

Řešení problémů prohledáváním Hledání optimální sekvence stavů, rozhodnutí, akcí

Tomáš Svoboda, Petr Pošík

Vision for Robots and Autonomous Systems, Center for Machine Perception
Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague

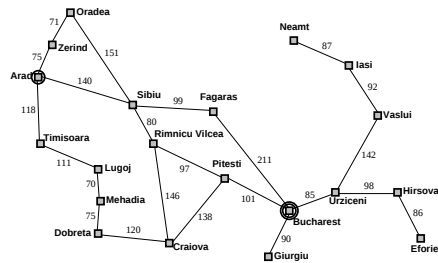
26. února 2025

1 / 33

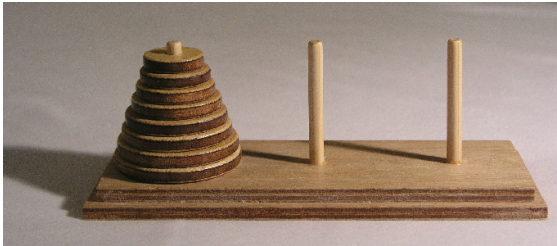
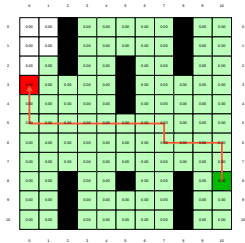
Notes

Ukážeme, že rozhodnutí/akce/řídící příkazy jsou pro deterministické problémy totéž.

Jaké problémy chceme řešit?



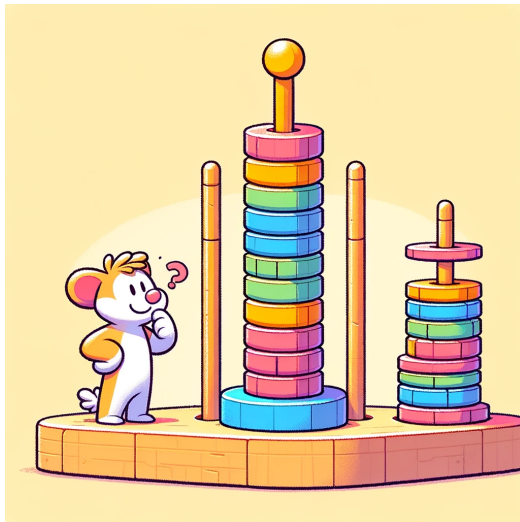
12	1	2	15
11	6	5	8
7	10	9	4
	13	14	3



1

¹CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=228623>

Klíčové je rozumět problému. DALL-E:

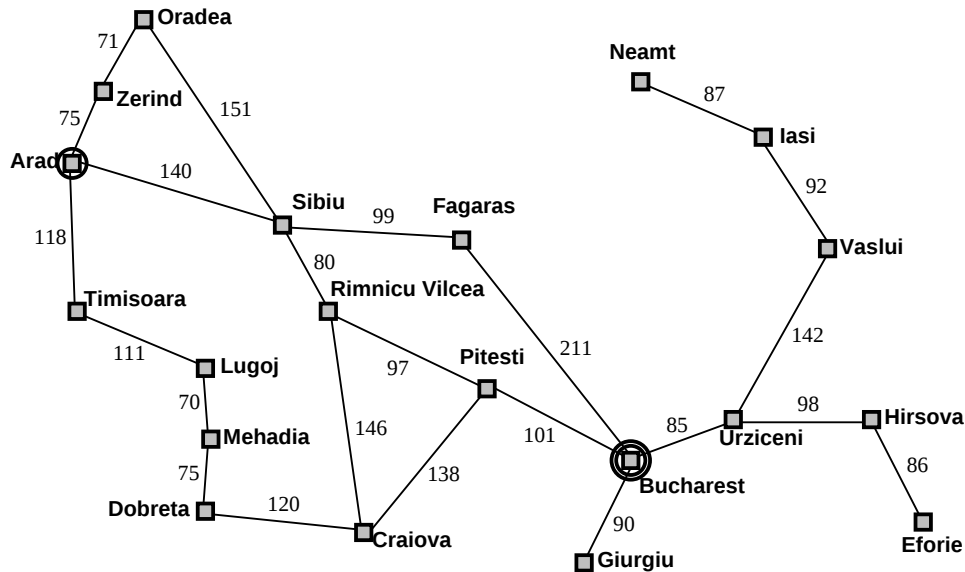


Notes

DALL-E vytvořil tyto obrázky, i když měl správné popis problému.

- ▶ Problém prohledávání. *Co chceme řešit?*
- ▶ Graf stavového prostoru. *Jak problém formalizujeme/reprezentujeme? Abstrakce problému.*
- ▶ Prohledávací stromy. *Vizualizace běhu algoritmu.*
- ▶ Strategie: které větve stromu vybírat?
- ▶ Vlastnosti strategie/algoritmu. *Paměť, čas, ...*
- ▶ Jak to naprogramovat?

Příklad: Cestujeme po Rumunsku



5 / 33

Notes

Začneme jednoduše: každý asi zná navigaci např. do auta - problém plánování cesty. Waze, Garmin, ... Zde lze problém poměrně přímo transformovat na graf: mapa je svým způsobem graf, kde stavy jsou města (křižovatky) a hrany jsou silnice a cesty mezi nimi.

Dokážete si představit i další problémy?

Například:

- Rozvoz zboží. Speciální případ: Problém obchodního cestujícího – každé město musí být navštíveno právě jednou.
- Plánování pohybu robota – mobilního robota nebo manipulátoru.
- Rozmístění funkčních bloků na VLSI čipech.
- ...

Příklad: Stavby a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)

akce (rozhodnutí): volba cesty z města

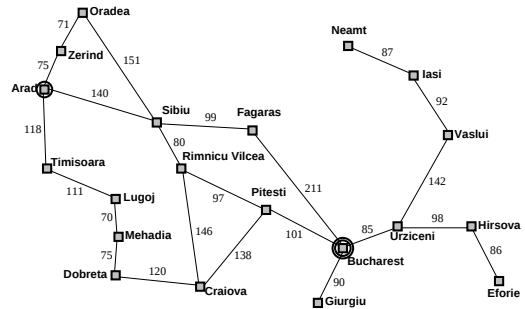
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ...:

Energie, čas, poplatky, ...



6 / 33

Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Stavby a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)
akce (rozhodnutí): volba cesty z města

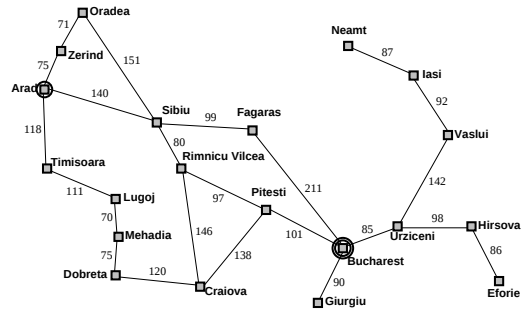
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ...:

Energie, čas, poplatky, ...



Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Stav a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)

akce (rozhodnutí): volba cesty z města

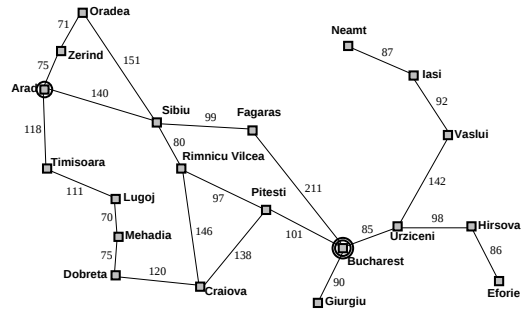
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ...:

Energie, čas, poplatky, ...



Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Stav a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)

akce (rozhodnutí): volba cesty z města

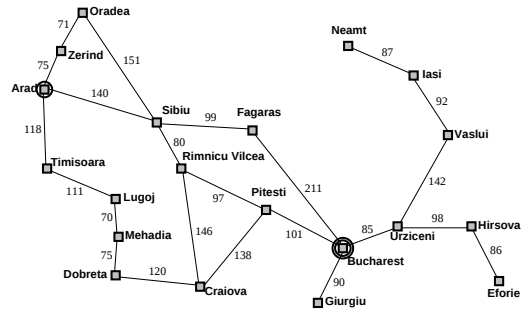
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ...:

Energie, čas, poplatky, ...



6 / 33

Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Stav a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)

akce (rozhodnutí): volba cesty z města

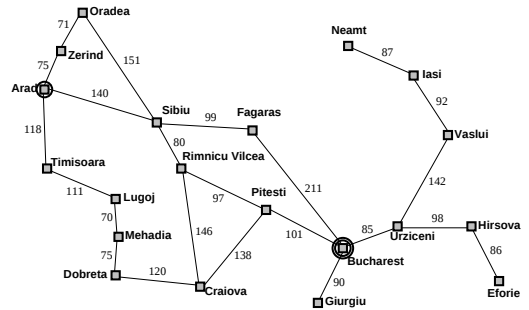
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ... :

Energie, čas, poplatky, ...



6 / 33

Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Stav a akce

Cíl:

být v Bukurešti

Formulace problému:

stav: město (v jakém městě se nacházíte)

akce (rozhodnutí): volba cesty z města

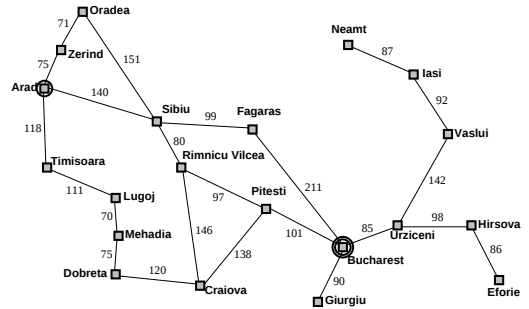
Řešení:

Posloupnost měst (cesta)

(posloupnost akcí/rozhodnutí [2])

Optimalita – náklady, ztráty, užitek, profit, ...:

Energie, čas, poplatky, ...



Notes

Klasický problém z knihy [2], používáme jej také.

Stavy a **akce** budou diskutovány v mnoha přednáškách a algoritmech. Je důležité jim dobře rozumět. Ve stavech (městech, křižovatkách) potřebujeme vybrat odbočku - výběr odbočky je akce. Předpokládáme, že touto akcí se dostaneme do dalšího stavu (města, křižovatky), což je výsledek (**result**) akce zvolené v nějakém stavu.

Příklad: Skládačka

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

stavy?

akce?

řešení?

náklady/ztráty?

7 / 33

Notes

- Stav: Pozice každé dlaždice a prázdného místa.
- Počet stavů: $9!$
- Úvodní stav: kterýkoli stav. (Poznamenejme, že kterýkoli cílový stav může být dosažen přesně z poloviny možných úvodních stavů.)
- Akce: Možné pohyby prázdného místa - Doleva, Doprava, Nahoru, Dolů (nebo jejich podmnožina)
- Test řešení/cíle: Kontrola, že pozice dlaždic v daném stavu odpovídají požadované konfiguraci.
- Cena řešení: počet kroků v cestě (každý krok má cenu 1)

Ukázkový problém (3.2.1) z [2].

Úloha prohledávání (Search Problem)

- ▶ **Stavový prostor** (včetně počátečního stavu): pozice, konfigurace hrací plochy, ...
 - ▶ Prostor/množina akcí : jed' do, Nahoru, Dolů, Doleva, ...
 - ▶ Přejchodový model : Pro daný stav a akci, vrať následný stav (a cenu nebo odměnu za přechod)
 - ▶ Test dosažení cíle : Máme hotovo?

8 / 33

Notes

Tuto terminologii budeme používat příštích 5-6 přednášek; objeví se i v Markovských rozhodovacích procesech a posilovaném učení.

Udělejte si mentální test: Jste robot, který se potřebuje dostat z vašeho domova do školy. Co by v této úloze byly stavy, akce, přechodový model, test dosažení cíle?

Úloha prohledávání (Search Problem)

- ▶ **Stavový prostor** (včetně počátečního stavu): pozice, konfigurace hrací plochy, ...
- ▶ **Prostor/množina akcí** : jed' do, Nahoru, Dolů, Doleva, ...
 - ▶ **Přechodový model** : Pro daný stav a akci, vrať následný stav (a cenu nebo odměnu za přechod)
 - ▶ **Test dosažení cíle** : Máme hotovo?

Notes

Tuto terminologii budeme používat příštích 5-6 přednášek; objeví se i v Markovských rozhodovacích procesech a posilovaném učení.

Udělejte si mentální test: Jste robot, který se potřebuje dostat z vašeho domova do školy. Co by v této úloze byly stavy, akce, přechodový model, test dosažení cíle?

Úloha prohledávání (Search Problem)

- ▶ **Stavový prostor** (včetně počátečního stavu): pozice, konfigurace hrací plochy, ...
- ▶ **Prostor/množina akcí** : jed' do, Nahoru, Dolů, Doleva, ...
- ▶ **Přechodový model** : Pro daný stav a akci, vrať následný stav (a **cenu** nebo **odměnu** za přechod)
- ▶ **Test dosažení cíle** : Máme hotovo?

