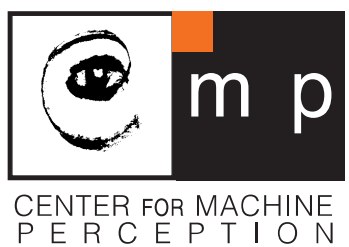
Prohledávání stavového prostoru

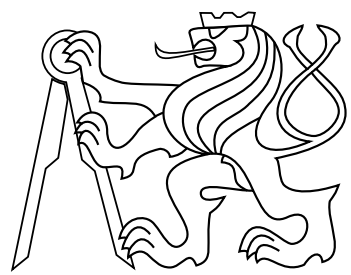
State space search

Jan Kybic

http://cmp.felk.cvut.cz/~kybic
kybic@fel.cvut.cz

2016–2017





Prohledávání stavového prostoru
State space search

Stavový prostor

- Množina stavů S.
- Počáteční stav $s_0 \in S$.
- Množina cílových stavů $T \subseteq S$.
- Seznam akcí A.
- Přechodová funkce $f : S \times A \to S$ (nemusí být úplná).

Hledáme sekvenci akcí $a_1, a_2, \ldots, a_N$ z A a sekvenci stavů začínající s_0 a splňující $s_i = f(s_{i-1}, a_i)$ z S, která převede systém do žádaného cílového stavu $s_N \in T$.

Prohledávání stavového prostoru
State space search

Stavový prostor

- Množina stavů S.
- Počáteční stav $s_0 \in S$.
- Množina cílových stavů $T \subseteq S$.
- Seznam akcí A.
- Přechodová funkce $f : S \times A \to S$ (nemusí být úplná).

Hledáme sekvenci akcí $a_1, a_2, \ldots, a_N$ z A a sekvenci stavů začínající s_0 a splňující $s_i = f(s_{i-1}, a_i)$ z S, která převede systém do žádaného cílového stavu $s_N \in T$.

Následník: $t \in \mathrm{succ}(s) \quad \Leftrightarrow \quad \exists a \in A;\ f(s, a) = t$

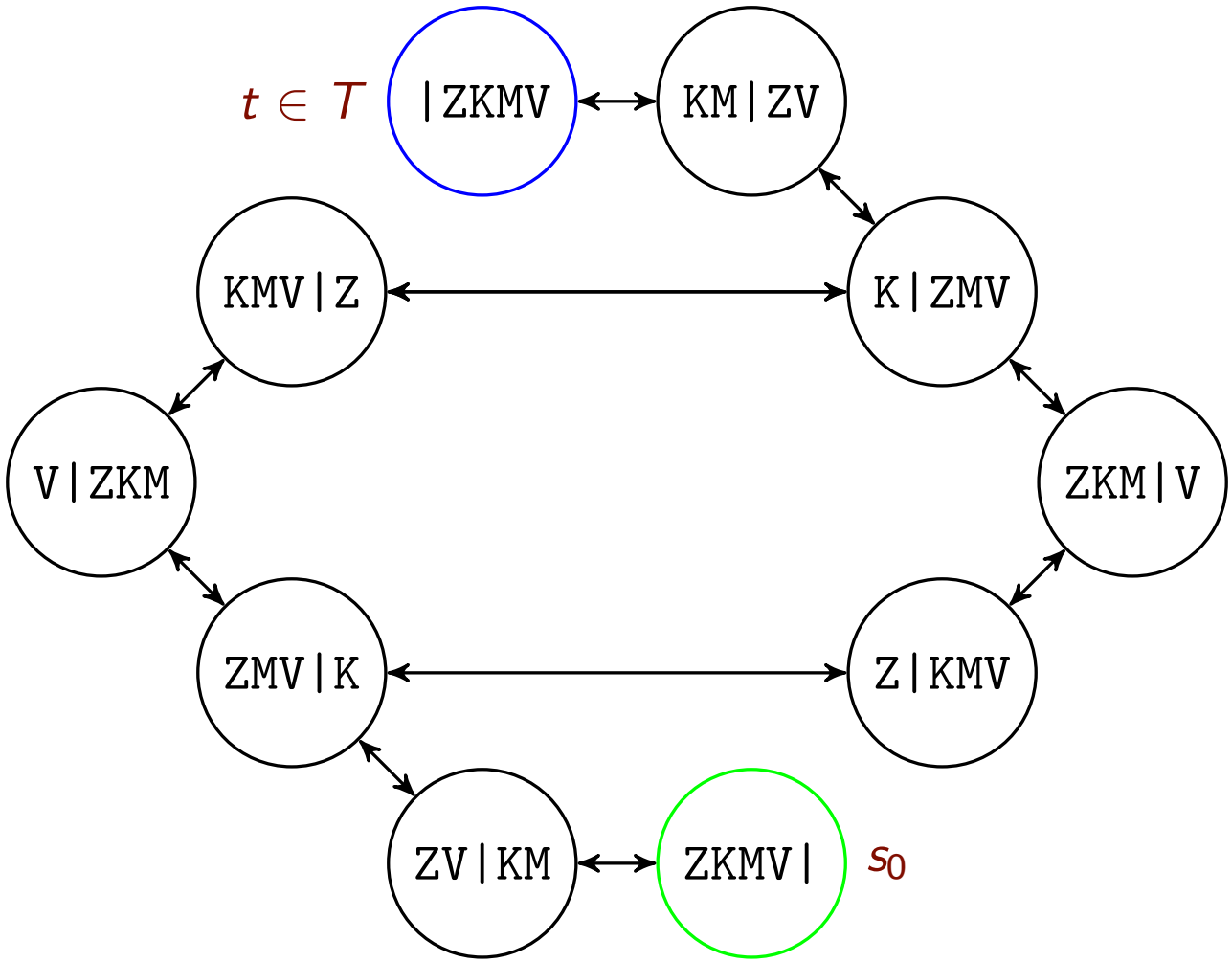
Příklady prohledávání stavového prostoru

- Plánování cesty (např. autem), procházení bludištěm
- Plánování pohybu robotů
- Plánování výrobních postupů
- Řešení hlavolamů (např. Rubikova kostka)
- Řešení diskrétních problémů s omezeními (např. Sudoku)

Koza, vlk, zelí

- Na břehu je koza, vlk, zelí, muž a loď.
- Do lodě se vejde jen muž a jedna věc.
- Bez dozoru muže sežere vlk kozu a koza zelí.
- Jak přesunout vše na druhý břeh?

