KSP 33-Z1-3: Petrův zmatený výlet

Dopředné prohledávání s memoizací

Seminář z algoritmizace

13. února 2026

Didaktické cíle nového přístupu

Tato metoda simuluje přirozený pohyb po mapě, ale na rozdíl od naivního přístupu řeší problém opakovaného procházení stejných cest. Žáci se zde naučí:

- **Memoizaci:** Ukládání výsledků již prozkoumaných cest do slovníku, aby se nemusely počítat znovu.
- **Barvení grafu (označování komponent):** Použití unikátních ID (1, 2, 3...) pro rozlišení jednotlivých pokusů.
- **Detekci cyklů:** Rozpoznání situace, kdy narazíme na políčko, které patří do *aktuální* prohledávané cesty (nekonečná smyčka).

Rozbor algoritmu

Princip fungování:

1. Postupně bereme každé nenavštívené políčko a začneme z něj cestu.
2. Každé nové cestě přidělíme unikátní číslo (`id_cesty`).
3. Políčka na cestě si označíme tímto číslem.
4. **Cesta končí, pokud:**
   - Dojdeme do `C` ⇒ mapujeme `id_cesty` na `#`.
   - Vypadneme z mapy ⇒ mapujeme `id_cesty` na `..`
   - Narazíme na **cizí číslo** ⇒ převezmeme výsledek ze slovníku a ušetříme čas.
   - Narazíme na **vlastní číslo** ⇒ našli jsme cyklus ⇒ mapujeme na `..`

Pseudokód dopředného prohledávání

```
navstiveno = pole nul o velikosti R x S
slovnik_vysledku = prazdny slovnik
id_cesty = 1

pro kazde pole (r, c):
    pokud navstiveno[r][c] == 0 a neni to 'C':
        aktualni_r, aktualni_c = r, c

        dokud pravda:
            pokud jsme mimo mapu:
                slovnik_vysledku[id_cesty] = '.'
                prerus cyklus
            pokud jsme v cili 'C':
                slovnik_vysledku[id_cesty] = '#'
                prerus cyklus

            stav = navstiveno[aktualni_r][aktualni_c]
            pokud stav == id_cesty: // Nasli jsme cyklus vlastni cesty
                slovnik_vysledku[id_cesty] = '.'
                prerus cyklus
            pokud stav != 0: // Narazili jsme na jiz prozkoumanou cestu
                slovnik_vysledku[id_cesty] = slovnik_vysledku[stav]
                prerus cyklus

            navstiveno[aktualni_r][aktualni_c] = id_cesty
            posun se ve smeru sipky

        id_cesty += 1
```

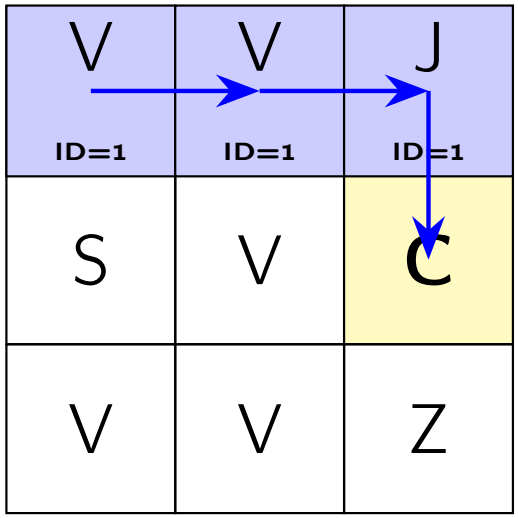
Modelový příklad: Značení cest a napojení

V	V	J
S	V	**C**
V	V	Z

Krok 1: Výchozí stav
Máme matici 3x3 s cílem `C`.
Hledáme od levého horního rohu.

Pojďme trasovat cesty...

Modelový příklad: Značení cest a napojení

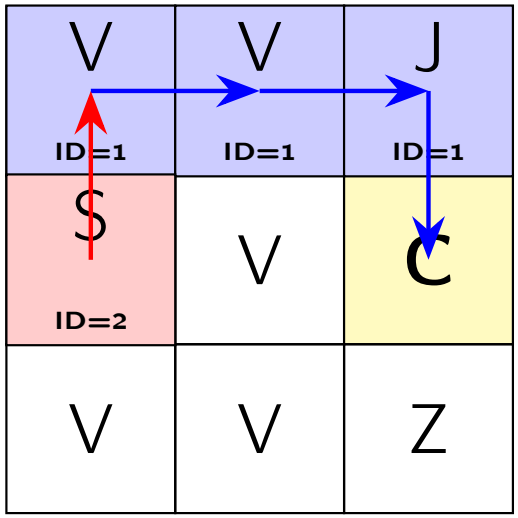


Krok 2: Cesta ID=1

Začínáme v `(0,0)`.
Jdeme po šipkách: `V` → `V` → `J` → `C`.
Políčkům po cestě přiřadíme `ID=1`.
Našli jsme cíl `C`. Cesta končí.

