

## Programování 2

---

# 12. cvičení, 04-05-2023

---

tags: Programování 2, čtvrtek 1 čtvrtek 2

## Farní oznamy

---

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.

### 2. Domácí úkoly

- Piškvorky - lehké, ale dalo se vymyslet i těžké řešení a dokonce i nefungující řešení
- Domino - velice typická úloha na prohlédávání do hloubky.
- Řez stromu - těžší úloha, potřebovala promyslet postup.

### 3. Zápočtový program:

Většina z vás má už domluvené téma zápočtového programu.

### 4. Průběh semestru:

- Toto je předposlední praktické cvičení
- 18. 5. bude zápočtový test:
  - Přijdete na cvičení v obvyklém termínu 10:40, resp. 15:40
  - 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
- Prohlédávání stavového prostoru: 8 dam
- Grafy a grafové algoritmy

# Piškvorky

Opakování a doplnění z minula



Na minulém cvičení jsme dospěli k následující implementaci (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, plus nekonečno. Jinak 0.

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

```
1 INFINITY = 1
2 MINUS_INFINITY = - INFINITY
3
4 empty_grid = ["."] * 9
5 triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8},
6             {0, 4, 8}, {2, 4, 6}]
7
8 def find_triple(grid, sign):
9     positions = {i for i in range(9) if grid[i] == sign}
10    result = [t for t in triples if t.issubset(positions)]
11    return result
12
13
14 def grade(grid) -> int:
15     if find_triple(grid, "x"):
16         return INFINITY
17     elif find_triple(grid, "o"):
18         return MINUS_INFINITY
19     else:
20         return 0
21
22
23 def get_sign(player: bool) -> str:
24     return "o" if player else "x"
```

Tisk mřížky:

```

1 def print_grid(grid) -> None:
2     print()
3     for i in range(3):
4         for j in range(3):
5             print(grid[3*i + j], end = " ")
6         print()
7     print(grade(grid))
8     print()

```

Strom:

```

1 class Node:
2     def __init__(self, grid):
3         self.grid = grid
4         self.df = self.grid.count(".")
5         self.player = (9 - self.df) % 2
6         self.score = grade(self.grid)
7         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:

```

1 def build_tree(start_grid:list[int] = empty_grid) -> Node:
2     node_dict = {}
3     root = Node(start_grid)
4     queue = deque([root])
5     node_dict[tuple(start_grid)] = root
6     n_nodes = 1
7     while queue:
8         node = queue.popleft()
9         if node.score != 0:
10             continue
11         sign = get_sign(node.player)
12         for pos in range(9):
13             if node.grid[pos] == ".":
14                 new_grid = node.grid.copy()
15                 new_grid[pos] = sign
16                 if tuple(new_grid) in node_dict:
17                     new_node = node_dict[tuple(new_grid)]
18                 else:
19                     new_node = Node(new_grid)
20                     node_dict[tuple(new_grid)] = new_node
21                 queue.append(new_node)
22                 n_nodes += 1

```

```

23         node.children.append(new_node)
24     print(n_nodes)
25     return root
26

```

A konečně min-max:

```

1  class Choice:
2      def __init__(self, choice, value):
3          self.choice = choice
4          self.value = value
5
6
7  def minmax(node):
8      if not node.children:
9          return Choice("end", node.score)
10
11     choices = [minmax(c) for c in node.children]
12     if node.player == 0:
13         max_result = max(c.value for c in choices)
14         max_choices = [i for i in range(len(node.children)) if choices[i].value
15 == max_result]
16         return Choice(max_choices, max_result)
17     else:
18         min_result = min(c.value for c in choices)
19         min_choices = [i for i in range(len(node.children)) if choices[i].value
20 == min_result]
21         return Choice(min_choices, min_result)
22
23 def play(start_grid = empty_grid):
24     tree = build_tree(start_grid)
25     current_node = tree
26     while True:
27         print_grid(current_node.grid)
28         choice = minmax(current_node)
29         if choice.choice == "end":
30             print("Game finished")
31             break
32         select = random.choice(choice.choice)
33         current_node = current_node.children[select]

```

Výsledný program:

```

1  from collections import deque
2  import random
3
4  INFINITY = 1
5  MINUS_INFINITY = - INFINITY
6

```