Stavy a přechody



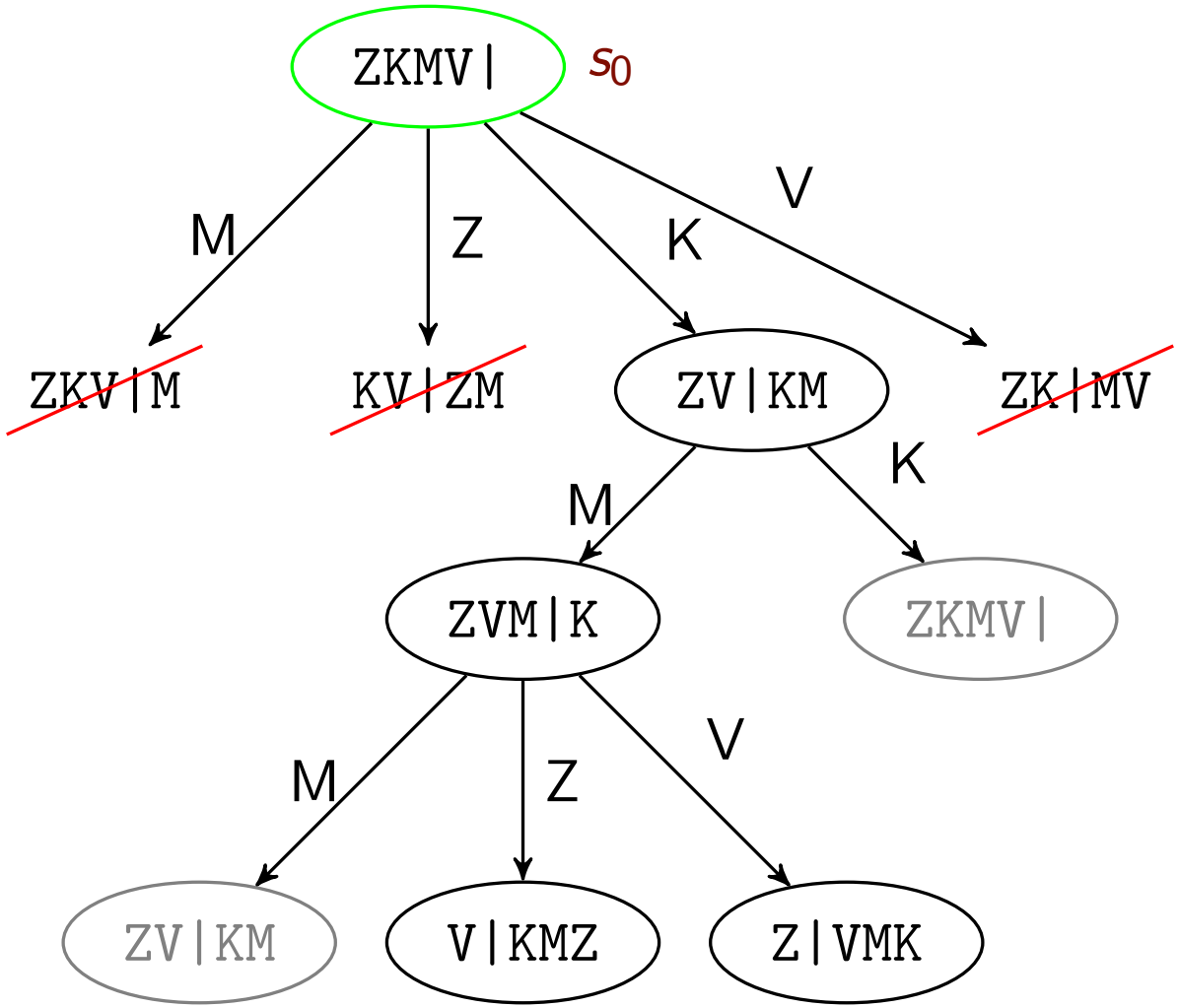
Hledáme cestu z počátečního do koncového stavu.

Prohledávání do hloubky
Depth first search (DFS), backtracking

V každém stavu zkusím něco udělat. Když už to nejde, tak se vrátím a zkusím něco jiného.

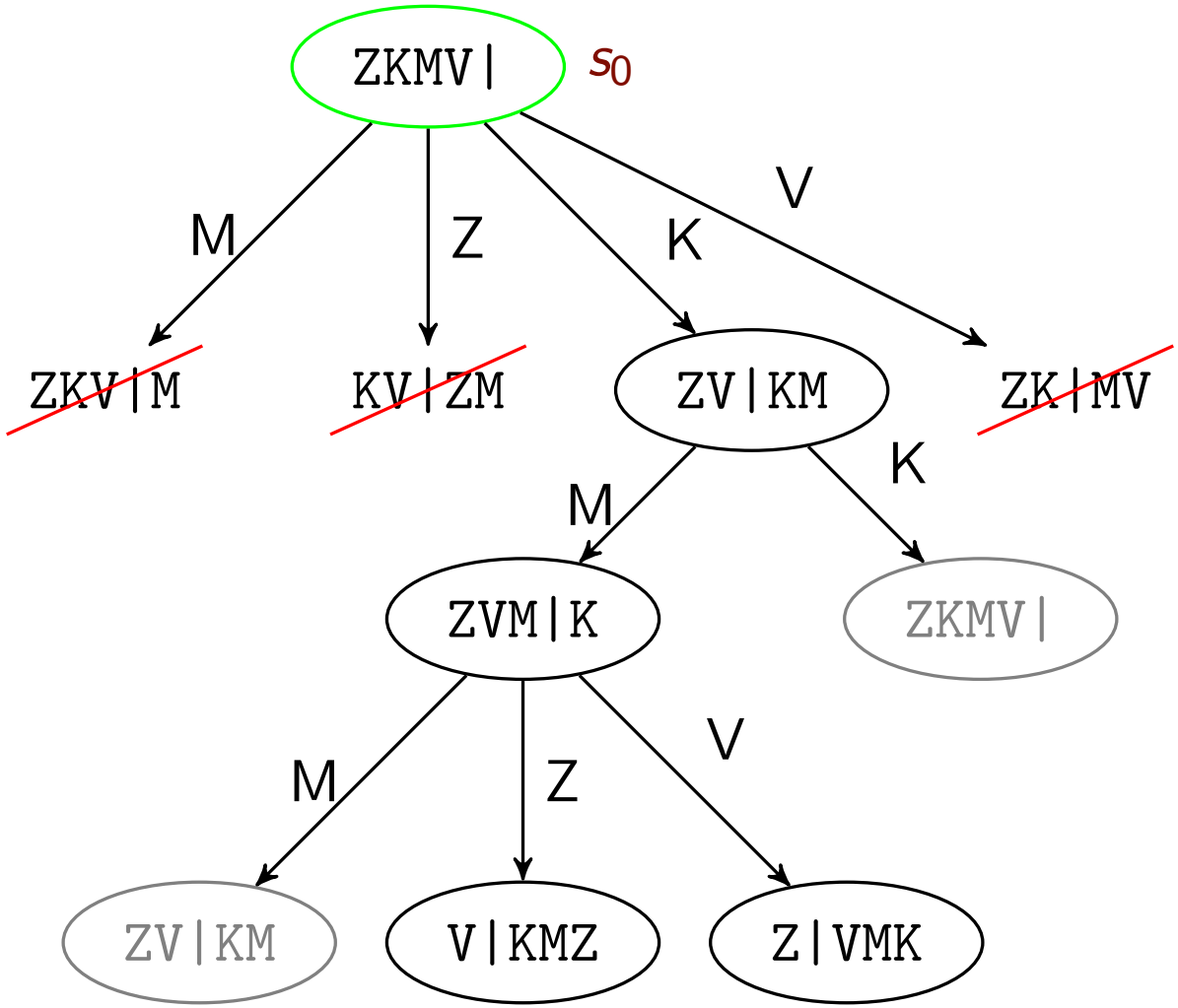
- Pokud jsem v cíli, hotovo.
- Existuje-li neprozkoumaný následník, rekurzivně ho prozkoumám.
- Pokud neexistuje, vrátím se o jednu akci zpět.
- Prozkoumané uzly si označuji.
- Pamatuji si, jak jsem se do aktuálního uzlu dostal.