Notes

Tuto terminologii budeme používat příštích 5-6 přednášek; objeví se i v Markovských rozhodovacích procesech a posilovaném učení.

Udělejte si mentální test: Jste robot, který se potřebuje dostat z vašeho domova do školy. Co by v této úloze byly stavy, akce, přechodový model, test dosažení cíle?

Úloha prohledávání (Search Problem)

- ▶ **Stavový prostor** (včetně počátečního stavu): pozice, konfigurace hrací plochy, ...
- ▶ **Prostor/množina akcí** : jed' do, Nahoru, Dolů, Doleva, ...
- ▶ **Přechodový model** : Pro daný stav a akci, vrať následný stav (a **cenu** nebo **odměnu** za přechod)
- ▶ **Test dosažení cíle** : Máme hotovo?

Notes

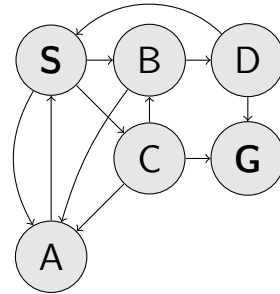
Tuto terminologii budeme používat příštích 5-6 přednášek; objeví se i v Markovských rozhodovacích procesech a posilovaném učení.

Udělejte si mentální test: Jste robot, který se potřebuje dostat z vašeho domova do školy. Co by v této úloze byly stavy, akce, přechodový model, test dosažení cíle?

Diskrétní stavový prostor

Graf stavového prostoru: reprezentace úlohy prohledávání

- ▶ Stavy $s \in \mathcal{S} = \{S, A, B, C, D, G\}$ (konečná množina)
- ▶ Hrany reprezentují akce a : pro každý stav s , $a \in \mathcal{A}(s)$ ($\mathcal{A}$ je také konečná množina)
- ▶ Přejchodová funkce $s' = \text{result}(s, a)$
- ▶ Počáteční (startovní) stav $s_0 \in \mathcal{S}$, $s_0 = S$.
- ▶ Množina cílových stavů $\mathcal{S}_G \subset \mathcal{S}$.



Každý stav se ve stavovém prostoru vyskytuje pouze *jednou*.

Notes

Formalizace skutečného problému – "vytvoření" grafu stavového prostoru – může být problematické samo o sobě. Počet stavů v praxi nemusí být konečný.

Spojitosť s grafovými algoritmy jako jsou Dijkstra, Floyd-Warshall.

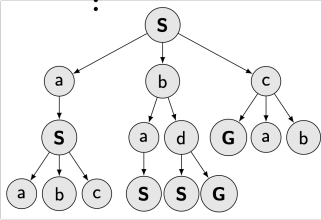
- Grafové algoritmy předpokládají úplnou informaci o grafu; je to jejich hlavní vstup.
- Pro mnoho reálných problémů, graf nemusí být znám předem.
- Graf stavového prostoru je *postupně odhalován během prohledávání*. Graf slouží spíš jako abstrakce, mentální model, než jako skutečná reprezentace dat.
- Mnoho reálných problémů má sice konečné, ale obrovské množství stavů. Pomyslete na $n - 1$ skládačku nebo na šachy: množství možných konfigurací je enormní.
- Řešení může ale být poměrně krátké (ležet v malé hloubce).

Dialog mezi agentem a prostředím (pohled programátora)

(search) agent

Q.insert(s_0)
 $s \leftarrow \text{Q.pop_first}()$

⋮



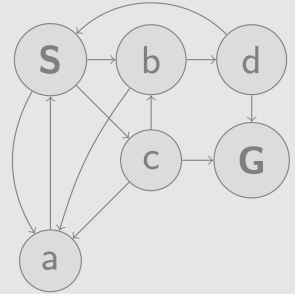
problem - env

$s_0, \mathcal{S}_G = \text{env.reset}()$

$\mathcal{A} = \text{env.get_actions}(s_0)$

$s' = \text{env.apply_transition}(s, a)$

⋮



Notes

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```

S

S

Q: (_, S)
visited: S

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

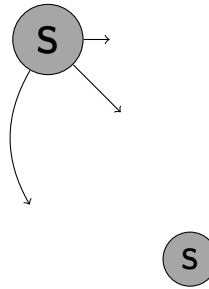
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q:
visited: S

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

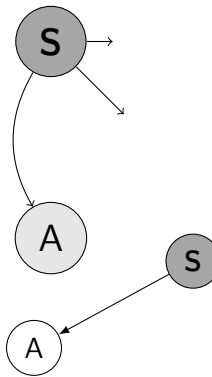
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q:
visited: S

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

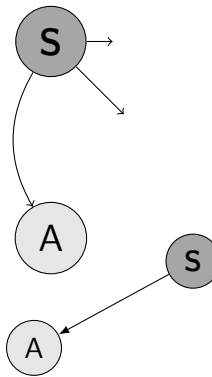
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A)
visited: S A

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

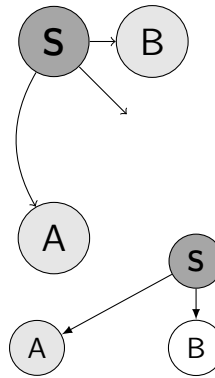
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A)
visited: S A

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

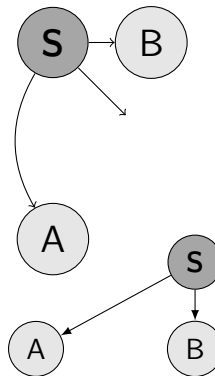
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B)
visited: S A B

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

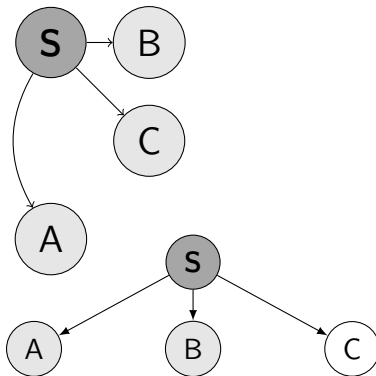
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B)
visited: S A B

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

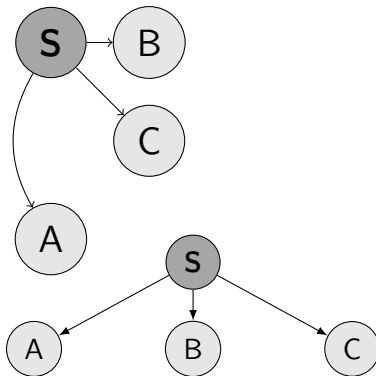
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B) (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

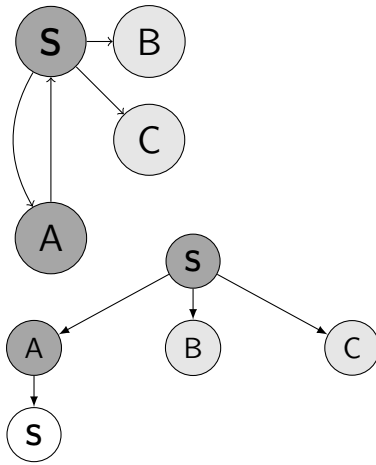
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,B) (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

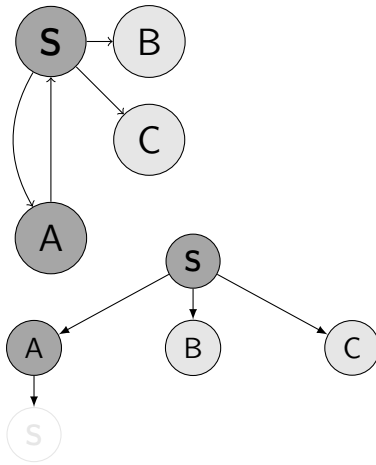
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,B) (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

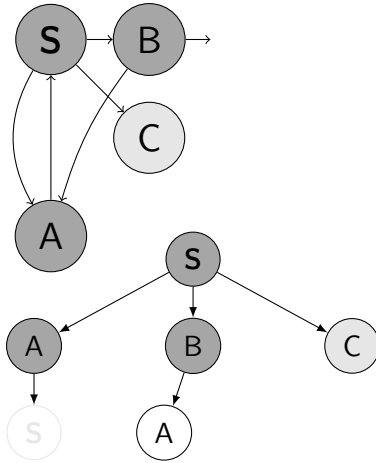
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

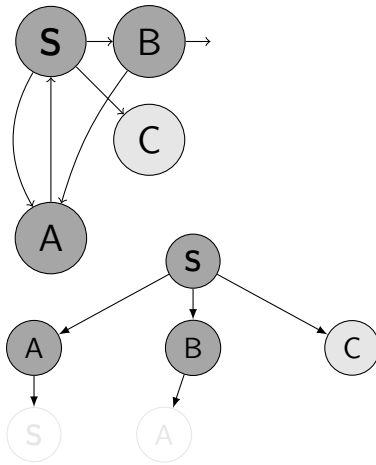
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

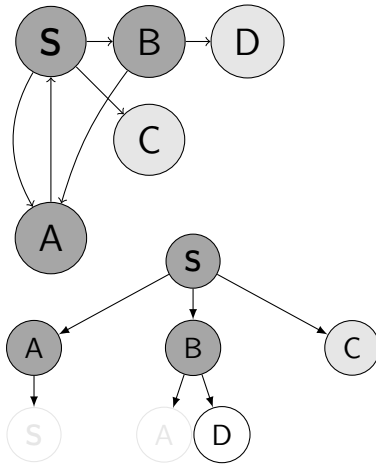
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,C)
visited: S A B C

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

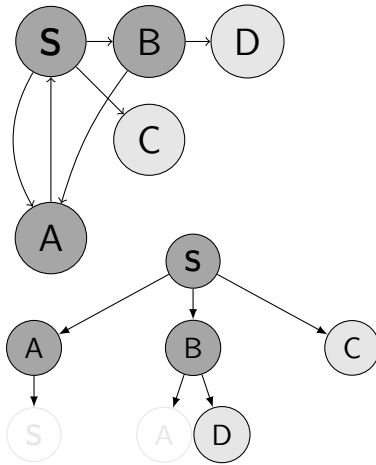
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,C) (B,D)
visited: S A B C D

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

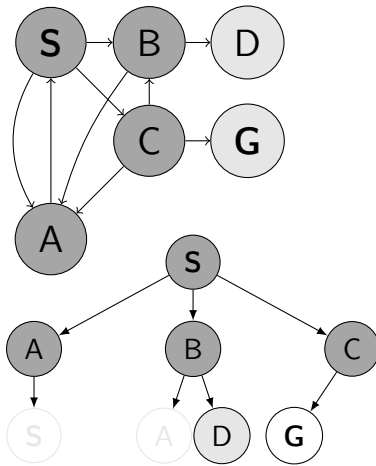
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D)
visited: S A B C D

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

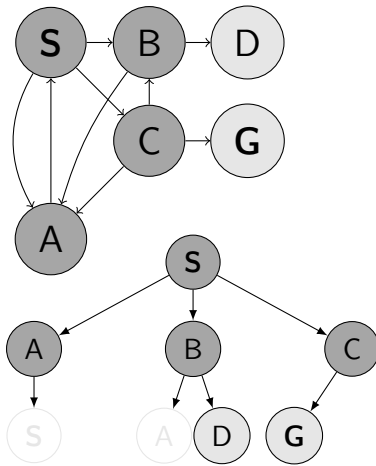
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D) (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

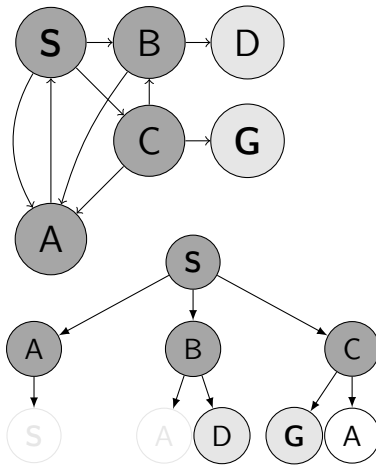
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D) (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

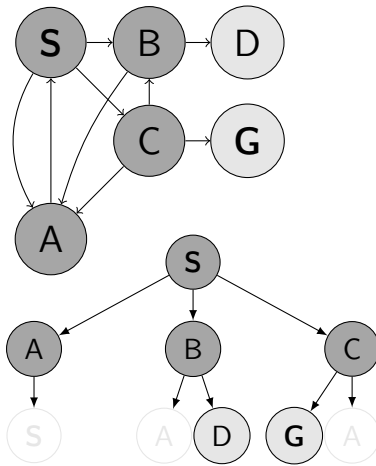
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D) (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

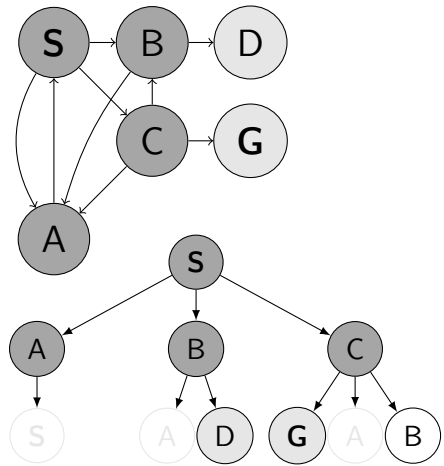
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D) (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