```

7 empty_grid = ["."] * 9
8 triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8},
9 {0, 4, 8}, {2, 4, 6}]
10
11 def find_triple(grid, sign):
12     positions = {i for i in range(9) if grid[i] == sign}
13     result = [t for t in triples if t.issubset(positions)]
14     return result
15
16
17 def grade(grid) -> int:
18     if find_triple(grid, "x"):
19         return INFINITY
20     elif find_triple(grid, "o"):
21         return MINUS_INFINITY
22     else:
23         return 0
24
25
26 def get_sign(player: bool) -> str:
27     return "o" if player else "x"
28
29
30 def print_grid(grid) -> None:
31     print()
32     for i in range(3):
33         for j in range(3):
34             print(grid[3*i + j], end = " ")
35         print()
36     print(grade(grid))
37     print()
38
39
40 class Node:
41     def __init__(self, grid):
42         self.grid = grid
43         self.df = self.grid.count(".")
44         self.player = (9 - self.df) % 2
45         self.score = grade(self.grid)
46         self.children = []
47
48
49 def build_tree(start_grid: list[int] = empty_grid) -> Node:
50     node_dict = {}
51     root = Node(start_grid)
52     queue = deque([root])
53     node_dict[tuple(start_grid)] = root
54     n_nodes = 1
55     while queue:
56         node = queue.popleft()
57         if node.score != 0:

```

```

58         continue
59     sign = get_sign(node.player)
60     for pos in range(9):
61         if node.grid[pos] == ".":
62             new_grid = node.grid.copy()
63             new_grid[pos] = sign
64             if tuple(new_grid) in node_dict:
65                 new_node = node_dict[tuple(new_grid)]
66             else:
67                 new_node = Node(new_grid)
68                 node_dict[tuple(new_grid)] = new_node
69                 queue.append(new_node)
70                 n_nodes += 1
71             node.children.append(new_node)
72     print(n_nodes)
73     return root
74
75
76 class Choice:
77     def __init__(self, choice, value):
78         self.choice = choice
79         self.value = value
80
81     def __str__(self):
82         return f"Choosing {self.choice} to reach {self.value}"
83
84
85 def minmax(node):
86     if not node.children:
87         return Choice("end", node.score)
88
89     choices = [minmax(c) for c in node.children]
90     if node.player == 0:
91         max_result = max(c.value for c in choices)
92         max_choices = [i for i in range(len(node.children)) if
93 choices[i].value == max_result]
94         return Choice(max_choices, max_result)
95     else:
96         min_result = min(c.value for c in choices)
97         min_choices = [i for i in range(len(node.children)) if
98 choices[i].value == min_result]
99         return Choice(min_choices, min_result)
100
101 def play(start_grid = empty_grid):
102     tree = build_tree(start_grid)
103     current_node = tree
104     while True:
105         print_grid(current_node.grid)
106         choice = minmax(current_node)
107         if choice.choice == "end":
108             print("Game finished")

```

```

108         break
109         select = random.choice(choice.choice)
110         current_node = current_node.children[select]
111
112
113     def main() -> None:
114         start_grid = input().split()
115         play(start_grid)
116
117
118     if __name__ == "__main__":
119         main()
120

```

Zjistili jsme, že se kód chová v některých situacích divně:

```

1  . * * o . . . . .
2  200
3
4  . * *
5  o . .
6  . . .
7  0
8
9
10 o * *
11 o . .
12 . . .
13 0
14
15
16 o * *
17 o * .
18 . . .
19 0
20
21
22 o * *
23 o * .
24 o . .
25 -1
26
27 Game finished
28
29 Process finished with exit code 0
30

```

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