Vyhledávací strom
(Search tree)



Vyhledávací strom $\neq$ graf přechodů.

Vyhledávací strom
(Search tree)



Vyhledávací strom $\neq$ graf přechodů. Zkuste dokreslit.

Prohledávání do hloubky — nalezení všech stavů

Prohledá stavový prostor do hloubky `maxdepth` počínaje stavem `problem`. Vrátí množinu nalezených stavů.

```
def all_states(problem,maxdepth=10):
  visited=set()
  def all_states_internal(state,depth):
    nonlocal visited
    if depth<maxdepth and state not in visited:
      visited|={state}  # označ stav jako navštívený
      for s in state.succ(): # všechny následovníci
          all_states_internal(s,depth+1)
  all_states_internal(problem,0)
  return visited
```

Prohledávání do hloubky — nalezení všech stavů

Prohledá stavový prostor do hloubky `maxdepth` počínaje stavem `problem`. Vrátí množinu nalezených stavů.

```
def all_states(problem,maxdepth=10):
  visited=set()
  def all_states_internal(state,depth):
    nonlocal visited
    if depth<maxdepth and state not in visited:
      visited|={state}  # označ stav jako navštívený
      for s in state.succ(): # všechny následovníci
          all_states_internal(s,depth+1)
  all_states_internal(problem,0)
  return visited


def print_all_states(problem,maxdepth=10):
  states=all_states(problem,maxdepth=maxdepth)
  for s in states:
    print(s," -> ",", ".join(map(str,s.succ())))
```

Soubor `dfs.py`.

Stavy

```
def print_states(): # definováno v modulu kozavlkzeli
  dfs.print_all_states(KozaVlkZeli())
```

```
import kozavlkzeli
kozavlkzeli.print_states() # volá 'all_states'
```

```
KMZ|V  ->  Z|KMV, K|MVZ
Z|KMV  ->  KMZ|V, MVZ|K
|KMVZ  ->  KM|VZ
K|MVZ  ->  KMZ|V, KMV|Z, KM|VZ
VZ|KM  ->  KMVZ|, MVZ|K
KMV|Z  ->  V|KMZ, K|MVZ
MVZ|K  ->  V|KMZ, Z|KMV, VZ|KM
KM|VZ  ->  |KMVZ, K|MVZ
KMVZ|  ->  VZ|KM
V|KMZ  ->  KMV|Z, MVZ|K
```

Soubor `kozavlkzeli.py`.

Prohledávání do hloubky — nalezení řešení

- Zastavím se po dosažení koncového stavu.
- Pamatuji si posloupnost stavů od s_0 do aktuálního.

```
def solve(problem,maxdepth=10):
  """ Vrátí posloupnost stavů nebo None """
  visited=set()
  def solve_internal(state,depth):
    nonlocal visited
    if depth<maxdepth and state not in visited:
      if state.final(): # koncový stav?
         return [state]
      visited|={state}  # označ stav jako navštívený
      for s in state.succ():
          r=solve_internal(s,depth+1)
          if r: # řešení nalezeno
            return [state]+r # méně efektivní
      return None # řešení nenalezeno
  return solve_internal(problem,0)
```

Příklad

```
def solve_kozavlkzeli(maxdepth=10):
  sol=dfs.solve(KozaVlkZeli(),maxdepth)
  if sol:
    print(" -> ".join(map(str,sol)))
  else:
    print("Řešení nenalezeno.")
```

```
import kozavlkzeli
kozavlkzeli.solve_kozavlkzeli()
```

```
ZKMV| -> ZV|KM -> ZMV|K -> V|ZKM -> KMV|Z ->
  K|ZMV -> KM|ZV -> |ZKMV
```

Implementace problému (1)

```
class KozaVlkZeli:
  """ stav problému = (levý_břeh,pravý_břeh), podmnožiny "MKVZ"
      akce = co je převáženo, tedy např. set("KZ")
  """

  def __init__(self,state=(frozenset("KVZM"),frozenset())):
    """ nastaví počáteční stav """
    self.state=state

  def final(self):
    """ je toto konečný stav? """
    return len(self.state[0])==0
```

Oddělili jsme problém a jeho řešení.

Soubor `kozavlkzeli.py`.

Implementace problému (2)

```
actions=list(map(frozenset,["KM","ZM","VM","M"])) # prom. třídy

def succ(self):
  """ vrátí přípustné akce v daném stavu """
  def safe(aset):
    """ Zkontroluj, zda 'aset' neobsahuje nepovolené dvojice """
    return not ( frozenset("KZ")<=aset  or frozenset("KV")<=aset )
  i=0 if "M" in self.state[0] else 1  # odkud jedeme
  successors=[]                       # následnící
  for aset in self.actions:           # možné akce
    if aset <= self.state[i]:         # ano, lze odvézt
      newstate=[None,None]
      newstate[i]=self.state[i] - aset
      newstate[1-i]=self.state[1-i] | aset
      if safe(newstate[i]):  # bezpečná situace?
         successors+=[KozaVlkZeli(tuple(newstate))]
  return successors
```

Implementace problému (3)

Speciální metody

```
# zamezení duplikací pro vkládání do množiny
def __eq__(self,a):
  return self.state==a.state

def __hash__(self):
  return hash(self.state)

# vrátí reprezentaci stavu jako řetězec
def __str__(self):
  return "".join(self.state[0])+"|"+"".join(self.state[1])
```

Implicitně rovnost objektů $\neq$ rovnost hodnot, ale identit.