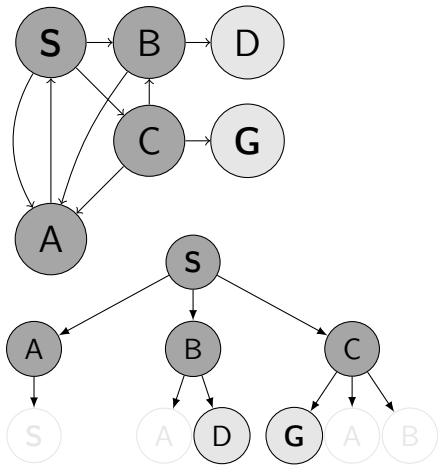
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (B,D) (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

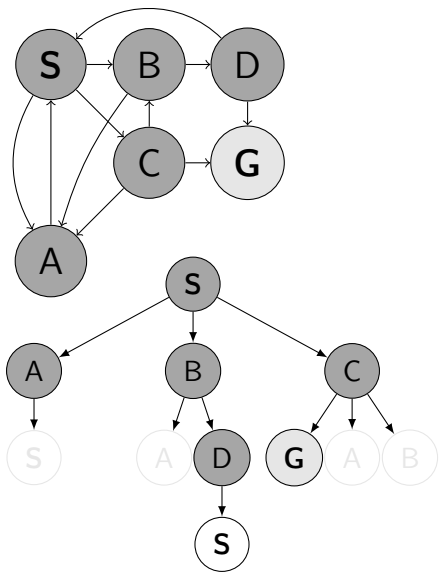
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

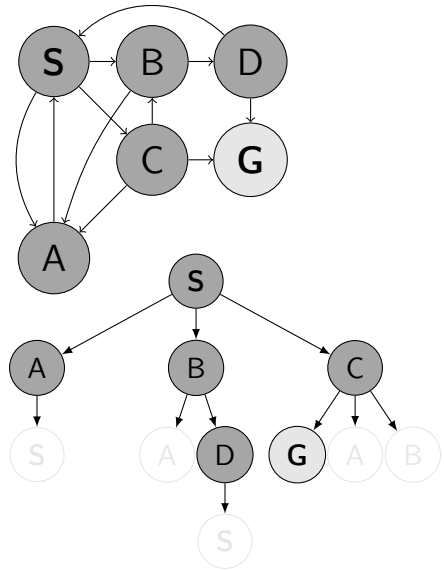
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

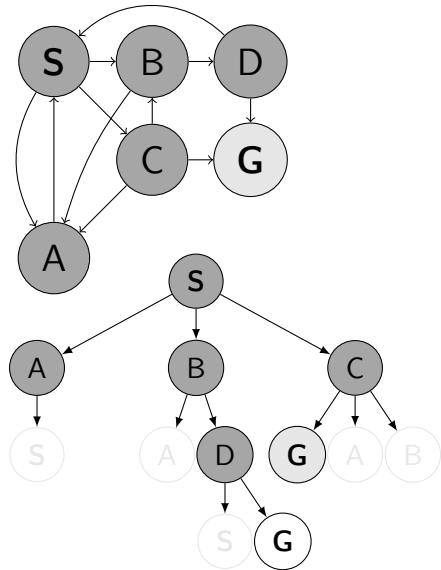
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

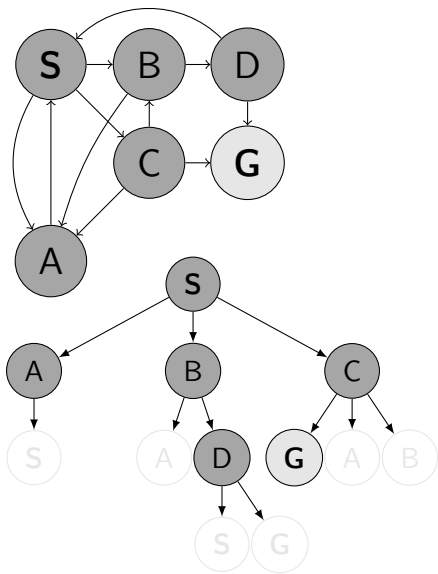
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,G)
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu.

Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy.

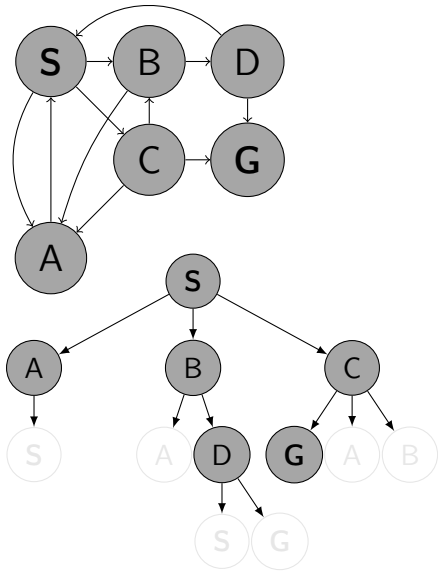
Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je?

Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

Do šířky (BFS): Q je FIFO (fronta)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q:
visited: S A B C D G

11 / 33

Notes

- Co dělá funkce/metoda Q.pop()?
- Potřebujeme nějak řešit duplikáty? Pokud ne, proč?
- Můžeme algoritmus zastavit a oznámit úspěch dříve?
- Jak nakonec vytvořit cestu?

Tento slide je klíčový k pochopení rozdílu mezi prohledáváním grafu a prohledáváním stromu. Vytvořte si strom prohledávání pomocí tužky, zaměřte se na Q a navštívené stavy. Jaké by bylo vhodné jméno pro datovou strukturu Q? Jaká datová struktura to vlastně je? Během vytváření prohledávacího stromu:

- bílý - výsledek (result) přechodu z daného stavu pomocí dané akce
- šedý - navštívený a uvnitř Q
- tmavě šedý - navštívený a prozkoumaný (už není v Q)
- zneviditelněný - zahozený

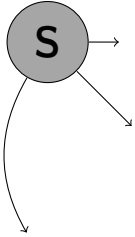
Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny



Najdete pro ně vhodná jména?

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

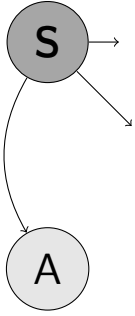
Najdete pro ně vhodná jména?



Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

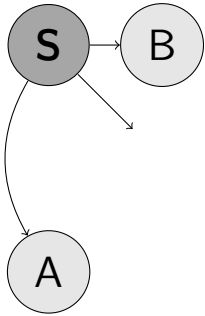
Najdete pro ně vhodná jména?



Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

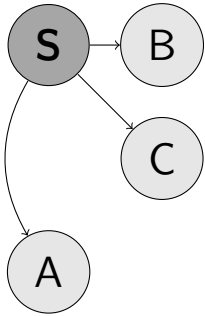
Najdete pro ně vhodná jména?



Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

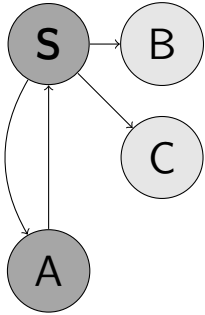
Najdete pro ně vhodná jména?



Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

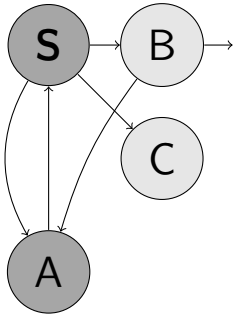
Najdete pro ně vhodná jména?



Notes

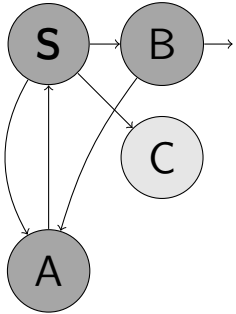
Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

Najdete pro ně vhodná jména?



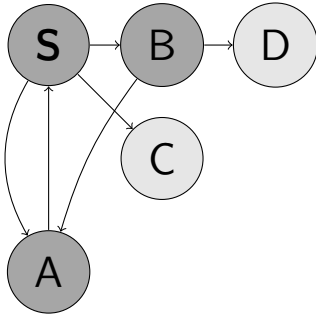
Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

Najdete pro ně vhodná jména?



Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

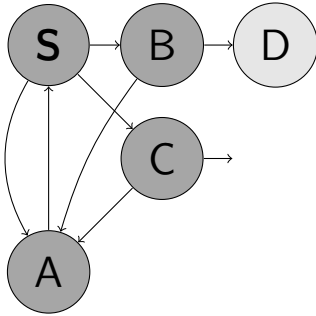
Najdete pro ně vhodná jména?



Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

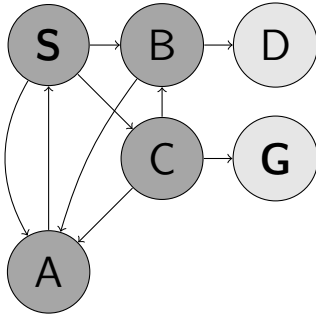
Najdete pro ně vhodná jména?



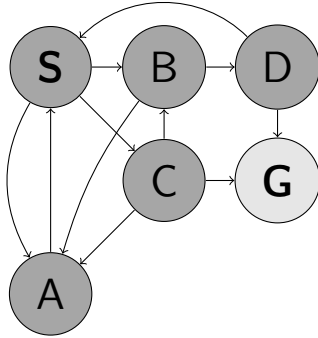
Notes

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny

Najdete pro ně vhodná jména?

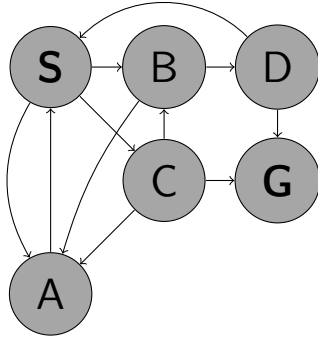


Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny



Najdete pro ně vhodná jména?

Prohledávací algoritmus dělí stavový prostor na 3 disjunktní množiny



Najdete pro ně vhodná jména?

Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

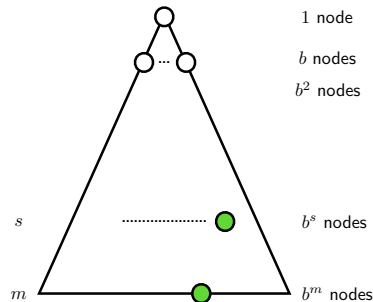
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

► Najde určité řešení (pokud existuje)? Úplný?

► Najde určité řešení s nejnižší cenou? Optimální?

► Kolik kroků (operací s uzlem) potřebuje? Časová složitost?

► Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

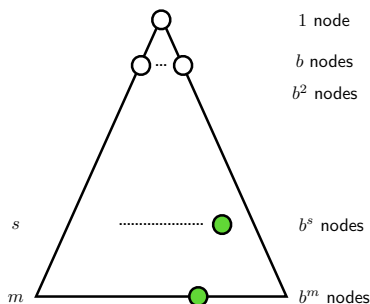
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

► Najde určité řešení (pokud existuje)? Úplný?

► Najde určité řešení s nejnižší cenou? Optimální?

► Kolik kroků (operací s uzlem) potřebuje? Časová složitost?

► Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

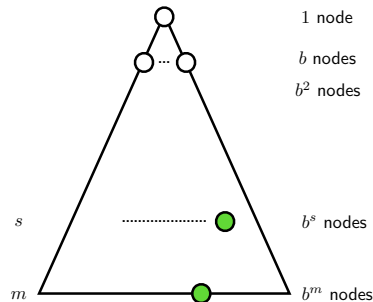
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

► Najde určité řešení (pokud existuje)? Úplný?

► Najde určité řešení s nejnižší cenou? Optimální?

► Kolik kroků (operací s uzlem) potřebuje? Časová složitost?

► Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

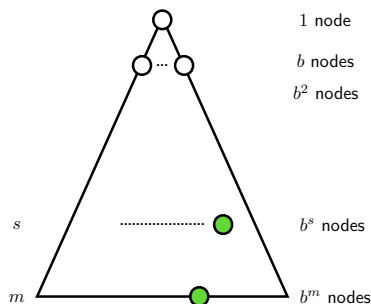
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

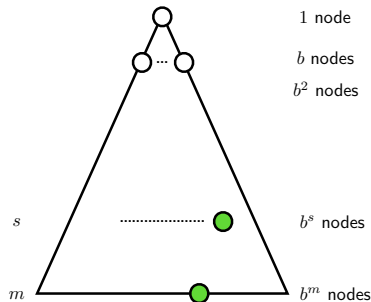
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali velikost stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

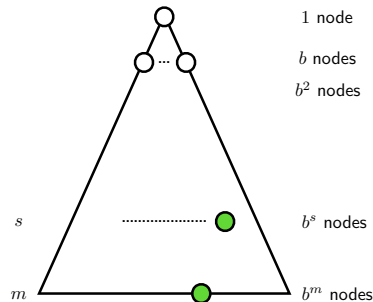
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali velikost stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

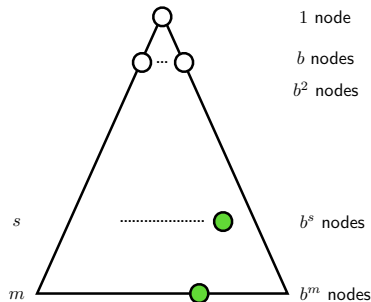
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali velikost stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

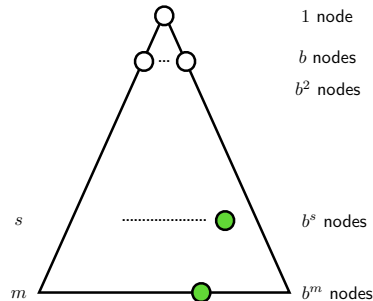
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

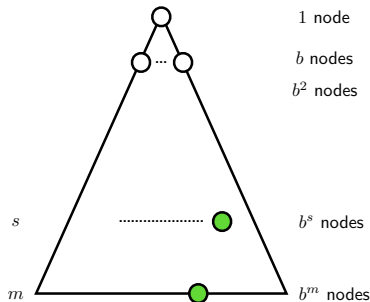
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali velikost stromu.

- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Vlastnosti prohledávání (prohledávacího algoritmu)

- ▶ Najde určité řešení (pokud existuje)? Úplný?
- ▶ Najde určité řešení s nejnižší cenou? Optimální?
- ▶ Kolik kroků (operací s uzlem) potřebuje? Časová složitost?
- ▶ Kolik uzlů si musí pamatovat? Prostorová/Paměťová složitost?

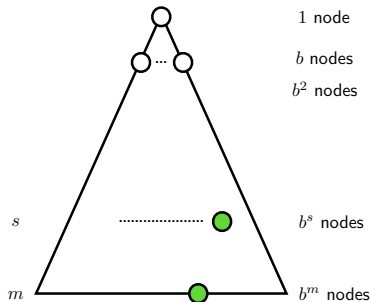
Kolik je uzlů v prohledávacím stromě? Jaké jsou parametry prohledávacího stromu?

13 / 33

Notes

Načrtni prohledávací strom (symbolicky, jako trojúhelník). Může růst nahoru nebo dolů. Jak byste charakterizovali/parametrizovali *velikost* stromu.

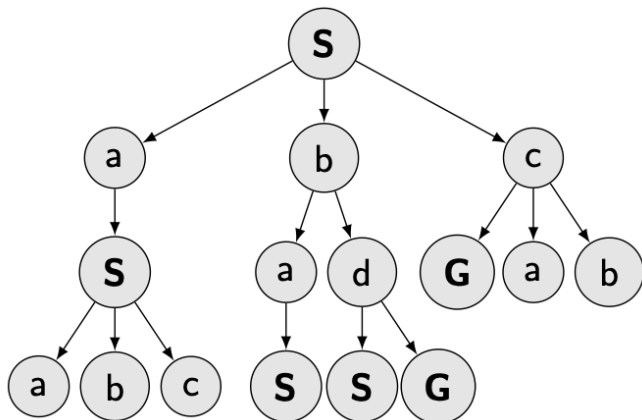
- Hloubka d uzlu ve stromě.
- Maximální hloubka stromu m . Může být ∞ .
- (Průměrný) faktor větvení b .
- s je nejmenší hloubka, v níž leží některý z Cílů.
- Kolik je uzlů v celém stromě?



Prohledávací strategie

Jak procházet/stavět prohledávací strom?

- ▶ **Hloubka** d uzlu ve stromě.
- ▶ **Maximální hloubka** stromu m .
Může být ∞ .
- ▶ (Průměrný) **faktor větvení** b .
- ▶ s je nejmenší hloubka, v níž leží některý z Cílů.
- ▶ Kolik je uzlů v celém stromě?



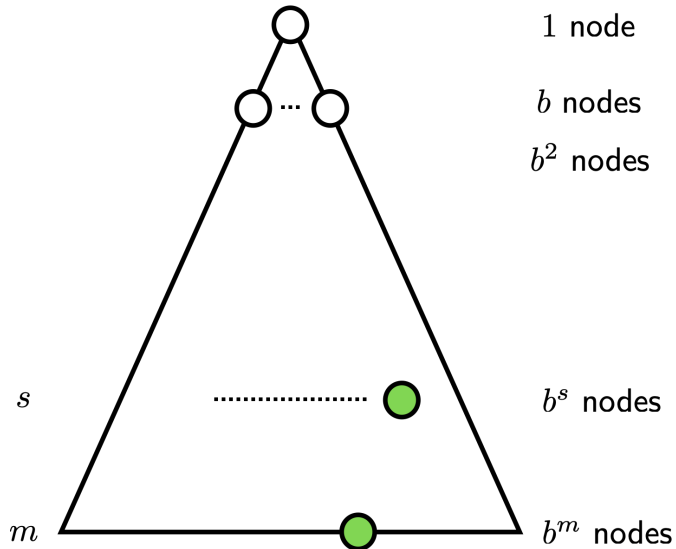
Notes

Pamatujte, že prohledávací strom se staví za běhu prohledávacího algoritmu. Nebo, lépe řečeno, strom je člověku srozumitelná reprezentace běhu algoritmu. I malé grafy (problémy) mohou vyústit ve velký prohledávací strom – záleží na prohledávacím algoritmu.

Prohledávací strategie

Jak procházet/stavět prohledávací strom?

- ▶ **Hloubka** d uzlu ve stromě.
- ▶ **Maximální hloubka** stromu m .
Může být ∞ .
- ▶ (Průměrný) **faktor větvení** b .
- ▶ s je nejmenší hloubka, v níž leží některý z Cílů.
- ▶ Kolik je uzlů v celém stromě?



14 / 33

Notes

Pamatujte, že prohledávací strom se staví za běhu prohledávacího algoritmu. Nebo, lépe řečeno, strom je člověku srozumitelná reprezentace běhu algoritmu. I malé grafy (problémy) mohou vyústit ve velký prohledávací strom – záleží na prohledávacím algoritmu.

Vlastnosti algoritmu prohl. do šířky (BFS)

Úplný?

Optimální?

Časová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

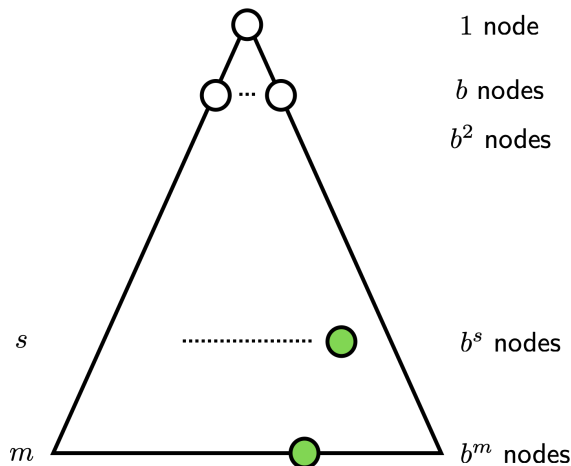
Prostorová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

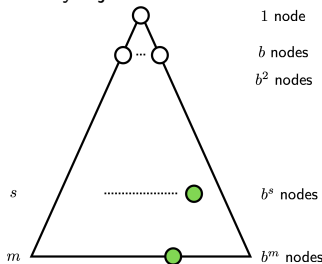


15 / 33

Notes

- Čas: musí projít celý strom až do hloubky s : b^s . Přesněji, musí projít $b + b^2 + b^3 + \dots + b^s$, ale poslední vrstva naprosto dominuje. Vyzkoušejte pár výpočtů pro různá b .
- Prostor: všechny otevřené uzly (frontier), b^s
- Úplnost: Ano!
- Optimalita: BFS najde řešení v nejmenší hloubce; pokud všechny přechody mají cenu 1, pak ano!

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Vlastnosti algoritmu prohl. do šířky (BFS)

Úplný?

Optimální?

Časová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

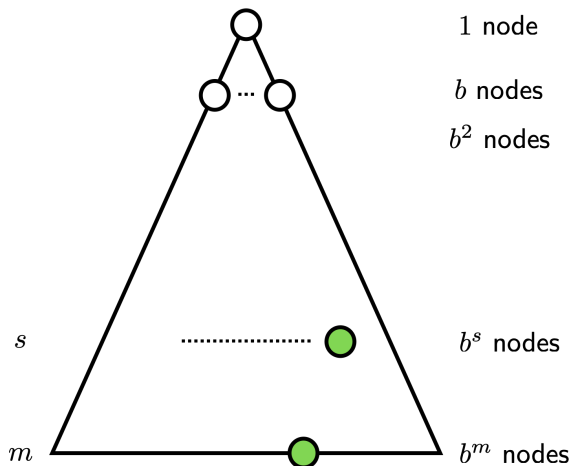
Prostorová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

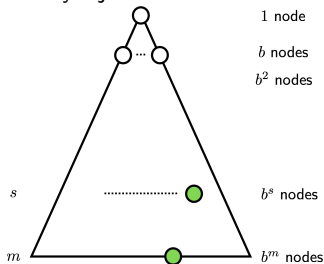


15 / 33

Notes

- Čas: musí projít celý strom až do hloubky s : b^s . Přesněji, musí projít $b + b^2 + b^3 + \dots + b^s$, ale poslední vrstva naprosto dominuje. Vyzkoušejte pár výpočtů pro různá b .
- Prostor: všechny otevřené uzly (frontier), b^s
- Úplnost: Ano!
- Optimalita: BFS najde řešení v nejmenší hloubce; pokud všechny přechody mají cenu 1, pak ano!

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Vlastnosti algoritmu prohl. do šířky (BFS)

Úplný?

Optimální?

Časová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

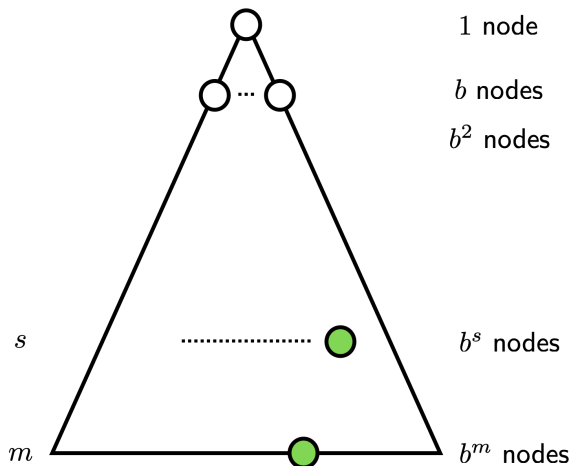
Prostorová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

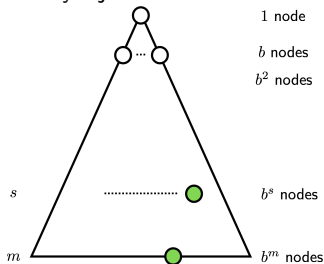


15 / 33

Notes

- Čas: musí projít celý strom až do hloubky s : b^s . Přesněji, musí projít $b + b^2 + b^3 + \dots + b^s$, ale poslední vrstva naprosto dominuje. Vyzkoušejte pár výpočtů pro různá b .
- Prostor: všechny otevřené uzly (frontier), b^s
- Úplnost: Ano!
- Optimalita: BFS najde řešení v nejmenší hloubce; pokud všechny přechody mají cenu 1, pak ano!

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Vlastnosti algoritmu prohl. do šířky (BFS)

Úplný?

Optimální?

Časová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

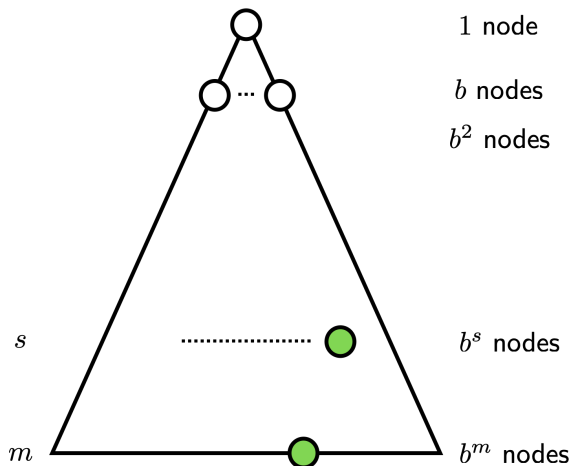
Prostorová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

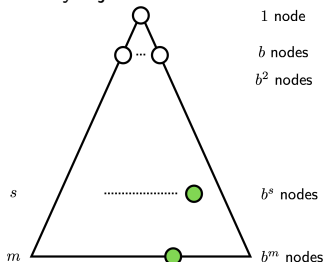


15 / 33

Notes

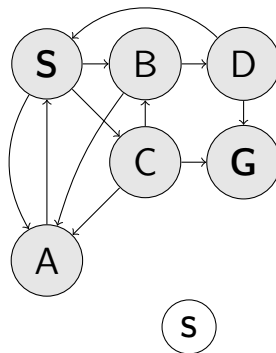
- Čas: musí projít celý strom až do hloubky s : b^s . Přesněji, musí projít $b + b^2 + b^3 + \dots + b^s$, ale poslední vrstva naprosto dominuje. Vyzkoušejte pár výpočtů pro různá b .
- Prostor: všechny otevřené uzly (frontier), b^s
- Úplnost: Ano!
- Optimalita: BFS najde řešení v nejmenší hloubce; pokud všechny přechody mají cenu 1, pak ano!

