# Základy algoritmizace

# 9. Hledání v grafech 2

doc. Ing. Jiří Vokřínek, Ph.D.
Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

# Základy algoritmizace

- Dnes:
  - Cesta v grafu
  - Stavové grafy

# Graf

- Graf je dvojice $G = <V, E>$, kde
  - $V$ je neprázdná množina vrcholů (uzlů, vertices, nodes)
  - $E \subseteq V \times V$ je množina uspořádaných dvojic vrcholů – hran (edges)

*U orientované hrany záleží na pořadí*

- Cesta v grafu
  - Taková posloupnost vrcholů, že mezi každými dvěma po sobě jdoucími je hrana

*Tj. sled*

  - Neopakují se v ní hrany

*Tj. tah*

  - Neopakují se v ní vrcholy
- Nejkratší cesta
  - Taková cesta, kde je počet hran (součet ohodnocení) minimální

# Reprezentace grafu

- Graf je abstraktní datový typ
- Reprezentujeme ho
  - Maticí sousednosti
  - Maticí incidence
  - **Seznamem sousedů**
- Operace
  - addVertex(x), removeVertex(x) – přidá/odebere vrchol
  - addEdge(x, y), removeEdge(x, y) – přidá/odebere hranu
  - adjacent(x, y) – test na sousednost vrcholů
  - neighbors(x) – vrací seznam sousedních vrcholů

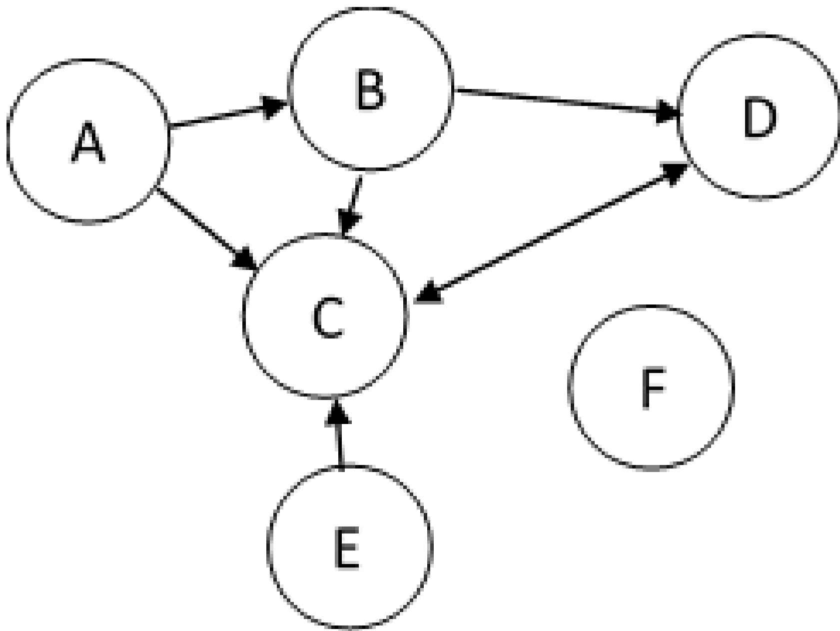
  - getVertexValue(x), setVertexValue(x, v) – hodnota vrcholu
  - getEdgeValue(x, y), setEdgeValue(x, y, v) – hodnota hrany

# Reprezentace grafu

- Seznam sousedů
  - Vrcholy jsou uloženy jako objekty
  - Umožňuje ukládat u vrcholů další data
  - Každý vrchol uchovává seznam sousedních vrcholů
  - Místo seznamu sousedů lze uchovávat seznam hran
    - Hrana je pak objektem uchovávajícím odkazy na uzly, které propojuje
    - Hrana může obsahovat další data, např. cenu
  - Lze jednoduše implementovat pomocí Dictionary
  - Vhodné pro dynamické „spojové“ struktury
  - Výhodné pro řídké grafy

# Reprezentace grafu

- Seznam sousedů – Dictionary




```
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['C'],
         'F': []}
```

# Cesta v grafu

- Příklad – hledání nejkratší cesty – rekurzivní varianta

*Z minulé přednášky*

```
def findShortestPath(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if not start in graph.keys():
        return None
    shortest = None
    for node in graph[start]:
        if node not in path:
            newpath = findShortestPath(graph, node, end, path)
            if newpath:
                if not shortest or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest
```

['A', 'C']