Nerekurzivní prohledávání do hloubky

Nalezení všech stavů

- *Otevřené (open)* uzly ukládáme do zásobníku.

```
def all_states(problem,maxdepth=10):
  visited=set()                        # již navštívené stavy
  waiting=stack.Stack()                # waiting = (stav,úroveň)
  waiting.push((problem,0))
  while not waiting.is_empty():
    state,level=waiting.pop()   # nový stav ke zpracování
    if state not in visited:
      visited|={state}    # označ stav jako navštívený
      if level<maxdepth:
        for s in state.succ():
          waiting.push((s,level+1))
  return visited
```

Soubor `dfs2.py`.

Nerekurzivní prohledávání do hloubky (2)

Nalezení cílového stavu

```
def solve(problem,maxdepth=10):
    visited={}                          # stavy->předchůdci
    waiting=stack.Stack()               # (stav, předchůdce,úroveň)
    waiting.push((problem,None,0))
    while not waiting.is_empty():
      state,prev,level=waiting.pop()     # nový stav
      if state not in visited:          # je opravdu nový?
        visited[state]=prev             # přechůdce
        if state.final():               # koncový stav?
          return find_path(state,visited)
        if level<maxdepth:
          for s in state.succ():
            waiting.push((s,state,level+1))
    return None                        # řešení nenalezeno
```

Soubor `dfs2.py`.

Nerekurzivní prohledávání do hloubky (3)

Nalezení cesty z cíle na začátek.

```
def find_path(state,visited):
  """ Vrátí posloupnost stavů od počátečního k cílovému """
  path=[]
  while state is not None:
    path+=[state]          # přidávat dozadu je efektivnější
    state=visited[state]
  return list(reversed(path))
```

Nerekurzivní prohledávání do hloubky (3)

Nalezení cesty z cíle na začátek.

```
def find_path(state,visited):
  """ Vrátí posloupnost stavů od počátečního k cílovému """
  path=[]
  while state is not None:
    path+=[state]          # přidávat dozadu je efektivnější
    state=visited[state]
  return list(reversed(path))
```

Použití

```
sol=dfs2.solve(KozaVlkZeli(),maxdepth)
if sol:
  print(" -> ".join(map(str,sol)))
else:
    print("Řešení nenalezeno.")
```

Soubor `dfs2.py`.

Vlastnosti prohledávání do hloubky

- Používá zásobník
- Malá paměťová náročnost $O(D)$.
- Časová náročnost velká
  - Faktor větvení b — $O(b^D)$
  - Počet stavů N — maximálně $O(N)$
- Vhodné pro fyzické prohledávání, nebo pokud změna stavu je výpočetně náročná.
- Jdeme, dokud nenarazíme, pak se vrátíme na nejbližší křižovatku, a zkusíme to jinudy.
- Typicky omezení maximální hloubky.
- Vždy najde řešení (bez omezení hloubky).
- Nenajde vždy nejkratší řešení.
- Pokud se rozhodne špatně, náprava může trvat dlouho.

D je max. hloubka stavového prostoru.

Iterativní prohlubování
Iterative deepening

Co dělat, pokud neznáme délku optimálního řešení.

- Konzervativně odhadneme $d_{\max}$
- Prohledávání do hloubky $d_{\max}$
- Není-li řešení nalezeno, zvětšíme $d_{\max}$.

Prohledávání do šířky
Breadth-first search

- Hledáme v pořadí délky sekvence akcí.
- Nejprve prozkoumáme všechny sousedy, pak jejich sousedy, atd.
- Paměťová náročnost velká, $O(b^d)$, maximálně $O(N)$.
- Časová náročnost velká, $O(b^d)$, maximálně $O(N)$.
- Vždy najde řešení (bez omezení hloubky).
- Vždy najde nejkratší řešení.
- Zásobník nahradíme frontou.

d je délka řešení

Prohledávání do šířky

Nalezení všech stavů

```
def all_states(problem,maxdepth=100):
  visited=set()                    # již navštívené stavy
  waiting=queue.Queue()            # waiting = dvojice
  waiting.enqueue((problem,0))
  while not waiting.is_empty():
    state,level=waiting.dequeue()  # nový stav ke zpracování
    if state not in visited:
      visited|={state}   # označ stav jako navštívený
      if level<maxdepth:
        for s in state.succ():
          waiting.enqueue((s,level+1))
  return visited
```

Soubor `bfs.py`.

Prohledávání do šířky

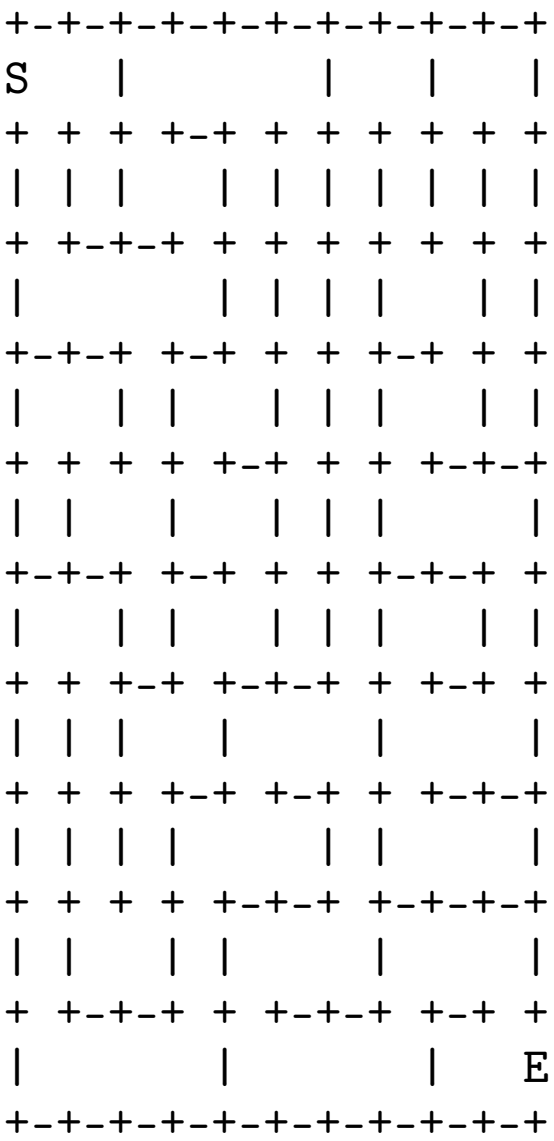
Nalezení cílového stavu

```
def solve(problem,maxdepth=100):
    visited={}                          # stavy->předchůdci
    waiting=queue.Queue()               # (stav, předchůdce,úroveň)
    waiting.enqueue((problem,None,0))
    while not waiting.is_empty():
      state,prev,level=waiting.dequeue()    # nový stav
      if state not in visited:      # je opravdu nový?
        visited[state]=prev         # zapamatujeme si přechůdce
        if state.final():           # koncový stav?
          return find_path(state,visited)
        if level<maxdepth:
          for s in state.succ():
            waiting.enqueue((s,state,level+1))
    return None                     # řešení nenalezeno
```