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:         Označ s' jako navštívený
11:         Q.insert(s, s')
12:       else
13:         Vyřeš duplikaci s' v Q
return Failure
```



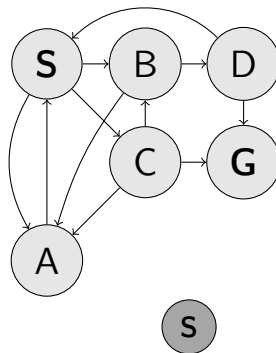
Q: (_, S)
visited: S

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



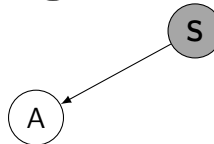
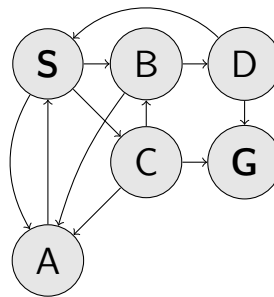
Q:
visited: S

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



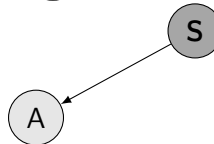
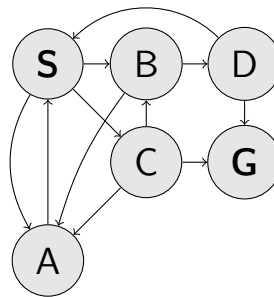
Q:
visited: S

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



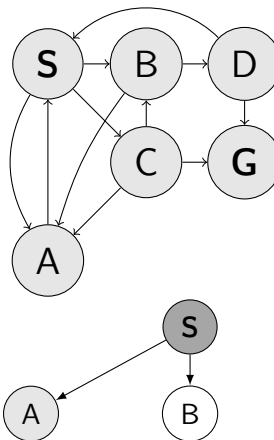
Q: (S,A)
visited: S A

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



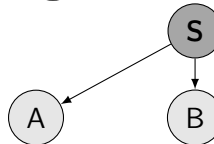
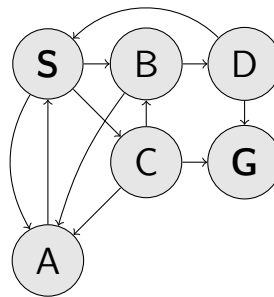
Q: (S,A)
visited: S A

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



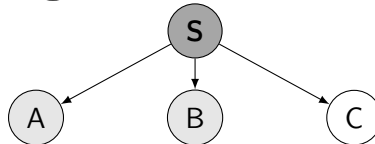
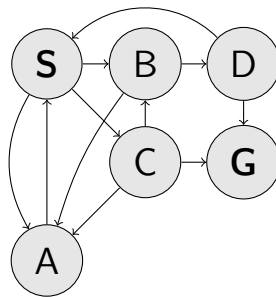
Q: (S,A) (S,B)
visited: S A B

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



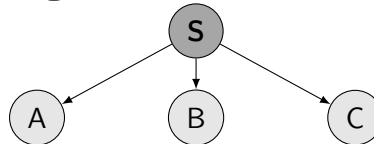
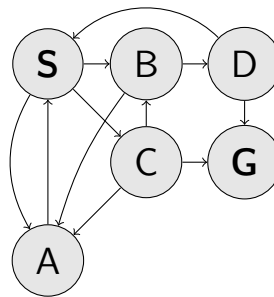
Q: (S,A) (S,B)
visited: S A B

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



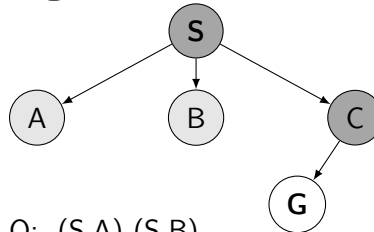
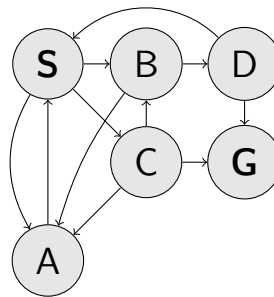
Q: (S,A) (S,B) (S,C)
visited: S A B C

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



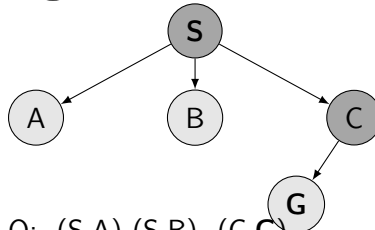
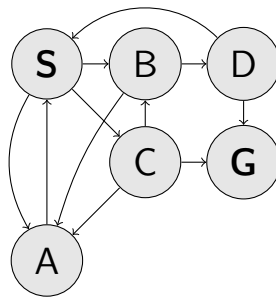
Q: (S,A) (S,B)
visited: S A B C

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



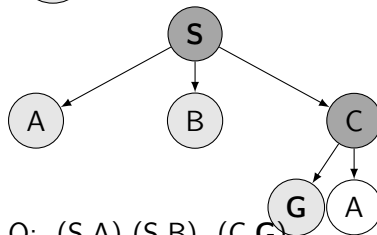
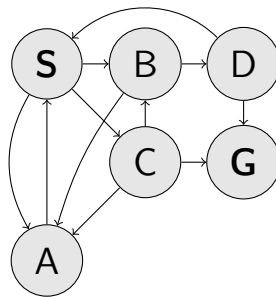
Q: (S,A) (S,B) (C,G)
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



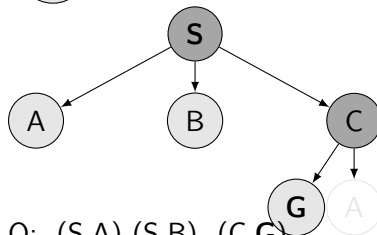
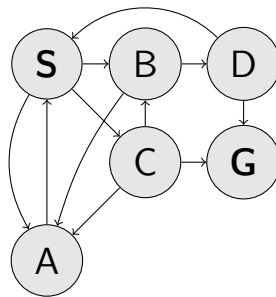
Q: (S,A) (S,B) (C,G)
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



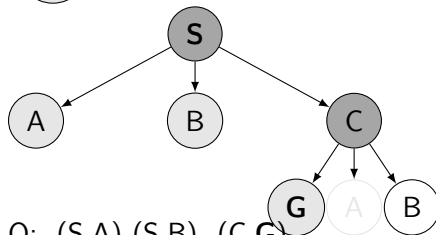
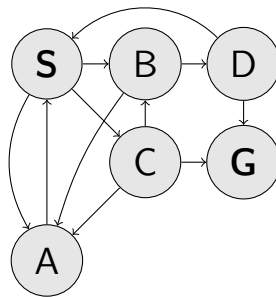
Q: (S,A) (S,B) (C,G)
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B) (C,G)

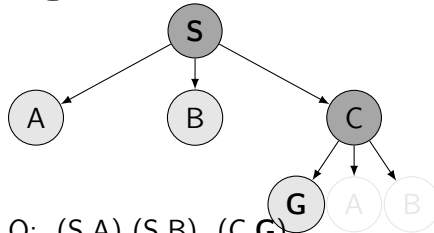
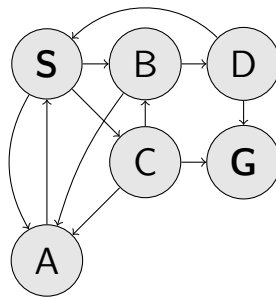
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B) (C,G)

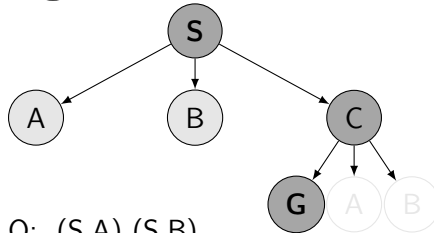
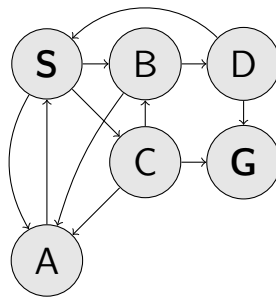
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Do hloubky (DFS): Q je LIFO (zásobník)

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s ← Q.pop()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s' ← result(s, a)
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s')
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A) (S,B)
visited: S A B C G

Notes

Potřebujeme nějak řešit duplikáty? Pokud ne, proč?

Vlastnosti algoritmu prohl. do hloubky (DFS)

Úplný?

Optimální?

Časová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

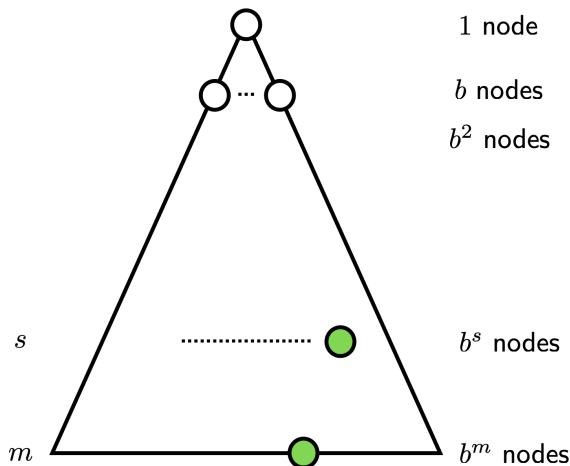
Prostorová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

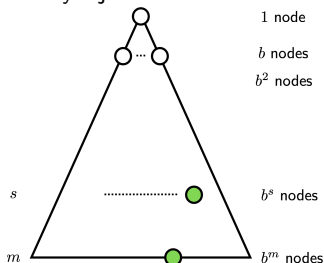


17 / 33

Notes

- Čas: může být potřeba projít celý strom, b^m
- Prostor: pouze aktuální cesta, bm (cesta z kořene do listu (m), plus "sourozenci" uzlů v cestě, kteří jsou taky otevření ($b \times m$))
- Úplnost: m může být ∞ , takže obecně ne!
- Optimalita: Ne! Vráť první nalezené řešení.

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Vlastnosti algoritmu prohl. do hloubky (DFS)

Úplný?

Optimální?

Časová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

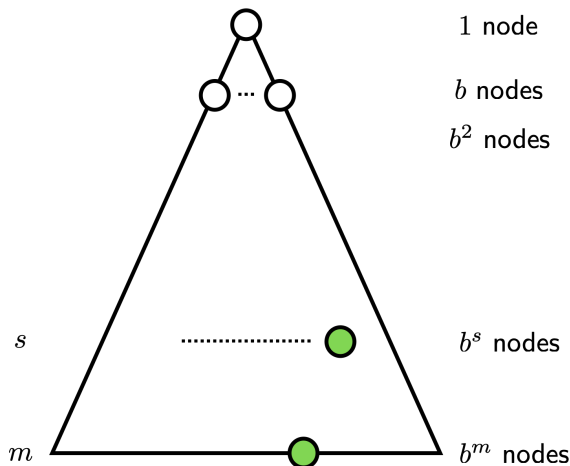
Prostorová složitost?

A $O(bm)$

B $O(b^m)$

C $O(m^b)$

D $O(b^s)$

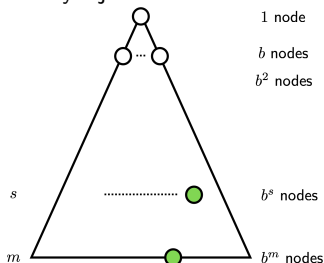


17 / 33

Notes

- Čas: může být potřeba projít celý strom, b^m
- Prostor: pouze aktuální cesta, bm (cesta z kořene do listu (m), plus "sourozenci" uzlů v cestě, kteří jsou taky otevření ($b \times m$))
- Úplnost: m může být ∞ , takže obecně ne!
- Optimalita: Ne! Vráť první nalezené řešení.

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



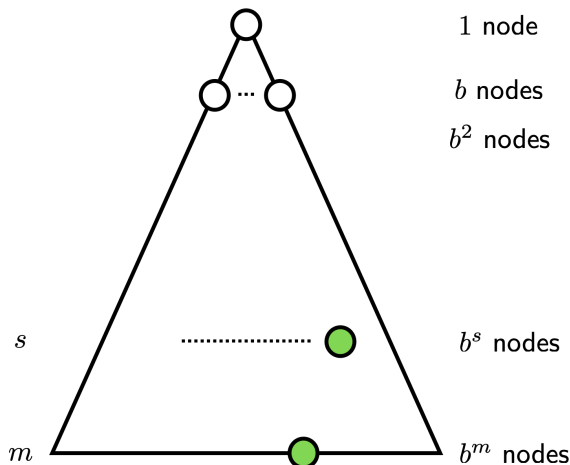
Vlastnosti algoritmu prohl. do hloubky (DFS)

Úplný?
Optimální?
Časová složitost?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$

Prostorová složitost?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$

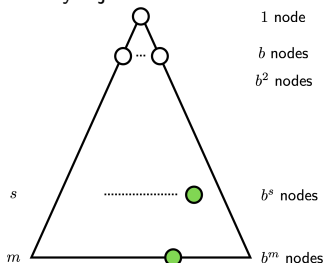


17 / 33

Notes

- Čas: může být potřeba projít celý strom, b^m
- Prostor: pouze aktuální cesta, bm (cesta z kořene do listu (m), plus "sourozenci" uzlů v cestě, kteří jsou taky otevření ($b \times m$))
- Úplnost: m může být ∞ , takže obecně ne!
- Optimalita: Ne! Vráť první nalezené řešení.

Premýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



Vlastnosti algoritmu prohl. do hloubky (DFS)

Úplný?

Optimální?

Časová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

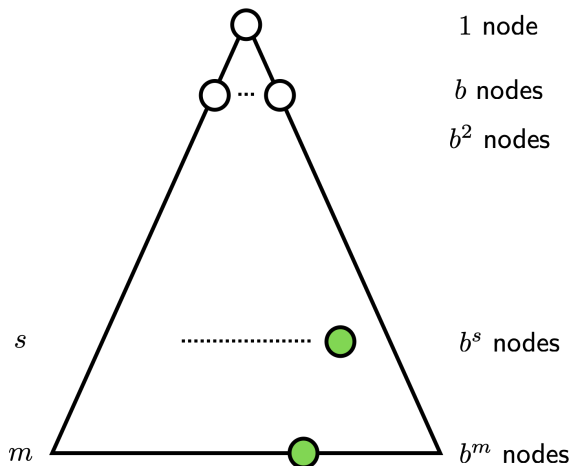
Prostorová složitost?

A $\mathcal{O}(bm)$

B $\mathcal{O}(b^m)$

C $\mathcal{O}(m^b)$

D $\mathcal{O}(b^s)$

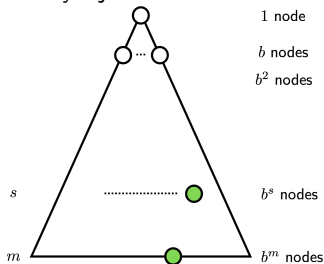


17 / 33

Notes

- Čas: může být potřeba projít celý strom, b^m
- Prostor: pouze aktuální cesta, bm (cesta z kořene do listu (m), plus "sourozenci" uzlů v cestě, kteří jsou taky otevření ($b \times m$))