```

1 def minmax(node):
2     agresivity = 0.8
3     if not node.children:
4         return Choice("end", node.score)
5
6     choices = [minmax(c) for c in node.children]
7     if node.player == 0:
8         max_result = max(c.value for c in choices)
9         max_choices = [i for i in range(len(node.children)) if choices[i].value
== max_result]
10        return Choice(max_choices, agresivity * max_result)
11    else:
12        min_result = min(c.value for c in choices)
13        min_choices = [i for i in range(len(node.children)) if choices[i].value
== min_result]
14        return Choice(min_choices, agresivity * min_result)
15
16

```

## Domácí úkoly

### 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.

### Piškvorková remíza

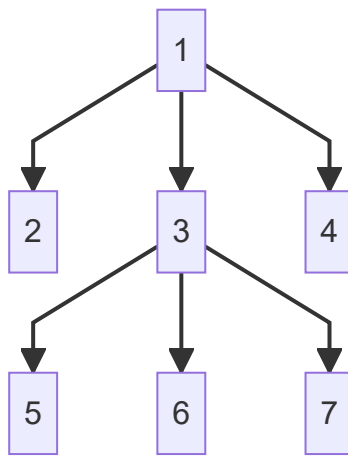
Základní postup:

1. Funkce pro kontrolu správnosti mřížky
2. Systém pro generování mřížek

Toto stačí a je to velmi obecný princip.

### Řezání stromu





Řez = odstranění hrany

Váha řezu =  $\max(\text{váha levého podstromu}, \text{váha pravého podstromu})$

K této úloze jsou kromě toho vyloženy podlé testy.

**Řešení:** Fungují běžné postupy s rekurzivním i nerekurzivním prohledáváním, jenomže jsou  $O(n^2)$ . Řešení, které je  $O(n)$ , se zakládá na "olupování listů".

```

1
2 class Vertex:
3     def __init__(self):
4         self.edges = set()
5         self.subtree_cost = 0
6
7     def add_edge(self, end, cost):
8         self.edges.add((end, cost))
9
10    def remove_edge(self, end, cost):
11        self.edges.remove((end, cost))
12
13    def add_subtree_cost(self, cost):
14        self.subtree_cost += cost
15
16    def get_subtree_cost(self):
17        return self.subtree_cost
18
19
20 n = int(input())
21 vertex_edges = [Vertex() for _ in range(n+1)]
22
23
24 def main() -> None:
25     global n
26     global vertex_edges
27     total_cost = 0
28     for i in range(n-1):
29         start, end, cost = [int(c) for c in input().split()]

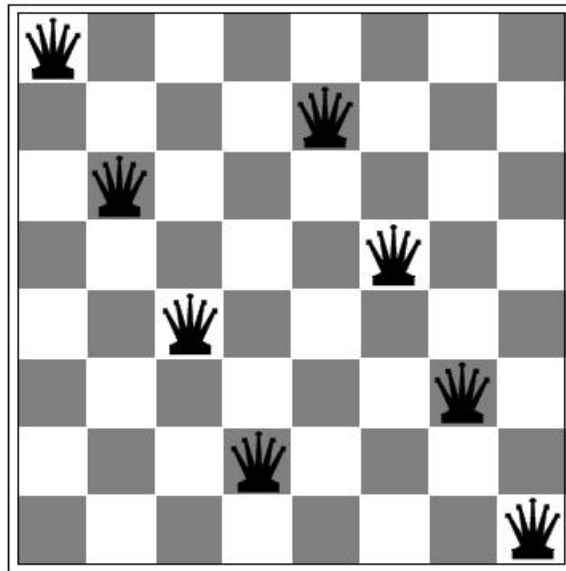
```

```

30     vertex_edges[start].add_edge(end, cost)
31     vertex_edges[end].add_edge(start, cost)
32     total_cost += cost
33     min_cost = 1.0e20
34     leaves = [i for i in range(1, n+1) if len(vertex_edges[i].edges) == 1]
35     while len(leaves) > 2:
36         new_leaves = set()
37         for leaf in leaves:
38             end, cost = vertex_edges[leaf].edges.pop()
39             leaf_cost = vertex_edges[leaf].get_subtree_cost()
40             cut_cost = max(total_cost - cost - leaf_cost, leaf_cost)
41             if cut_cost < min_cost:
42                 min_cost = cut_cost
43             vertex_edges[end].add_subtree_cost(leaf_cost + cost)
44             vertex_edges[end].remove_edge(leaf, cost)
45             if len(vertex_edges[end].edges) == 1:
46                 new_leaves.add(end)
47         leaves = new_leaves
48     if len(leaves) == 2:
49         current = leaves.pop()
50         last = leaves.pop()
51         while current != last:
52             other, cost = vertex_edges[current].edges.pop()
53             leaf_cost = vertex_edges[current].get_subtree_cost()
54             cut_cost = max(total_cost - cost - leaf_cost, leaf_cost)
55             if cut_cost < min_cost:
56                 min_cost = cut_cost
57             if total_cost - cost - leaf_cost < leaf_cost:
58                 break
59             vertex_edges[other].add_subtree_cost(leaf_cost + cost)
60             vertex_edges[other].remove_edge(current, cost)
61             current = other
62     print(min_cost)
63
64
65 if __name__ == "__main__":
66     main()
67

```

## Ještě rekurze: Problém osmi dam



### Řešení:

V každém řádku, sloupci a na každé levo-pravé a pravo-levé diagonále máme maximálně jednu dámu.

- Implementujeme šachovnici jako *slovník* s klíčem `(sloupec, radek)` a seznamem dam, které mají dané pole pod kontrolou.
- Pozice dam si pamatujeme v seznamu.
- Toto není optimální řešení, o možných zlepšeních si povíme.

Kód v `Ex12/eight_queens.py`