Soubor `bfs.py`.

Bludiště

maze



Nalezněte cestu od startu (S) k cíli (E).

Implementace bludiště

```
class Maze:
  def __init__(self,filename):
    self.m=list(map(lambda x:x.rstrip('\n'),
           open(filename,'rt').readlines() ))
    self.ny=len(self.m)
    assert(self.ny>0)
    self.nx=len(self.m[0])
    assert(all(map(lambda r: len(r)==self.nx,self.m)))
    i="".join(self.m).find('S')
    assert(i>=0)
    self.sy= i // self.nx                    # pozice S
    self.sx= i %  self.nx
    j="".join(self.m).find('E')
    assert(j>=0)
    self.ey= j // self.nx                    # pozice E
    self.ex= j %  self.nx
```

Soubor `maze.py`.

Implementace bludiště (2)

```
class MazeState:
  """ objekt reprezentující pozici v bludišti """
  def __init__(self,maze,y=None,x=None): # 'maze' typu 'Maze'
    self.maze=maze
    self.x=x if x is not None else maze.sx
    self.y=y if y is not None else maze.sy


  def final(self):
    return self.maze.m[self.y][self.x]=='E'
```

Implementace bludiště (3)

```
actions=((1,0),(0,1),(-1,0),(0,-1))  # down, right, up, left

def succ(self):
  successors=[]    # následnící současného stavu
  for dy,dx in self.actions: # možné akce
    y=self.y+dy
    x=self.x+dx
    if (x>=0 and x<self.maze.nx and y>=0 and y<self.maze.ny
          and self.maze.m[y][x] in ' SE'):
      successors+=[MazeState(self.maze,y,x)]
  return successors
```

Implementace bludiště (4)

```
def __eq__(self,a):
  return self.x==a.x and self.y==a.y and self.maze==a.maze

def __hash__(self):
  return hash((self.x,self.y,self.maze))

def __str__(self):
  """ vrátí reprezentaci stavu jako řetězec """
  return "("+str(self.y)+","+str(self.x)+")"
```

Hledání cesty v bludišti — prohledávání do šířky

```
import bfs

def solve_maze_bfs(filename='maze.txt',maxdepth=500):
  m=Maze(filename)
  sol=bfs.solve(MazeState(m),maxdepth=maxdepth)
  if sol:
    print("Délka cesty=",len(sol))
    m.print(path=sol)
  else:
    print("Řešení nenalezeno.")
```

Hledání cesty v bludišti — prohledávání do šířky

```
maze.solve_maze_bfs()
```

```
Navštíveno  329  stavů.
Délka cesty= 107
+-+-+-+-+-+-+-+-+-+-+-+
S#  |#####  |   |   |
+#+ +#+-+#+ + + + + +
|#| |###|#| | | | | |
+#+-+-+#+#+ + + + + +
|#######|#| | |   | |
+-+-+ +-+#+ + +-+ + +
|   | |###| | |   | |
+ + + +#+-+ + + +-+-+
| |   |###| | |     |
+-+-+ +-+#+ + +-+-+ +
|###| |###| | |   | |
+#+#+-+#+-+-+ + +-+ +
|#|#|###|#####|     |
+#+#+#+-+#+-+#+ +-+-+
|#|#|#|###  |#|     |
+#+#+#+#+-+-+#+-+-+-+
|#|###|#|#####|#####|
+#+-+-+#+#+-+-+#+-+#+
|#######|#######|  #E
+-+-+-+-+-+-+-+-+-+-+
```

Soubor `maze.py`.

Hledání cesty v bludišti — prohledávání do hloubky

```
maze.solve_maze_dfs2()
```

```
Navštíveno  334  stavů.
Délka cesty= 107
+-+-+-+-+-+-+-+-+-+-+-+
S#  |#####  |   |   |
+#+ +#+-+#+ + + + + +
|#| |###|#| | | | | |
+#+-+-+#+#+ + + + + +
|#######|#| | |   | |
+-+-+ +-+#+ + +-+ + +
|   | |###| | |   | |
+ + + +#+-+ + + +-+-+
| |   |###| | |     |
+-+-+ +-+#+ + +-+-+ +
|###| |###| | |   | |
+#+#+-+#+-+-+ + +-+ +
|#|#|###|#####|     |
+#+#+#+-+#+-+#+ +-+-+
|#|#|#|###  |#|     |
+#+#+#+#+-+-+#+-+-+-+
|#|###|#|#####|#####|
+#+-+-+#+#+-+-+#+-+#+
|#######|#######|  #E
+-+-+-+-+-+-+-+-+-+-+
```

Soubor `maze.py`.