- Úplnost: m může být ∞ , takže obecně ne!
- Optimalita: Ne! Vráť první nalezené řešení.

Přemýšlejte o složitostech vzhledem k $|\mathcal{S}|$ a $|\mathcal{A}|$ (Teorie grafů: vrcholy, hrany)



DFS s iterativním prohlubováním (ID-DFS)

► Začněte s `maxdepth = 1`

► Spusťte DFS s omezenou hloubkou. Bylo nalezeno řešení?

► Pokud řešení nebylo nalezeno, vše zapomeňte, zvyšte `maxdepth` a jděte na předchozí krok.

Zapomínání všeho mezi jednotlivými kroky - není to obrovské plýtvání?

18 / 33

Notes

Kolik výpočtů opakujeme? Jak moc plýtváme? „Horní úrovně“, které jsou blízko u kořene, se opakují mnohokrát. Na druhou stranu, ve stromě je většina uzlů v dolních úrovních, a počítá se počet prošlých uzlů. Přesněji, pro řešení v hloubce s , uzly poslední úrovně (v hloubce s) jsou vygenerovány pouze jednou, uzly na předposlední úrovni $2 \times$, ... potomci kořenového uzlu jsou vygenerováni $s \times$. Porovnejme počty uzlů vygenerovaných algoritmy ID-DFS a BFS:

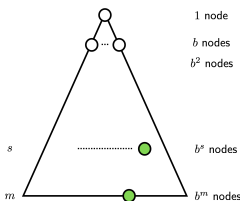
$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \dots + (1)b^s$$

$$N(\text{BFS}) = b + b^2 + b^3 + \dots + b^s$$

Vyzkoušejte nějaké výpočty pro různé s a b pro $b = 10$ a $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$



(Příklad z [2].)

DFS s iterativním prohlubováním (ID-DFS)

- ▶ Začněte s `maxdepth = 1`
- ▶ Spusťte DFS s omezenou hloubkou. Bylo nalezeno řešení?
 - ▶ Pokud řešení nebylo nalezeno, vše zapomeňte, zvyšte `maxdepth` a jděte na předchozí krok.

Zapomínání všeho mezi jednotlivými kroky - není to obrovské plýtvání?

18 / 33

Notes

Kolik výpočtů opakujeme? Jak moc plýtváme? „Horní úroveň“, které jsou blízko u kořene, se opakují mnohokrát. Na druhou stranu, ve stromě je většina uzlů v dolních úrovních, a počítá se počet prošlých uzlů. Přesněji, pro řešení v hloubce s , uzly poslední úrovně (v hloubce s) jsou vygenerovány pouze jednou, uzly na předposlední úrovni $2x$, ... potomci kořenového uzlu jsou vygenerovány $s \times$. Porovnejme počty uzlů vygenerovaných algoritmy ID-DFS a BFS:

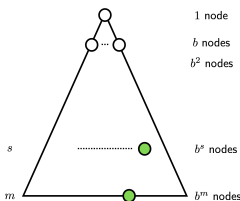
$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \dots + (1)b^s$$

$$N(\text{BFS}) = b + b^2 + b^3 + \dots + b^s$$

Vyzkoušejte nějaké výpočty pro různé s a b pro $b = 10$ a $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$



(Příklad z [2].)

DFS s iterativním prohlubováním (ID-DFS)

- ▶ Začněte s `maxdepth = 1`
- ▶ Spusťte DFS s omezenou hloubkou. Bylo nalezeno řešení?
- ▶ Pokud řešení nebylo nalezeno, vše zapomeňte, zvyšte `maxdepth` a jděte na předchozí krok.

Zapominání všeho mezi jednotlivými kroky - není to obrovské plýtvání?

18 / 33

Notes

Kolik výpočtů opakujeme? Jak moc plýtváme? „Horní úrovně“, které jsou blízko u kořene, se opakují mnohokrát. Na druhou stranu, ve stromě je většina uzlů v dolních úrovních, a počítá se počet prošlých uzlů. Přesněji, pro řešení v hloubce s , uzly poslední úrovně (v hloubce s) jsou vygenerovány pouze jednou, uzly na předposlední úrovni $2x$, ... potomci kořenového uzlu jsou vygenerováni $s \times$. Porovnejme počty uzlů vygenerovaných algoritmy ID-DFS a BFS:

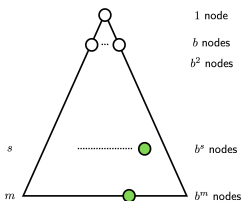
$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \dots + (1)b^s$$

$$N(\text{BFS}) = b + b^2 + b^3 + \dots + b^s$$

Vyzkoušejte nějaké výpočty pro různé s a b pro $b = 10$ a $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$



(Příklad z [2].)

DFS s iterativním prohlubováním (ID-DFS)

- ▶ Začněte s `maxdepth = 1`
- ▶ Spusťte DFS s omezenou hloubkou. Bylo nalezeno řešení?
- ▶ Pokud řešení nebylo nalezeno, vše zapomeňte, zvýšte `maxdepth` a jděte na předchozí krok.

Zapomínání všeho mezi jednotlivými kroky - není to obrovské plýtvání?

18 / 33

Notes

Kolik výpočtů opakujeme? Jak moc plýtváme? „Horní úrovně“, které jsou blízko u kořene, se opakují mnohokrát. Na druhou stranu, ve stromě je většina uzlů v dolních úrovních, a počítá se počet prošlých uzlů. Přesněji, pro řešení v hloubce s , uzly poslední úrovně (v hloubce s) jsou vygenerovány pouze jednou, uzly na předposlední úrovni $2 \times$, ... potomci kořenového uzlu jsou vygenerovány $s \times$. Porovnejme počty uzlů vygenerovaných algoritmy ID-DFS a BFS:

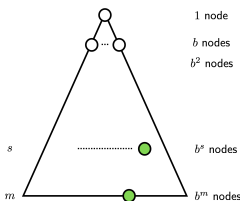
$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \dots + (1)b^s$$

$$N(\text{BFS}) = b + b^2 + b^3 + \dots + b^s$$

Vyzkoušejte nějaké výpočty pro různé s a b pro $b = 10$ a $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

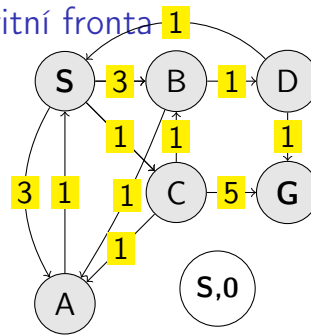


(Příklad z [2].)

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (_, S, 0)
visited: S

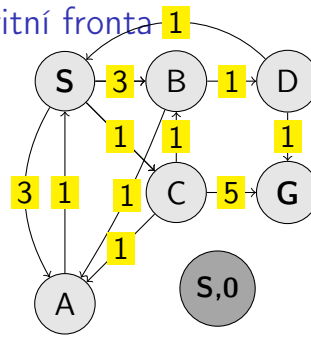
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q:
visited: S

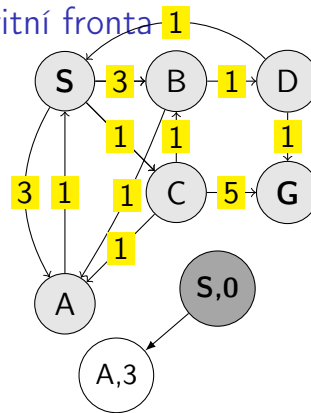
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q:
visited: S

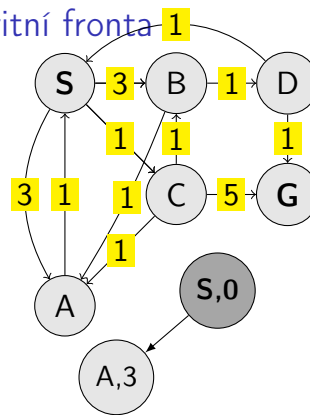
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3)

visited: S A

Notes

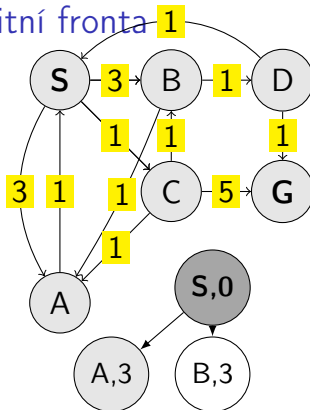
- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

1