```

1 class Chessboard:
2     def __init__(self):
3         """Just create chessboard"""
4         self.chessboard = dict([(i,j),set()] for i, j in product(range(SIZE),
5 range(SIZE)))
6         self.queens = []
7
8     def is_in_range(self, k, l):
9         return (k,l) in self.chessboard.keys()
10
11     def is_available(self, i, j):
12         """Is this field available for a queen?"""
13         return len(self.chessboard[i,j]) == 0

```

Hodí se umět vytisknout šachovnici:

```

1     def print(self):
2         chart = [["_"] for _ in range(SIZE)] for _ in range(SIZE)]
3         for pos, occ in self.chessboard.items():
4             if len(occ) > 0:
5                 i, j = pos

```

```

6         chart[i][j] = "o"
7     for i, j in self.queens:
8         chart[i][j] = "o"
9     for i in range(SIZE):
10        print(*chart[i])
11
12 o o o o o o o _
13 o o o o o o o o
14 o o o o o o o o
15 o o o o o o o o
16 o o o o o o o o
17 o o o o o o o o
18 o o o o o o o o
19 o o o o o o o o

```

Další věcí, kterou budeme potřebovat, je funkce, která položí dámu na dané pole a zapamatuje si dámu kontrolovaná pole tak, že dámu půjde lehce odstranit.

První věcí je seznam polí, která kontroluje daná dáma: Od polohy dámy bookujeme pole v osmi směrech.

```

1     def queen_fields(self, i, j):
2         """Return a list of fields controlled by a queen at (i, j)"""
3         steps = [(s, t) for s, t in product([-1,0,1], repeat=2) if not s==t==0]
4         fields = set()
5         for s, t in steps:
6             k = i
7             l = j
8             while self.is_in_range(k, l):
9                 fields.add((k, l))
10                k = k + s
11                l = l + t
12        return fields

```

Umístění a zrušení dámy:

```

1     def place_queen(self, i, j):
2         """Place a new queen at i, j"""
3         self.queens.append((i,j))
4         for k,l in self.queen_fields(i, j):
5             self.chessboard[k, l].add((i,j))
6
7     def remove_queen(self):
8         """Remove most recently added queen"""
9         i, j = self.queens.pop()
10        for k, l in self.queen_fields(i, j):
11            try:
12                self.chessboard[k, l].remove((i,j))
13            except KeyError:
14                print(f"Error removing ({i=}, {j=} from {self.chessboard[k,
15}}")

```

Budeme postupně umísťovat dámy do sloupců šachovnice a hledat pozice, v nichž nebudou kolidovat. Prohledáváme do hloubky - když nic nenajdeme, vrátíme se o krok zpět.

```

1  def place_queens(k = 0):
2      global Chessboard
3      if k == 8:
4          print("\nSolution:")
5          Chessboard.print()
6          return 8
7      for i in range(SIZE):
8          if not Chessboard.is_available(k, i):
9              continue
10         Chessboard.place_queen(k,i)
11         place_queens(k+1)
12         Chessboard.remove_queen()
13     return k
14

```

Nakonec všechno sestavíme.