Další bludiště

Do hloubky

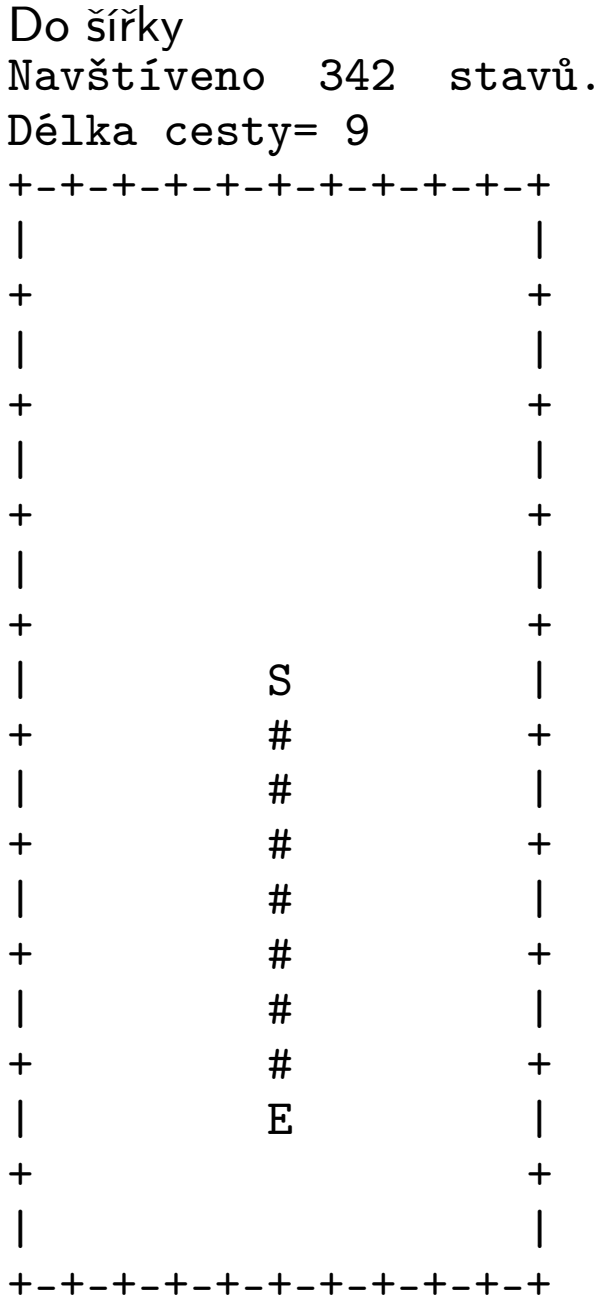
```
Navštíveno  733  stavů.
Délka cesty= 363
+-+-+-+-+-+-+-+-+-+-+-+
S####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################|
+####################+
|####################E
+-+-+-+-+-+-+-+-+-+-+-+
```

Do šířky

```
Navštíveno  1370  stavů.
Délka cesty= 39
+-+-+-+-+-+-+-+-+-+-+-+
S#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|#                    |
+#                    +
|####################E
+-+-+-+-+-+-+-+-+-+-+-+
```

Informované prohledávání
Informed search

- Víme kam jdeme
- Začneme nejslibnějšími akcemi
- Značné urychlení
- Může/nemusí zaručit nalezení řešení



Prioritní prohledávání (hladové)

Best-first search/priority search/greedy search

- Akce mají ceny *(prozatím jednotkové)*
- Pro každý stav s odhadneme cenu $h(s)$ dosažení minima
  - Pro 2D bludiště $h = |x - x_E| + |y - y_E|$
- Akce čekající na zpracování uložíme do **prioritní fronty** a budeme je brát v pořadí dle h.
- Nalezení řešení je zaručeno.
- Nalezení minima není zaručeno.
- Časová a prostorová složitost jako u prohledávání do šířky.

Implementace prioritního vyhledávání

Nová metoda pro ohodnocení stavů (zde v `MazeState`)

```
  def cost(self):
    """ Evaluate the cost of the current state """
    return abs(self.x-self.maze.ex)+abs(self.y-self.maze.ey)
```

Pomocná třída pro vkládání do prioritní fronty

```
class HeapQItem:
  def __init__(self,cost,payload):
    self.cost=cost
    self.payload=payload

  def get(self):
    return self.payload

  def __lt__(self,other):
    """ tato metoda zařídí porovnatelnost """
    return self.cost<other.cost
```

Soubor `maze.py`, `prioritysearch.py`.

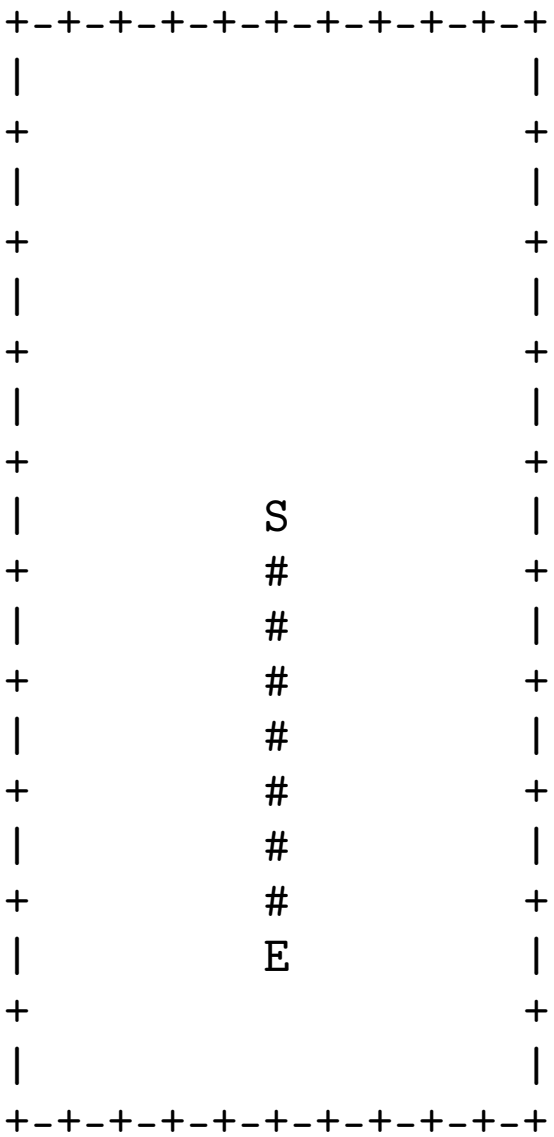
Implementace prioritního vyhledávání (2)

```
def solve(problem,maxdepth=100):
  visited={}                  # stav-> předchůdce
  waiting=[]                  # (cena,(stav,předchůdce,úroveň))
  heapq.heappush(waiting,
         HeapQItem(problem.cost(),(problem,None,0)))
  numvisited=0
  while len(waiting)>0:
    numvisited+=1
    state,prev,level=heapq.heappop(waiting).get()
    if state not in visited:     # je opravdu nový?
      visited[state]=prev        # zapamatujeme si přechůdce
      if state.final():          # koncový stav?
        print("Visited ",numvisited," states.")
        return find_path(state,visited)
      if level<maxdepth:
        for s in state.succ():
          heapq.heappush(waiting,
                 HeapQItem(s.cost(),(s,state,level+1)))
  return None                    # řešení nenalezeno
```