# Hledání nejkratších cest

- Předchozí algoritmus není příliš efektivní

*Proč?*

- Prohledává všechny cesty (do hloubky) a vybírá nejkratší

*Lze to lépe?*

- Příklad – hledání nejkratší cesty mezi městy
  - Graf reprezentujeme hranami mezi městy a jejich vzdáleností
  - Pro každý prohledávaný uzel si ukládáme město, vzdálenost od startu a cestu do něj
  - BFS s cenou cest – prioritní fronta
    - Vždy vyjmeme „nejlevnější" město a pokud to není cíl, prohledáme ho
    - Pro prohledané město vkládáme do fronty všechny jeho možné „následníky"
    - Pokud je fronta prázdná, končíme neúspěchem

# Hledání nejkratších cest

- Příklad – hledání nejkratší cesty mezi městy
  - Graf reprezentujeme hranami mezi městy a jejich vzdáleností

```
class CityGraph:
    def __init__(self):
        self.edges = {}
        self.distances = {}

    def addEdge(self, fromNode, toNode, distance):
        if fromNode not in self.edges.keys():
            self.edges[fromNode] = []
        if toNode not in self.edges.keys():
            self.edges[toNode] = []
        self.edges[fromNode].append(toNode)
        self.edges[toNode].append(fromNode)
        self.distances[(fromNode, toNode)] = distance
        self.distances[(toNode, fromNode)] = distance
```

# Cesta z (do) města

```
graph = CityGraph()
graph.addEdge('A', 'B', 10)
graph.addEdge('A', 'C', 20)
graph.addEdge('B', 'D', 15)
graph.addEdge('C', 'D', 30)
graph.addEdge('B', 'E', 50)
graph.addEdge('D', 'E', 30)
graph.addEdge('E', 'F', 5)
graph.addEdge('F', 'G', 2)

print(graph.edges)
print(graph.distances)
```

{'A': ['B', 'C'], 'F': ['E', 'G'], 'C': ['A', 'D'], 'G': ['F'], 'B': ['A', 'D', 'E'], 'E': ['B', 'D', 'F'], 'D': ['B', 'C', 'E']}

{('E', 'D'): 30, ('E', 'F'): 5, ('A', 'B'): 10, ('E', 'B'): 50, ('B', 'E'): 50, ('D', 'B'): 15, ('D', 'E'): 30, ('C', 'D'): 30, ('B', 'A'): 10, ('F', 'E'): 5, ('B', 'D'): 15, ('C', 'A'): 20, ('D', 'C'): 30, ('G', 'F'): 2, ('A', 'C'): 20, ('F', 'G'): 2}

# Hledání nejkratších cest

- Příklad – hledání nejkratší cesty mezi městy
  - Graf reprezentujeme hranami mezi městy a jejich vzdáleností
  - Pro každý prohledávaný uzel si ukládáme město, vzdálenost od startu a cestu do něj

```
class SearchNode:
    def __init__(self, vertex, cost, path):
        self.cost = cost
        self.path = path
        self.vertex = vertex
```

# Hledání nejkratších cest

- Příklad – hledání nejkratší cesty mezi městy
  - Graf reprezentujeme hranami mezi městy a jejich vzdáleností
  - Pro každý prohledávaný uzel si ukládáme město, vzdálenost od startu a cestu do něj
  - BFS s cenou cest – prioritní fronta
    - Vždy vyjmeme „nejlevnější“ město a pokud to není cíl, prohledáme ho
    - Pro prohledané město vkládáme do fronty všechny jeho možné „následníky“
    - Pokud je fronta prázdná, končíme neúspěchem

# Cesta z (do) města

```
def findBFSbyCost(self, start, end):
    queue = []
    queue.append(SearchNode(start,0,[start]))
    while queue:
        bestNode = None
        bestPrice = 0
        for sN in queue:
            if bestNode == None or bestPrice > sN.cost:
                bestNode = sN
                bestPrice = sN.cost
        queue.remove(bestNode)
        if bestNode.vertex == end:
            return bestNode.cost, bestNode.path
        for newNode in self.edges[bestNode.vertex]:
            if newNode in bestNode.path:
                continue
            path = bestNode.path + [newNode]
            cost = bestNode.cost + \
                self.distances[(bestNode.vertex, newNode)]
            queue.append(SearchNode(newNode, cost, path))
```