Slovník: `{1: '#'}`

Modelový příklad: Značení cest a napojení

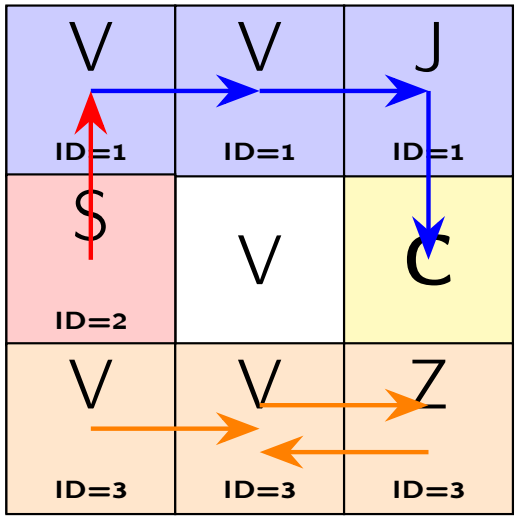


Krok 3: Cesta ID=2 (Memoizace)

Další volné políčko je `(1,0)`.
Šipka S (nahoru) nás vede na `(0,0)`.
Ale ouha, `(0,0)` už má `ID=1`!
Cesta 2 hned končí.
Převezme se výsledek cesty 1.

Slovník: `{1: '#', 2: '#'}`

Modelový příklad: Značení cest a napojení



Krok 4: Cesta ID=3 (Detekce cyklu)

Začínáme v `(2,0)`.

Jdeme: `V` → `V` → `Z` → a zpět `na (2,1)`.

Narazili jsme na vlastní `ID=3`. Cyklus!

Cesta se zacyklila, do `C` nedojde.

Slovník: `{1: '#', 2: '#', 3: '.'}`

Kód v Pythonu (Část 1: Inicializace)

```python
def vyres_dopredu():
    R, S = map(int, input().split())
    mapa = [input().strip() for _ in range(R)]

    # Matice pro ukladani cisla cesty (0 znamena nenavstiveno)
    navstiveno = [[0] * S for _ in range(R)]

    # Slovnik provazujici cislo cesty se znakem '#' nebo '.'
    vysledky_cest = {}
    id_cesty = 1

    # Pomocny slovnik pro snadny posun
    posuny = {
        'S': (-1, 0),
        'J': (1, 0),
        'V': (0, 1),
        'Z': (0, -1)
    }
```

Kód v Pythonu (Část 2: Hlavní logika)

```
    for r in range(R):
        for c in range(S):
            # Preskocime cil a uz prozkoumana policka
            if navstiveno[r][c] != 0 or mapa[r][c] == 'C':
                continue

            curr_r, curr_c = r, c
            while True:
                # 1. Kontrola opusteni mapy
                if curr_r < 0 or curr_r >= R or curr_c < 0 or curr_c >= S:
                    vysledky_cest[id_cesty] = '.'
                    break
                # 2. Kontrola nalezeni cile
                if mapa[curr_r][curr_c] == 'C':
                    vysledky_cest[id_cesty] = '#'
                    break

                stav = navstiveno[curr_r][curr_c]
                # 3. Kontrola cyklu ve vlastni ceste
                if stav == id_cesty:
                    vysledky_cest[id_cesty] = '.'
                    break
                # 4. Napojeni na cizi, jiz vyhodnocenou cestu
                elif stav != 0:
                    vysledky_cest[id_cesty] = vysledky_cest[stav]
                    break
```

Kód v Pythonu (Část 2: Hlavní logika)

Pokračování kódu

```
                # Oznaceni policka a posun dal
                navstiveno[curr_r][curr_c] = id_cesty
                dr, dc = posuny[mapa[curr_r][curr_c]]
                curr_r += dr
                curr_c += dc

            id_cesty += 1
```

Kód v Pythonu (Část 3: Tvorba výstupu)

```
    # Vygenerovani a vypis vysledne mapy
    for r in range(R):
        vysledny_radek = []
        for c in range(S):
            if mapa[r][c] == 'C':
                vysledny_radek.append('C')
            else:
                # Podivame se, jake ID ma policko,
                # a podle toho ze slovniku vytahneme znak
                id_policka = navstiveno[r][c]
                znak = vysledky_cest.get(id_policka, '.') # Fallback
                vysledny_radek.append(znak)
        print("".join(vysledny_radek))

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

Metodické tipy pro učitele

- **Význam pomocného slovníku (`posuny`):** Ukažte žákům, jak se vyhnout dlouhým blokům `if-elif` pomocí mapování směrů na vektory (dr, dc). Značně to zpřehlední kód.

- **Rozdíl mezi `stav == id_cesty` a `stav != 0`:** Toto je kritický moment pro pochopení algoritmu. První případ znamená cyklus aktuálního průchodu (šli jsme do kruhu). Druhý případ znamená úsporu času (napojili jsme se na již vyřešený kus mapy).

- **Paměťová vs. Časová složitost:** Oproti zpětnému BFS potřebujeme pole stejné velikosti pro evidenci čísel `navstiveno` a navíc paměť pro slovník `vysledky_cest`. Časová složitost však zůstává lineární a efektivní, tj. $\mathcal{O}(R \times S)$.