Příklad informovaného prohledávání

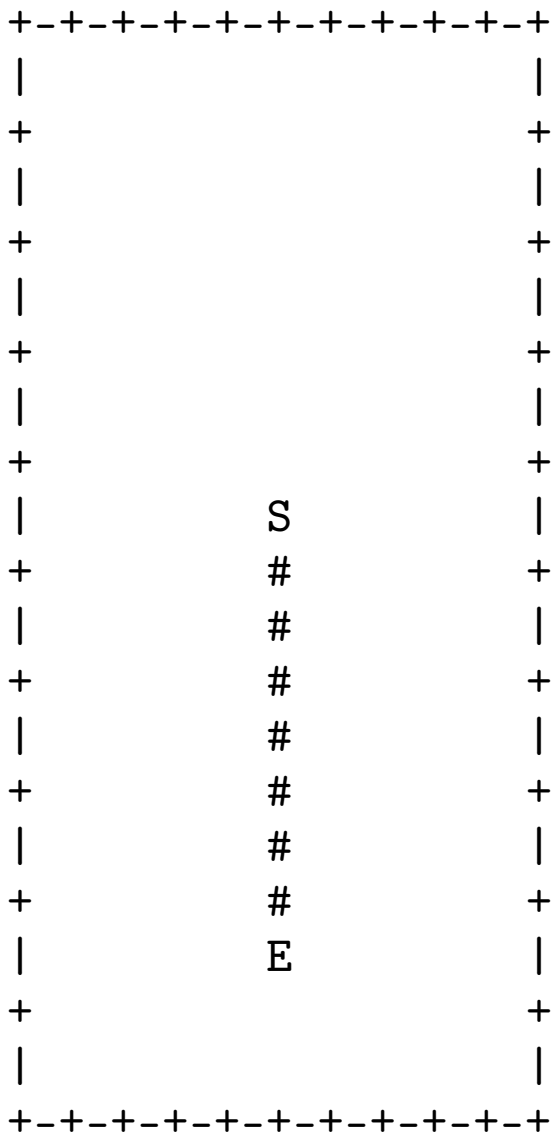
Do šířky

```
Navštíveno  342  stavů.
Délka cesty= 9
```



Prioritní hledání

```
Navštíveno  9  stavů.
Délka cesty= 9
```



Algoritmus A^*

- Prioritní hledání ignoruje cenu aktuální cesty
  - $g(s)$ — cena cesty z kořene do uzlu s
- A^* uvažuje akce v pořadí dle odhadu celkové ceny $f(s) = g(s) + h(s)$
  - Neprodlužuje příliš dlouhé cesty.
- Přípustná/optimistická heuristika ($h(s) \leq d(s, T)$)
  - Algoritmus A* nalezne řešení.
- Nalezení optimální řešení není zaručeno.
  - Můžeme najít lepší cestu do již navštíveného uzlu.
  - Je možné přidat aktualizaci.
  - Je-li heuristika *monotónní* ($h(x) \leq h(y) + d(x, y)$), pak je navštívíme každý uzel nejvýše jednou, řešení je optimální.
- Časová a prostorová složitost jako u prohledávání do šířky.

Algoritmus A^* — implementace

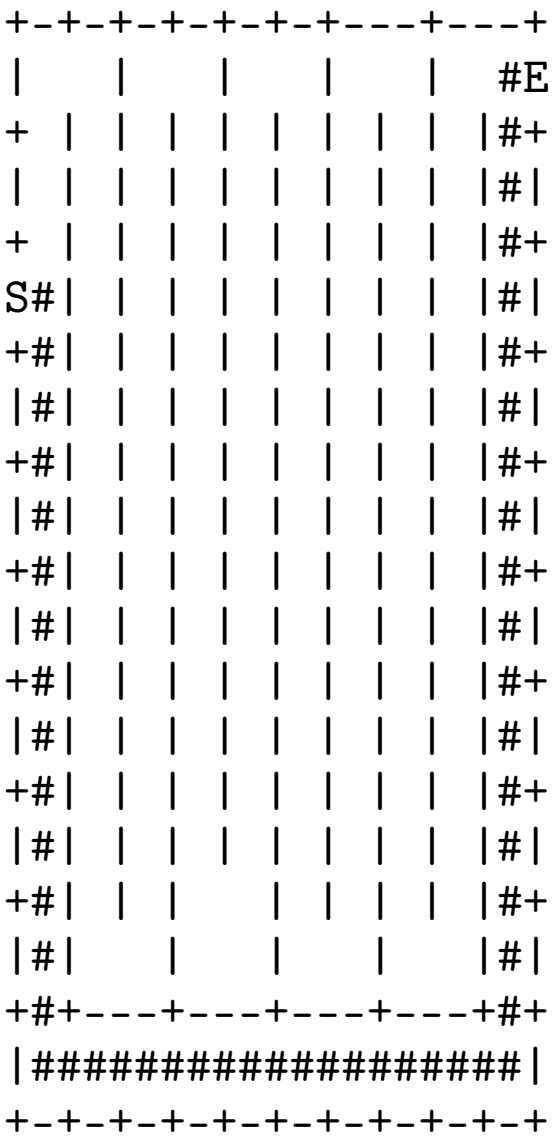
```
def solve(problem,maxdepth=100):
  visited={}                  # stav->předchůdce
  waiting=[]                  # (cena,(stav,předchůdce,úroveň))
  heapq.heappush(waiting,
          HeapQItem(problem.cost(),(problem,None,0)))
  numvisited=0
  while len(waiting)>0:
    numvisited+=1
    state,prev,level=heapq.heappop(waiting).get()
    if state not in visited:      # je opravdu nový?
      visited[state]=prev         # přechůdce
      if state.final():           # koncový stav?
        return find_path(state,visited)
      if level<maxdepth:
        for s in state.succ():
          heapq.heappush(waiting,
              HeapQItem(s.cost()+level,(s,state,level+1)))
  return None                     # řešení nenalezeno
```

Soubor `astarsearch.py`.