# Hledání nejkratších cest

- Příklad – hledání nejkratší cesty mezi městy
  - Graf reprezentujeme hranami mezi městy a jejich vzdáleností
  - Pro každý prohledávaný uzel si ukládáme město, vzdálenost od startu a cestu do něj
  - BFS s cenou cest – prioritní fronta
    - Vždy vyjmeme „nejlevnější" město a pokud to není cíl, prohledáme ho
    - Pro prohledané město vkládáme do fronty všechny jeho možné „následníky"
    - Pokud je fronta prázdná, končíme neúspěchem

```
print(graph.findBFSbyCost('A', 'F'))
```

(60, ['A', 'B', 'D', 'E', 'F'])

# Cesta z (do) města

```
def find
    queu
    queu
    while queue:
        bestNode = None
        bestPrice = 0
        for sN in queue:
            if bestNode == None or bestPrice > sN.cost:
                bestNode = sN
                bestPrice = sN.cost
        queue.remove(bestNode)
        if bestNode.ver
            return best
        for newNode in self.edges[bestNode.vertex]:
            if newNode in bestNode.path:
                continue
            path = bestNode.path + [newNode]
            cost = bestNode.cost + \
                                        ex, newNode)]
            queue.append(SearchNode(newNode, cost, path))
```

**Nejde to lépe?**

*Zamyslete se za domácí úkol ;-)*

**Kontrola cíle až při vyjmutí??**

**Je potřeba udržovat pro každý node celou cestu??**

**Co když stejný vertex již je ve frontě (jiná cena a cesta)??**

# Cesta z (do) města

```
def findBFSbyCost(self, start, end):
    queue = []
    queue.append(SearchNode(start,0,[start]))
    while queue:
        bestN
        bestP
        for s
            i                                          ].cost:


        queue.remove(bestNode)
        if bestN
            retu
        for newNode in self.edges[bestNode.vertex]:
            if new
                cc
            path = bestNode.path + [newNode]
            cost
                                                            lode)]
            queue.append(SearchNode(newNode, cost, path))
```

**Prioritní fronta**

**Ukončující podmínka**

**Zabránění cyklu**

**Rozbalení následovníků**

# Hledání cest

**Prioritní fronta**

**Ukončující podmínka**

**Zabránění cyklu**

**Rozbalení následovníků**

# Hledání cest

- Obecné hledání cesty v grafu

Prioritní **fronta**

**Ukončující podmínka**

**Zabránění cyklu**

**Rozbalení následovníků**

- Existují i výkonnější algoritmy

*Typicky využívají trojúhelníkovou nerovnost*

- Vhodné i pro grafy, které nejsou explicitně zadány
- Hledání řešení ve stavovém prostoru problému
  - Techniky umělé inteligence
  - Rozbalení následovníků odpovídá generování přípustných následujících stavů

# Prohledávání stavového prostoru

- Hledání řešení ve stavovém prostoru problému
  - Každý stav systému je reprezentován vrcholem grafu
  - Akce, která mění stav systému tvoří hranu v grafu
    - Hrany mohou být orientované
  - Řešení se hledá pomocí nalezení cesty v grafu
    - Tj. sekvencí akcí, které mění systém z počátečního stavu do cílového
    - Jakákoliv cesta řeší problém (např. algoritmus DFS)
    - Většinou nás zajímá nejkratší cesta (např. algoritmus BFS)
  - Do každého stavu může vést více cest
  - Graf může obsahovat cykly
  - Velikost grafu výrazně roste s množstvím aplikovatelných akcí
    - Množství hran, tedy i cest
    - Množství stavů
    - Délka cesty nemusí být omezena