```

1  # Place SIZE queens on a chessboard such that
2  # 1. No pair of queens attack each other
3  # 2. Each field is under control of a queen
4
5  from itertools import product
6
7  SIZE = 8
8
9
10 @lambda cls: cls() # Create class instance immediately
11 class Chessboard:
12     def __init__(self):
13         """Just create chessboard"""
14         self.chessboard = dict([(i,j),set()] for i, j in product(range(SIZE),
15 range(SIZE)))
16         self.queens = []
17
18     def is_in_range(self, k, l):
19         return (k,l) in self.chessboard.keys()
20
21     def is_available(self, i, j):
22         """Is this field available for a queen?"""
23         return len(self.chessboard[i,j]) == 0
24
25     def queen_fields(self, i, j):
26         """Return a list of fields controlled by a queen at (i, j)"""
27         steps = [(s, t) for s, t in product([-1,0,1], repeat=2) if not s==t==0]
28         fields = set()
29         for s, t in steps:

```

```

29         k = i
30         l = j
31         while self.is_in_range(k, l):
32             fields.add((k, l))
33             k = k + s
34             l = l + t
35         return fields
36
37     def place_queen(self, i, j):
38         """Place a new queen at i, j"""
39         self.queens.append((i,j))
40         for k,l in self.queen_fields(i, j):
41             self.chessboard[k, l].add((i,j))
42
43     def remove_queen(self):
44         """Remove most recently added queen"""
45         i, j = self.queens.pop()
46         for k, l in self.queen_fields(i, j):
47             try:
48                 self.chessboard[k, l].remove((i,j))
49             except KeyError:
50                 print(f"Error removing ({i=}, {j=} from {self.chessboard[k,
51 l]}")
52
53     def print(self):
54         chart = [["_ " for _ in range(SIZE)] for _ in range(SIZE)]
55         for pos, occ in self.chessboard.items():
56             if len(occ) > 0:
57                 i, j = pos
58                 chart[i][j] = "o"
59         for i, j in self.queens:
60             chart[i][j] = "o"
61         for i in range(SIZE):
62             print(*chart[i])
63
64     def place_queens(k = 0):
65         global Chessboard
66         if k == 8:
67             print("\nSolution:")
68             Chessboard.print()
69             return 8
70         for i in range(SIZE):
71             if not Chessboard.is_available(k, i):
72                 continue
73             Chessboard.place_queen(k,i)
74             place_queens(k+1)
75             Chessboard.remove_queen()
76         return k
77
78
79     def main():

```