```

1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure

```



Q: (S,A,3)
visited: **S** A

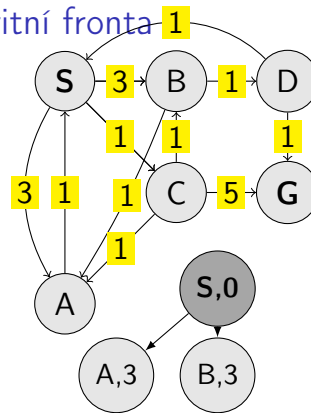
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá `cost_from_start`?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3)
visited: S A B

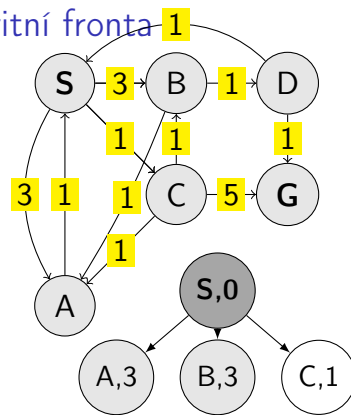
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3)

visited: S A B

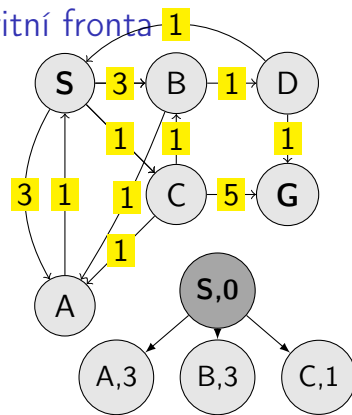
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,C,1) (S,A,3) (S,B,3)
visited: S A B C

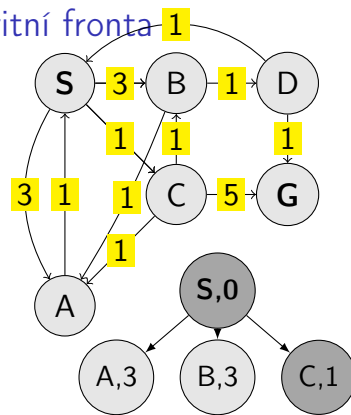
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3)

visited: S A B C

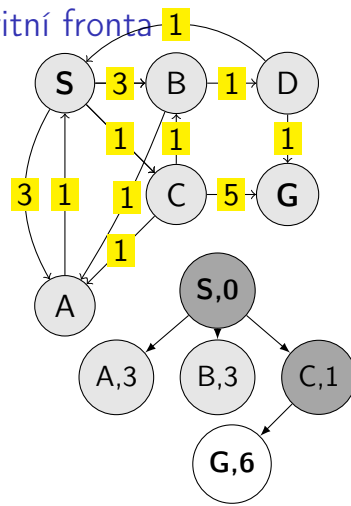
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3)

visited: S A B C

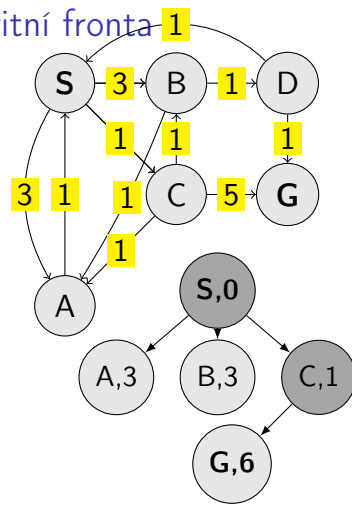
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3) (C,G,6)
visited: S A B C G

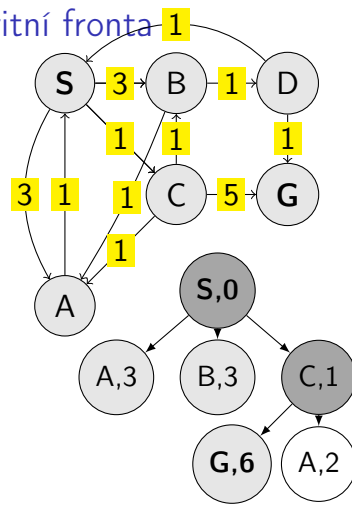
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



Q: (S,A,3) (S,B,3) (C,G,6)
visited: S A B C G

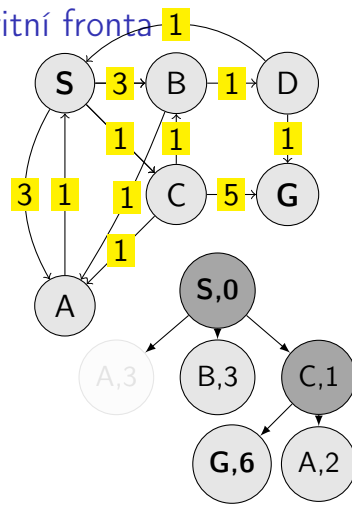
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,A,2) (S,B,3) (C,G,6)
visited: S A B C G

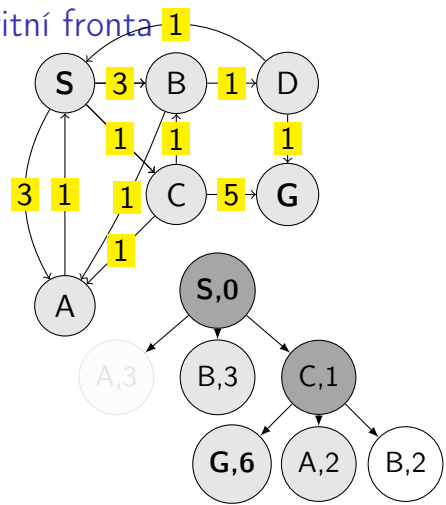
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,A,2) (S,B,3) (C,G,6)
visited: S A B C G

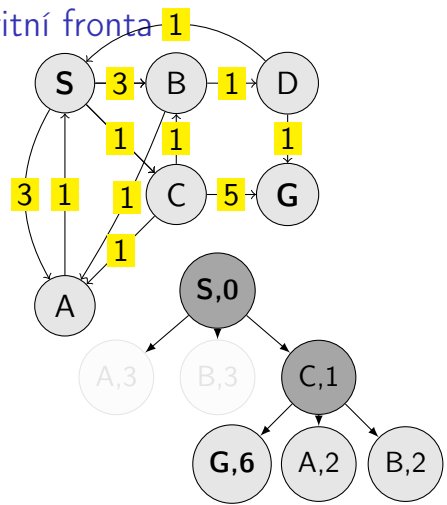
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,A,2) (C,B,2) (C,G,6)
visited: S A B C G

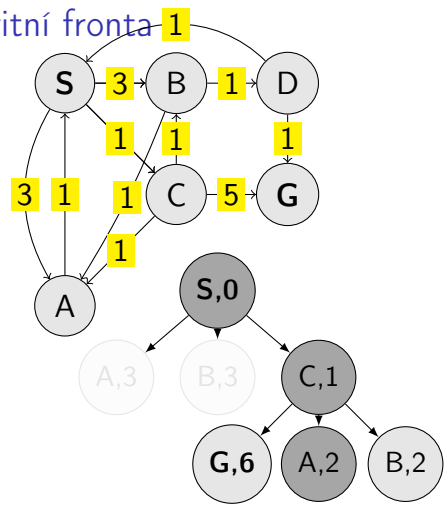
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Q: (C,B,2) (C,G,6)
visited: S A B C G

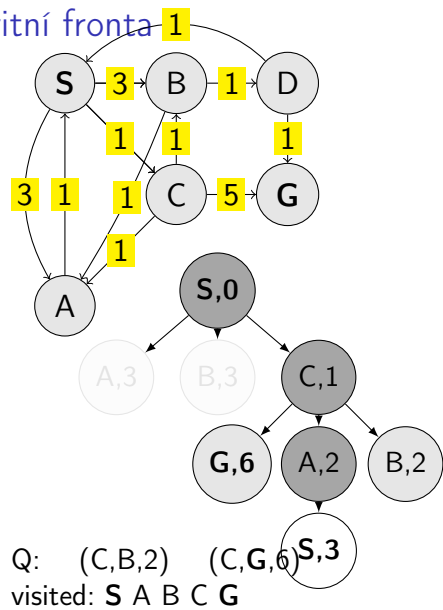
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



Notes

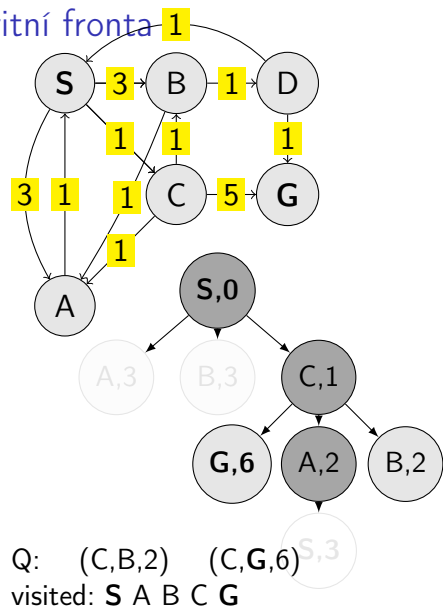
- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```

1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
    
```



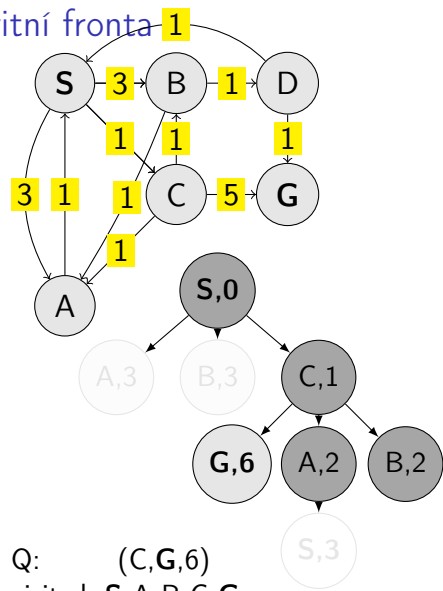
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



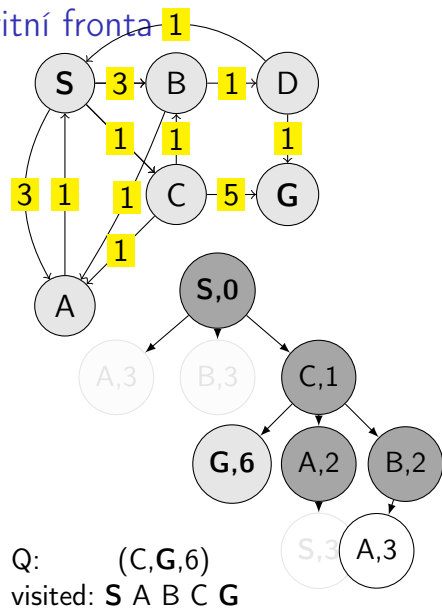
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



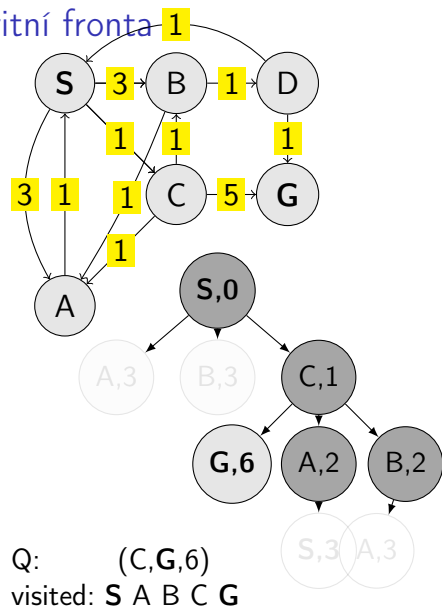
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



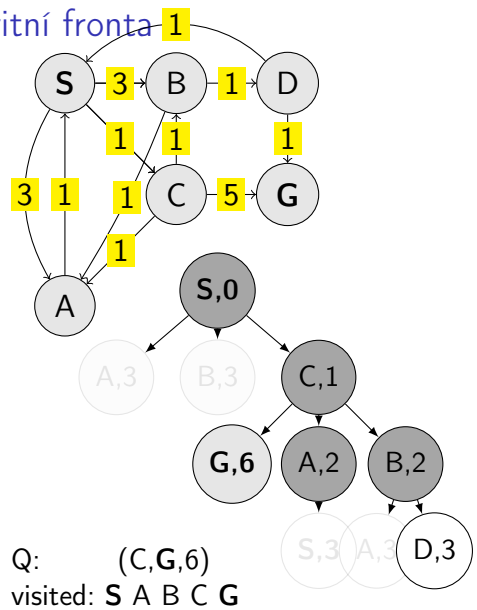
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