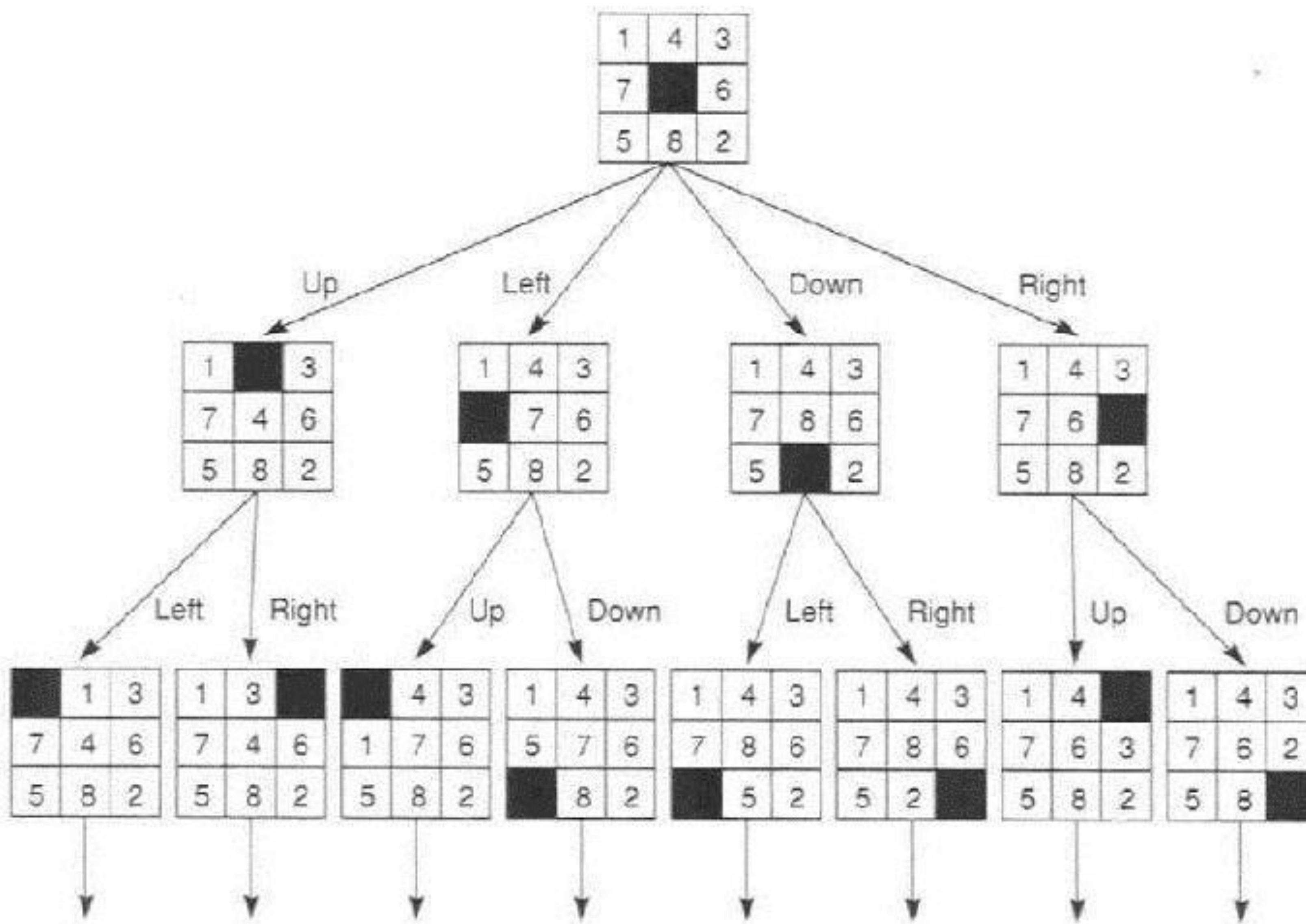
*Mnoho reálných problému nelze řešit prohledáním celého stavového prostoru*