Těžké bludiště

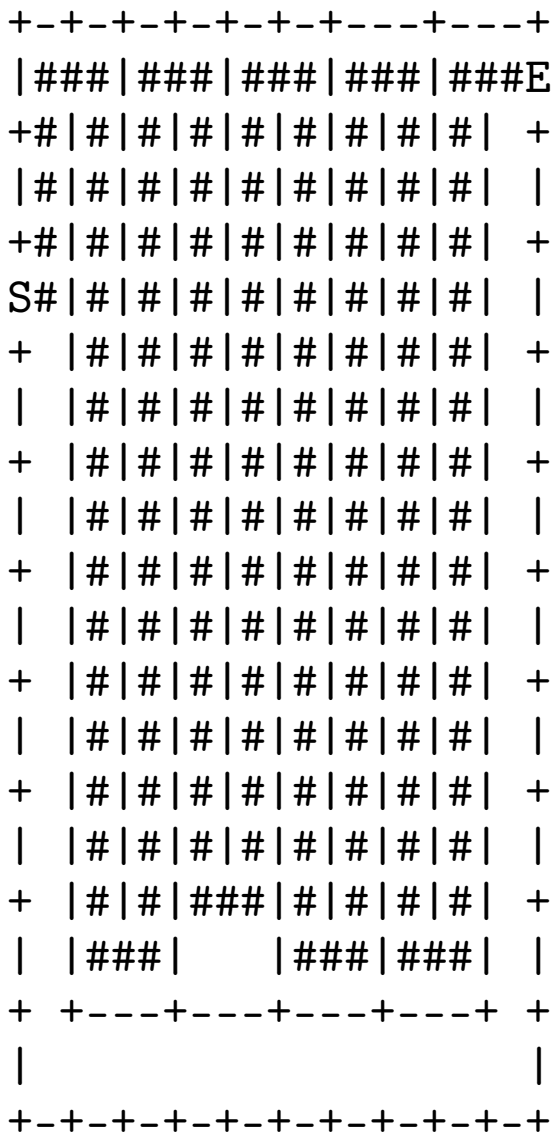
Do šířky

```
Navštíveno  203  stavů.
Délka cesty= 53
```



Prioritní hledání

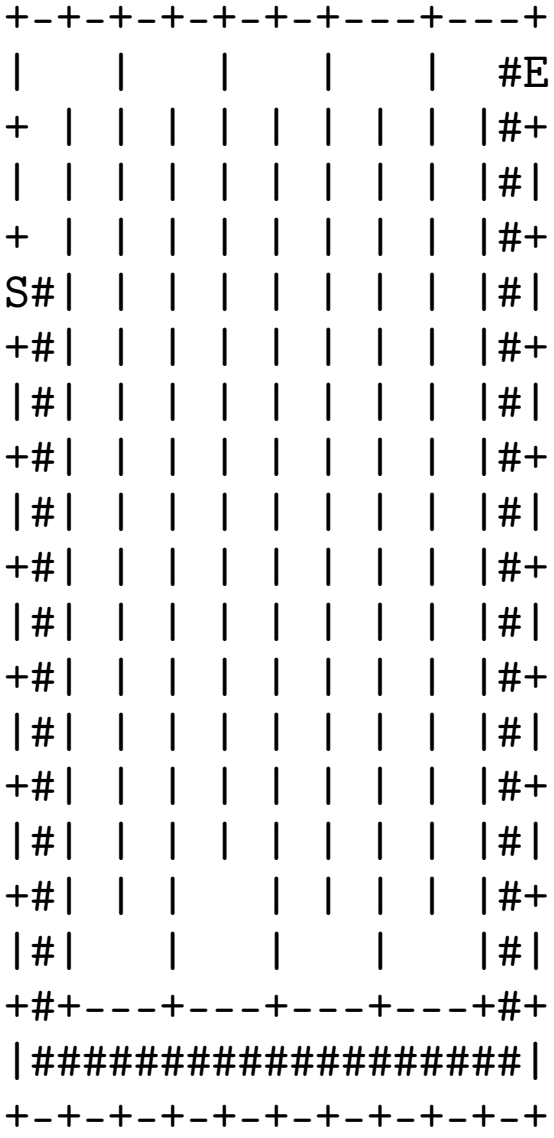
```
Navštíveno  284  stavů.
Délka cesty= 151
```



Těžké bludiště (2)

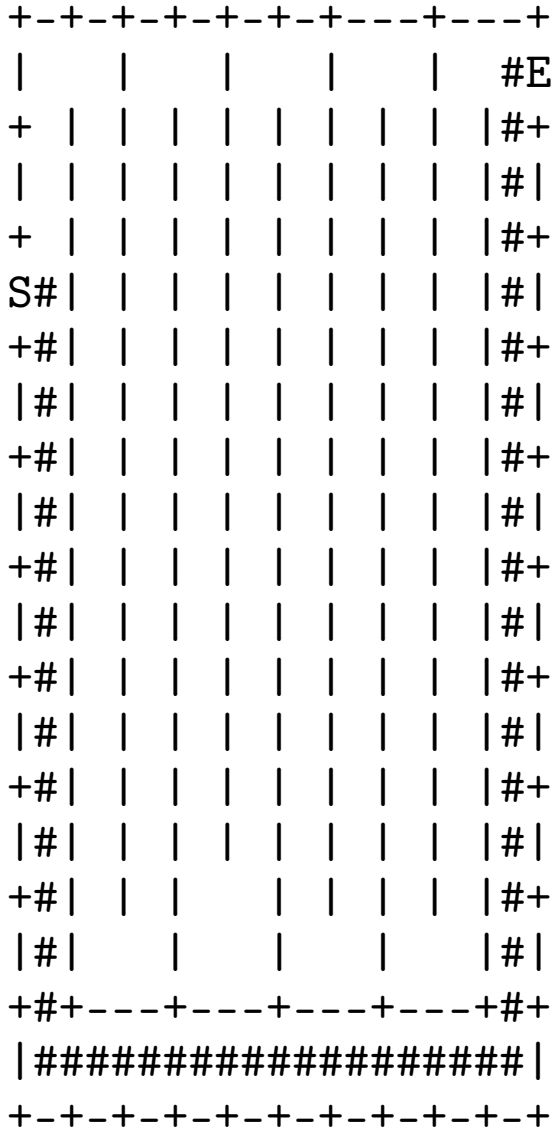
Do šířky

```
Navštíveno  203  stavů.
Délka cesty= 53
```



A^* algoritmus

```
Navštíveno  108  stavů.
Délka cesty= 53
```



Náměty na domácí práci

- Řešte úlohu misionáři a kanibalové.
- Řešte úlohu přelévání vody.
- Při prohledávání kontrolujte co nejdříve, jestli jsme nenašli cílový stav.
- Při nalezení řešení uchovávejte i posloupnost akcí.
- Při hledání v bludišti si zapisujte navštívená místa rovnou do bludiště. Vizualizujte navštívené stavy, udělejte animaci hledání.
- Místo celého stavu si pamatujte jen změnu od minulého stavu.
- Doplňte možnost akcí různých cen v algoritmu A^* i prioritního hledání. Vyzkoušejte např. bludišti s možností pohybu po úhlopříčce.