```

80     place_queens(0)
81
82
83     if __name__ == '__main__':
84         main()
85

```

Řešení je hodně, takže se u nalezeného řešení nazastavujeme a pokračujeme dál.

Vylepšení:

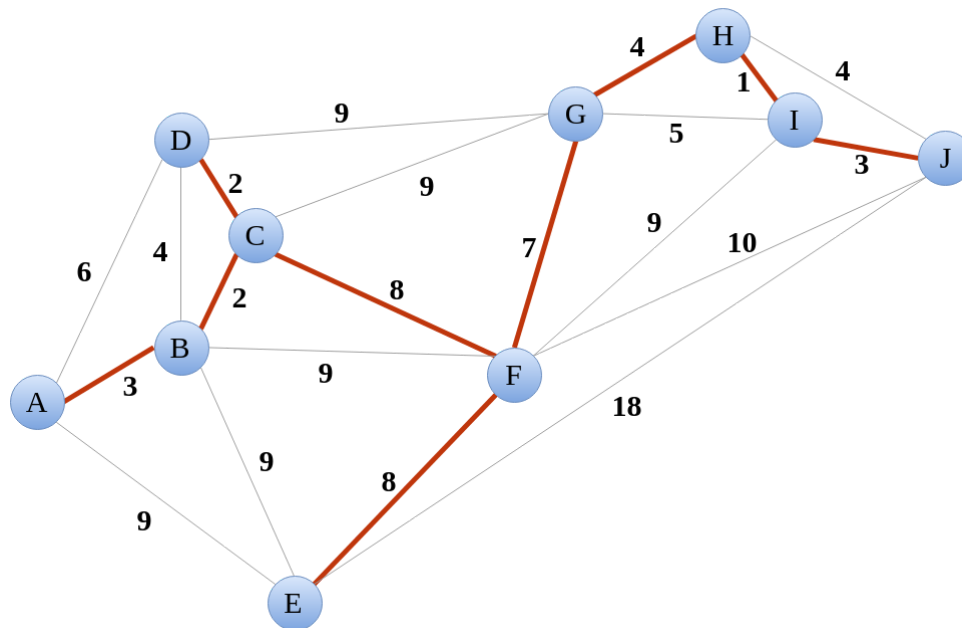
- Namísto obsazenosti polí šachovnice sledovat obsazení řádků, sloupů a diagonál.

Výhoda:

- 1D pole
- Unikátní obsazenost, takže stačí logická pole.
- Rychlejší nastavování a vyhledávání.

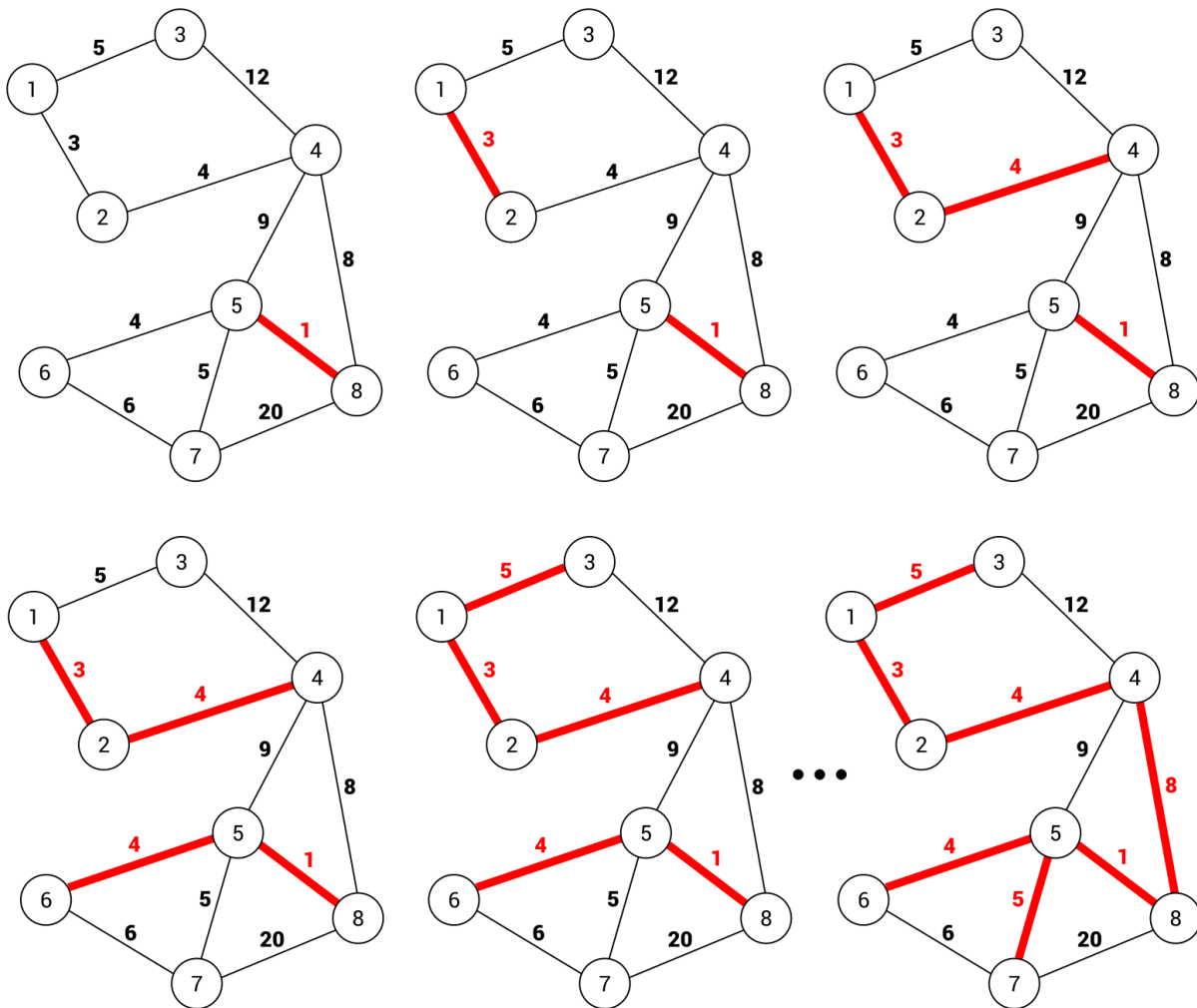
## Grafové algoritmy 1:

### 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`)