# Prohledávání stavového prostoru

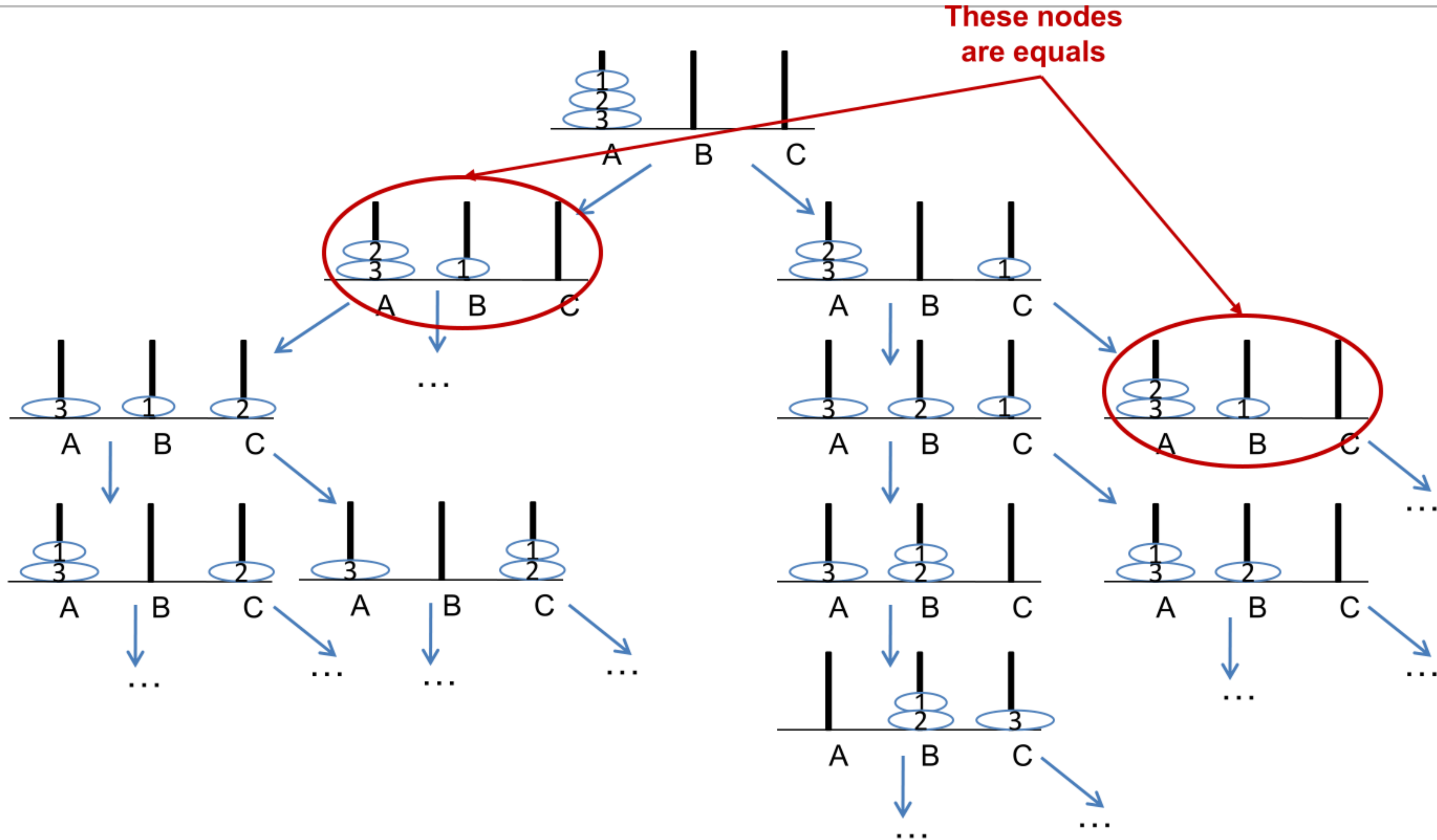
- Representace grafu
  - Vrchol obsahuje popis stavu systému
  - Exsituje *recept* (expand) jak generovat všechny následovníky, tedy všechny možné vývoje stavu za pomoci dostupných akcí
  - Vhodná reprezentace pomocí „spojového“ grafu

*Často generovaného dynamicky při prohledávání*

# Příklad stavového prostoru hry



# Příklad stavového prostoru hry



# Prohledávání stavového prostoru

- Algoritmus
  - Seznam OPEN – obsahuje stavy k prohledání
  - Seznam CLOSED – obsahuje již prohledané stavy
  - Dokud není seznam OPEN prázdný:
    - Vyjmi stav z OPEN
    - Pokud je to cíl, konec
    - Pokud ne, vlož ho do CLOSED a všechny jeho následovníky do OPEN
    - Opakuj od začátku
  - Podle chování seznamu OPEN lze měnit chování algoritmu:
    - Zásobník = prohledávání do šířky
    - Fronta = prohledávání do hloubky
    - Prioritní fronta = hledání nejkratší cesty (vzhledem k ceně akcí)

  - Hra dvou hráčů – střídají se tahy (pouze jeden můžeme ovlivnit)

# Prohledávání stavového prostoru

- Př. Piškvorky (3x3)
  - Počáteční stav = prázdné pole
  - Akce – střídání X a O (množství akcí v každém stavu je rovno počtu volných polí) – 9,8,7,6 …
  - 9! různých cest v grafu (362880) – horní odhad
  - Max. délka cesty je 9 (většinou kratší) – lze prohledat hrubou silou
  - 3^9 stavů (19683) – horní odhad (mnoho nedosažitelných)
  - Stavy se střídají podle toho, který hráč může ovlivnit výběr akce

- Příklad strategie
  - Vybírat akce, které eliminují nemožnost výhry
  - Vybírat akce, které vedou do stavu, kdy soupeř nemůže zabránit výhře

# Základy algoritmizace

- Dnes:
  - Hledání nejkratší cesty
  - Obecné prohledávání stavového prostoru

**Příště** složitost algoritmů a optimalizace kódu