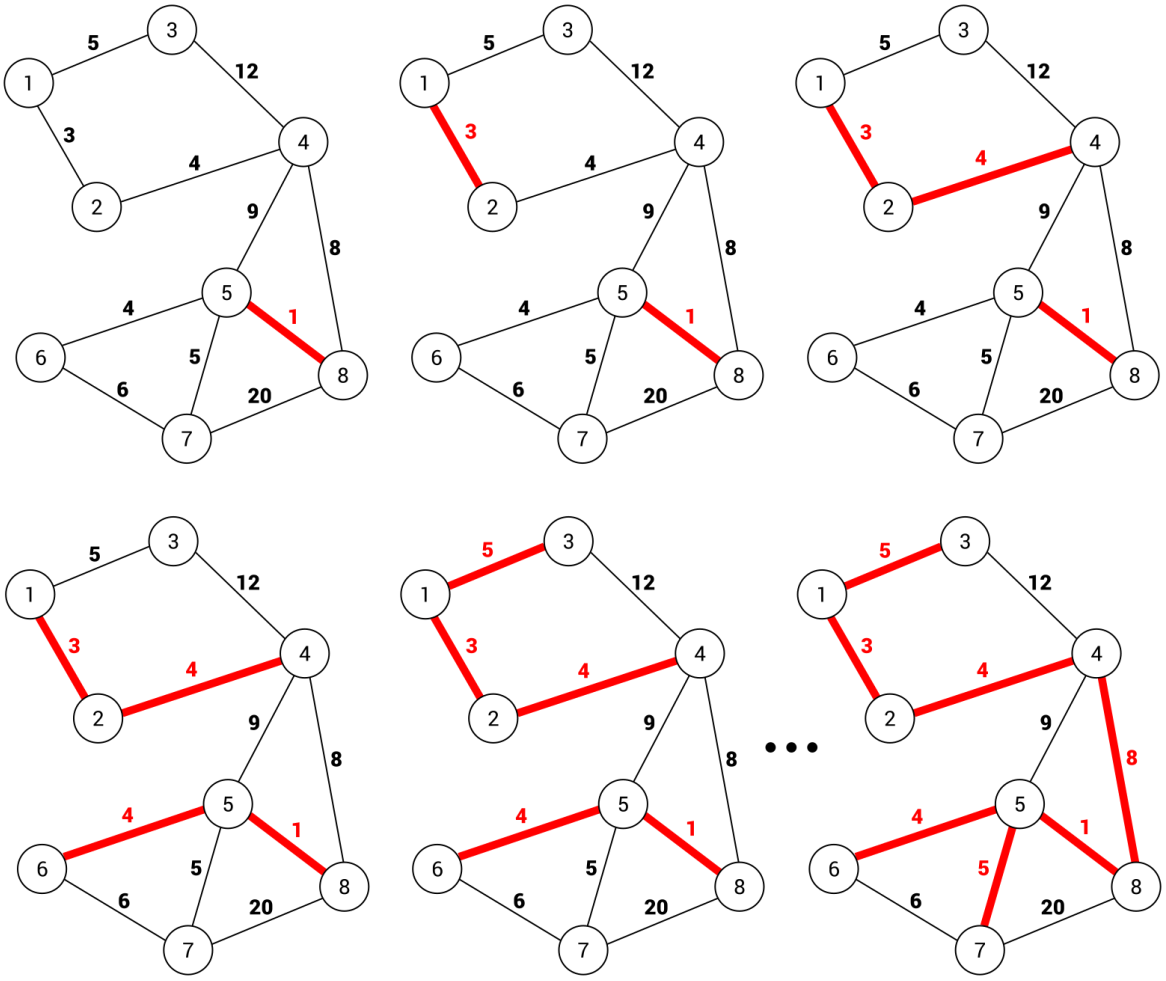
Grafové algoritmy 1:

Minimální kostra - Minimum spanning tree



Kruskalův algoritmus:

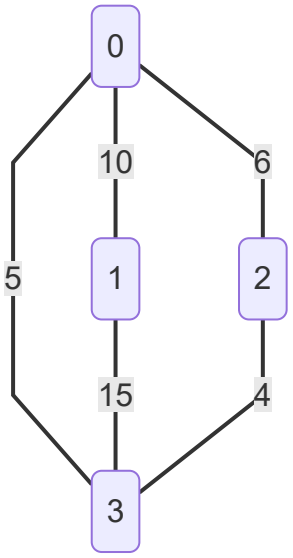
- Každý vrchol začíná jako samostatná komponenta
- Komponenty vzájemně spojujeme nejlehčí hranou, ale tak, abychom nevytvářeli cykly.



Definice grafu (kód v `Ex12/kruskal_mst.py`)

```
class Graph:

    def __init__(self, vertices):
        self.n_vertices = vertices  # No. of vertices
        self.graph = []  # triples from, to, weight

    def add_edge(self, start, end, weight):
        self.graph.append([start, end, weight])


def main() -> None:
    g = Graph(4)
    g.add_edge(0, 1, 10)
    g.add_edge(0, 2, 6)
    g.add_edge(0, 3, 5)
    g.add_edge(1, 3, 15)
    g.add_edge(2, 3, 4)

    g.kruskal_mst()


if __name__ == '__main__':
    main()
```

Výchozí graf:




```
# A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])
```

Hledáme, ke které komponentě grafu patří vrchol `i`. Je-li samostatnou komponentou, vracíme samotný vrchol. Pokud ne, rekurzivně prohledáváme předky vrcholu.

```
# A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
```

Sjednocení komponent grafu: "Věšíme" menší na větší, `rank` je počet spojených prvků, není nutně rovný výšce stromu.

Výsledný algoritmus:

```
    def kruskal_mst(self):
```

```
        result = []  # This will store the resultant MST

        # An index variable, used for sorted edges
        i_sorted_edges = 0

        # An index variable, used for result[]
        i_result = 0

        # Step 1:  Sort all the edges in
        # non-decreasing order of their
        # weight.  If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.n_vertices):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while i_result < self.n_vertices - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i_sorted_edges]
            i_sorted_edges = i_sorted_edges + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge doesn't
            #  cause cycle, include it in result
            #  and increment the indexof result
            # for next edge
            if x != y:
                i_result = i_result + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        print("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)
```

Výsledek pro náš graf:

```
Edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19
```