```

1  class Graph:
2
3      def __init__(self, vertices):
4          self.n_vertices = vertices # No. of vertices
5          self.graph = [] # triples from, to, weight
6
7      def add_edge(self, start, end, weight):
8          self.graph.append([start, end, weight])
9
10
11 def main() -> None:
12     g = Graph(4)
13     g.add_edge(0, 1, 10)
14     g.add_edge(0, 2, 6)
15     g.add_edge(0, 3, 5)
16     g.add_edge(1, 3, 15)
17     g.add_edge(2, 3, 4)
18
19     g.kruskal_mst()
20
21

```

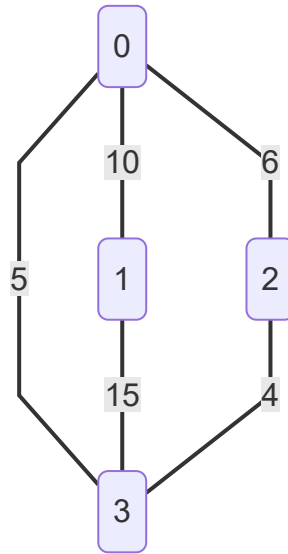


```

22 if __name__ == '__main__':
23     main()

```

Výchozí graf:



```

1 # A utility function to find set of an element i
2   # (uses path compression technique)
3   def find(self, parent, i):
4       if parent[i] == i:
5           return i
6       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.

```

1 # A function that does union of two sets of x and y
2   # (uses union by rank)
3   def union(self, parent, rank, x, y):
4       xroot = self.find(parent, x)
5       yroot = self.find(parent, y)
6
7       # Attach smaller rank tree under root of
8       # high rank tree (Union by Rank)
9       if rank[xroot] < rank[yroot]:
10          parent[xroot] = yroot
11       elif rank[xroot] > rank[yroot]:
12          parent[yroot] = xroot
13
14       # If ranks are same, then make one as root
15       # and increment its rank by one
16       else:
17          parent[yroot] = xroot
18          rank[xroot] += 1
19

```

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

Výsledný algoritmus:

```
1  def kruskal_mst(self):
2
3      result = [] # This will store the resultant MST
4
5      # An index variable, used for sorted edges
6      i_sorted_edges = 0
7
8      # An index variable, used for result[]
9      i_result = 0
10
11     # Step 1: Sort all the edges in
12     # non-decreasing order of their
13     # weight. If we are not allowed to change the
14     # given graph, we can create a copy of graph
15     self.graph = sorted(self.graph,
16                          key=lambda item: item[2])
17
18     parent = []
19     rank = []
20
21     # Create V subsets with single elements
22     for node in range(self.n_vertices):
23         parent.append(node)
24         rank.append(0)
25
26     # Number of edges to be taken is equal to V-1
27     while i_result < self.n_vertices - 1:
28
29         # Step 2: Pick the smallest edge and increment
30         # the index for next iteration
31         u, v, w = self.graph[i_sorted_edges]
32         i_sorted_edges = i_sorted_edges + 1
33         x = self.find(parent, u)
34         y = self.find(parent, v)
35
36         # If including this edge doesn't
37         # cause cycle, include it in result
38         # and increment the index of result
39         # for next edge
40         if x != y:
41             i_result = i_result + 1
42             result.append([u, v, w])
43             self.union(parent, rank, x, y)
44         # Else discard the edge
45
46     minimumCost = 0
47     print("Edges in the constructed MST")
```

```
48         for u, v, weight in result:
49             minimumCost += weight
50             print("%d -- %d == %d" % (u, v, weight))
51         print("Minimum Spanning Tree", minimumCost)
52
```

Výsledek pro náš graf:

```
1  Edges in the constructed MST
2  2 -- 3 == 4
3  0 -- 3 == 5
4  0 -- 1 == 10
5  Minimum Spanning Tree 19
6
```