19 / 33

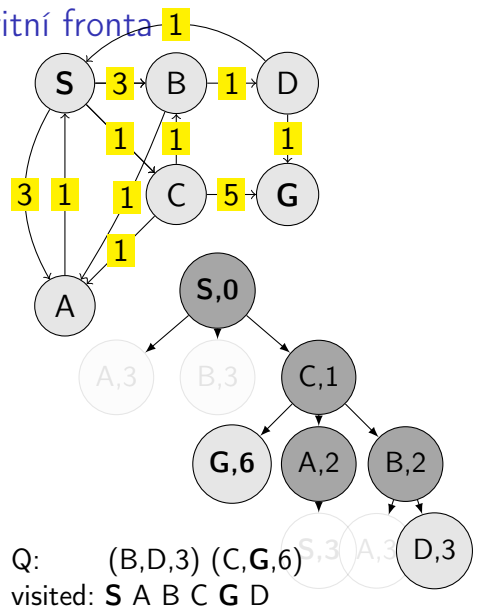
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:     else
13:       Vyřeš duplikaci s' v Q
return Failure
```



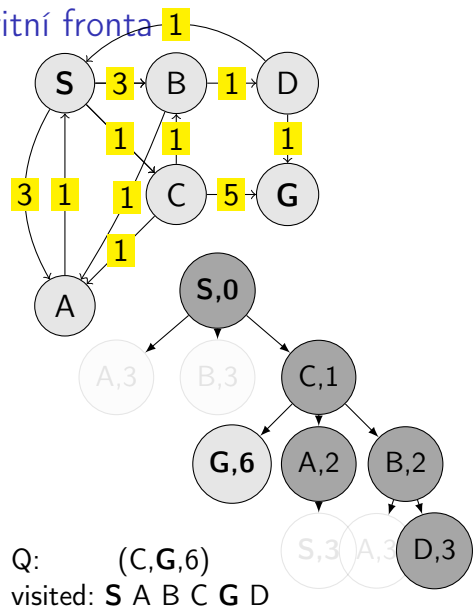
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



19 / 33

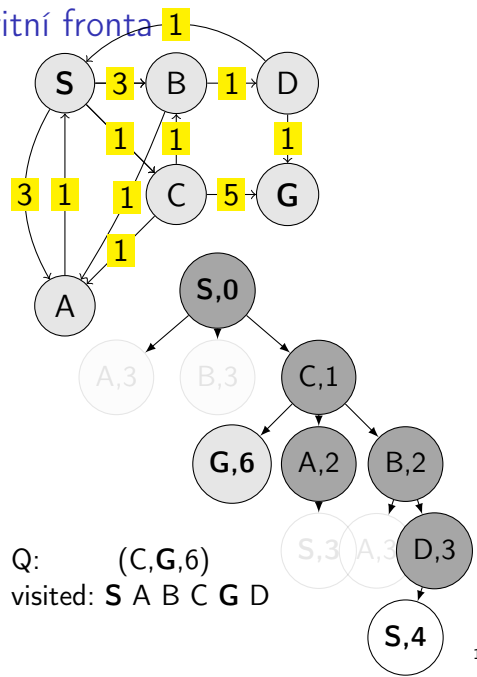
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



19 / 33

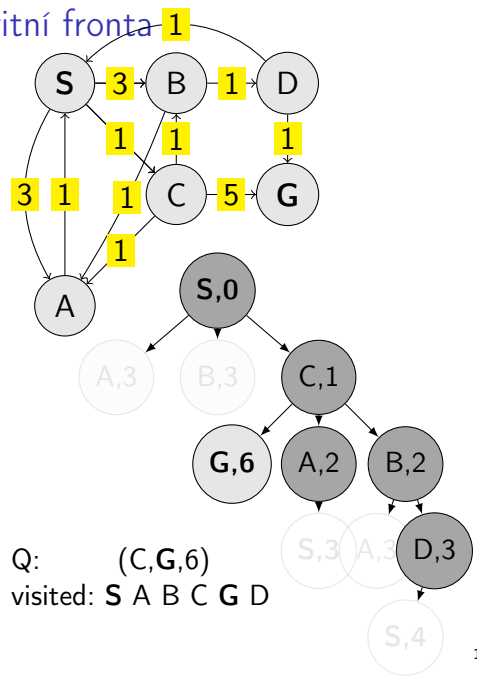
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



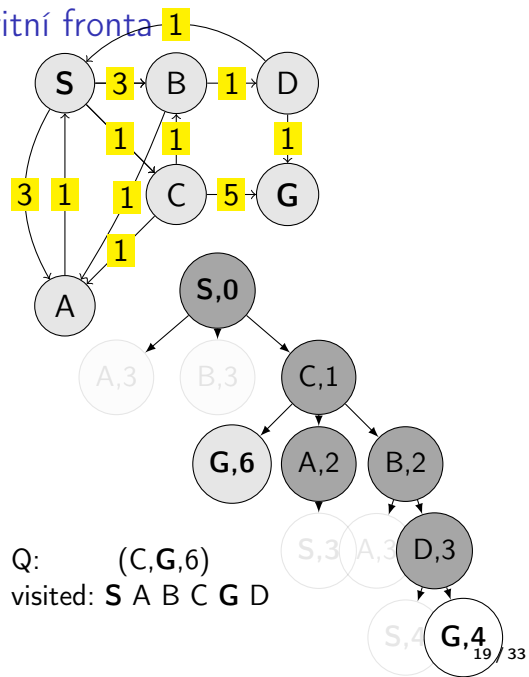
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



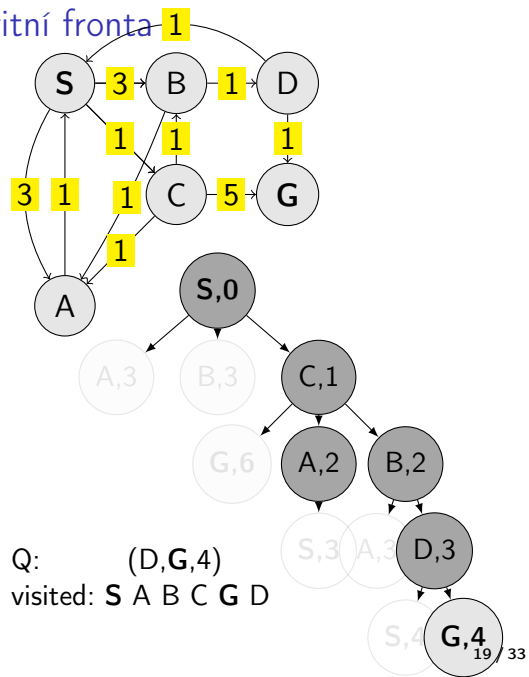
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnoměrnou cenou

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



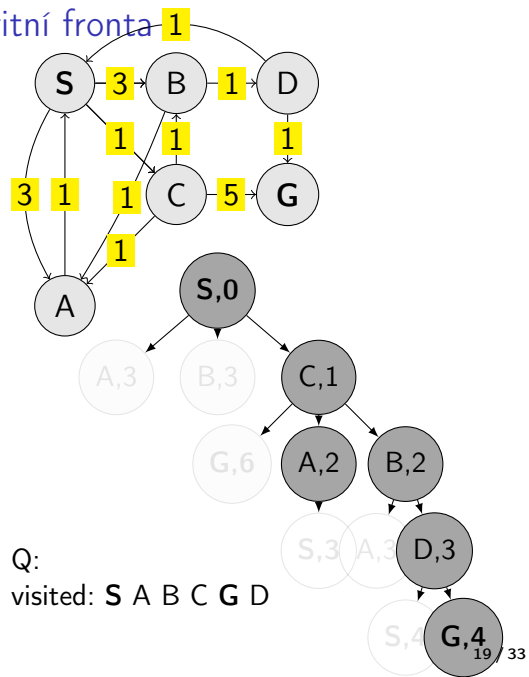
Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnoměrnou cenou?

Uniform Cost Search (Dijkstra): Q je prioritní fronta

Prohledávání se stejnou cenou

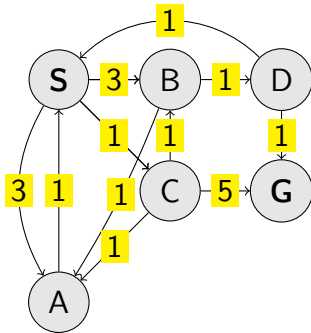
```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0) a označ s0 jako navštívený
3:   while Q není prázdná do
4:     p, s, _ ← Q.pop_first()
5:     parent[s] ← p
6:     if s ∈ SG then return Success
7:     for all a ∈ A(s) do
8:       s', c ← result(s, a)      ▷ c je cena
9:       if s' není navštívený then
10:        Označ s' jako navštívený
11:        Q.insert(s, s', cost_from_start)
12:       else
13:        Vyřeš duplikaci s' v Q
return Failure
```



Notes

- Je potřeba nějak řešit duplikaci stavů v Q? Pokud ne, proč?
- Jak se počítá cost_from_start?
- Proč se algoritmu říká prohledávání se stejnou cenou?

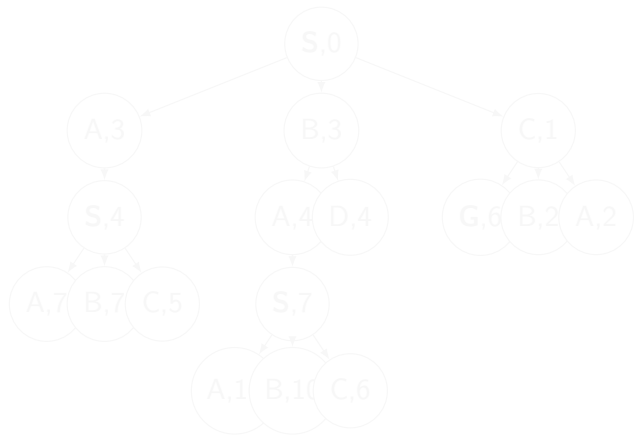
Vlastnosti algoritmu UCS



Úplný?

Optimální?

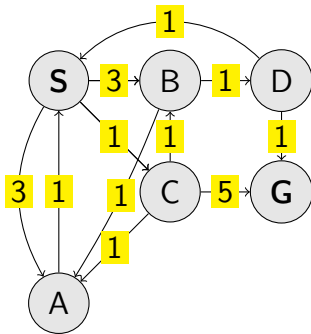
Časová a prostorová složitost?



Notes

Části (úplného) prohledávacího stromu se opakují, ale s různými cenami

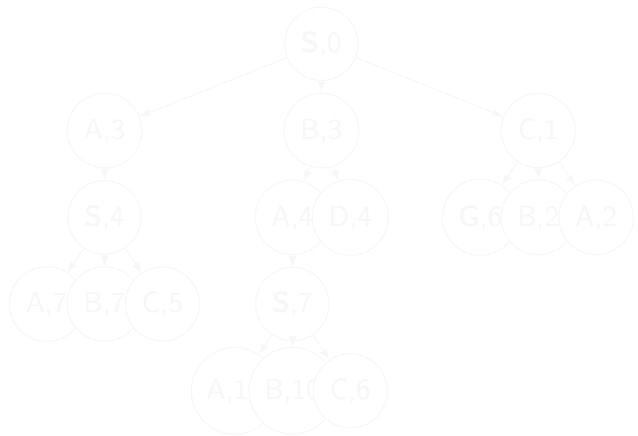
Vlastnosti algoritmu UCS



Úplný?

Optimální?

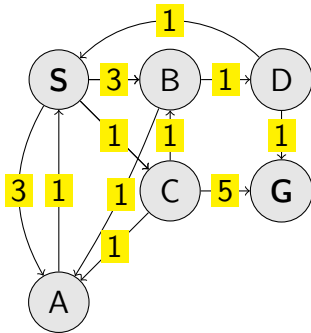
Časová a prostorová složitost?



Notes

Části (úplného) prohledávacího stromu se opakují, ale s různými cenami

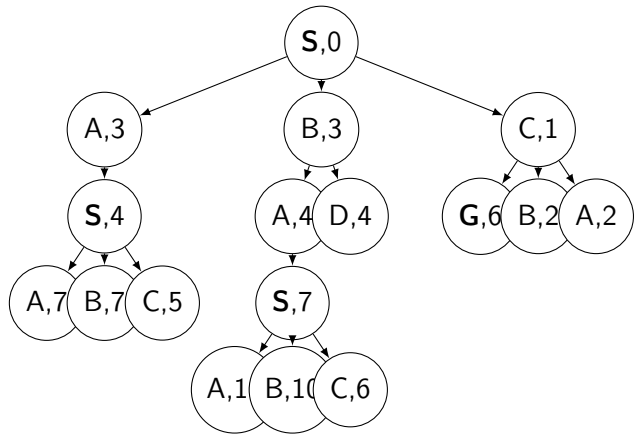
Vlastnosti algoritmu UCS



Úplný?

Optimální?

Časová a prostorová složitost?



Notes

Části (úplného) prohledávacího stromu se opakují, ale s různými cenami

Výběr uzlu pomocí $\text{argmin } f(n)$. Prohledávaný uzel: $n = (p, s, \text{cost_value})$

Výběr dalšího uzlu k prozkoumání (operace pop):

$$\text{node} \leftarrow \underset{n \in Q}{\text{argmin}} f(n)$$

Jakou funkci $f(n)$ používají algoritmy DFS, BFS a UCS? Spárujte je!

- | | |
|--------|---------------------------------------|
| ▶ DFS: | ▶ $f(n) = n.\text{cost_from_start}$ |
| ▶ BFS: | ▶ $f(n) = n.\text{depth}$ |
| ▶ UCS: | ▶ $f(n) = -n.\text{depth}$ |

Výhoda: prioritní fronta jako univerzální datová struktura pro otevřené uzly (frontier)
(Přesto, zásobník(LIFO) a fronta (FIFO) jsou konceptuálně perfektní datové struktury pro DFS a BFS.)

Nevýhoda: všechny uvedené $f(n)$ odpovídají součtu cen ze startu do n , cost_from_start . Co nám schází?

21 / 33

Notes

- DFS: $f(n) = -n.\text{depth}$
- BFS: $f(n) = n.\text{depth}$
- UCS: $f(n) = n.\text{path_cost}$

Dívají se lidé zpátky, když plánují cestu? Je pohled zpátky vůbec důležitý? Pokud ano, kdy?

Výběr uzlu pomocí $\text{argmin } f(n)$. Prohledávaný uzel: $n = (p, s, \text{cost_value})$

Výběr dalšího uzlu k prozkoumání (operace pop):

$$\text{node} \leftarrow \underset{n \in Q}{\text{argmin}} f(n)$$

Jakou funkci $f(n)$ používají algoritmy DFS, BFS a UCS? Spárujte je!

- | | |
|--------|---------------------------------------|
| ▶ DFS: | ▶ $f(n) = n.\text{cost_from_start}$ |
| ▶ BFS: | ▶ $f(n) = n.\text{depth}$ |
| ▶ UCS: | ▶ $f(n) = -n.\text{depth}$ |

Výhoda: prioritní fronta jako univerzální datová struktura pro otevřené uzly (frontier)
(Přesto, zásobník(LIFO) a fronta (FIFO) jsou konceptuálně perfektní datové struktury pro DFS a BFS.)

Nevýhoda: všechny uvedené $f(n)$ odpovídají součtu cen ze startu do n , cost_from_start . Co nám schází?

21 / 33

Notes

- DFS: $f(n) = -n.\text{depth}$
- BFS: $f(n) = n.\text{depth}$
- UCS: $f(n) = n.\text{path_cost}$

Dívají se lidé zpátky, když plánují cestu? Je pohled zpátky vůbec důležitý? Pokud ano, kdy?

Výběr uzlu pomocí $\text{argmin } f(n)$. Prohledávaný uzel: $n = (p, s, \text{cost_value})$

Výběr dalšího uzlu k prozkoumání (operace pop):

$$\text{node} \leftarrow \underset{n \in Q}{\text{argmin}} f(n)$$

Jakou funkci $f(n)$ používají algoritmy DFS, BFS a UCS? Spárujte je!

- | | |
|--------|---------------------------------------|
| ▶ DFS: | ▶ $f(n) = n.\text{cost_from_start}$ |
| ▶ BFS: | ▶ $f(n) = n.\text{depth}$ |
| ▶ UCS: | ▶ $f(n) = -n.\text{depth}$ |

Výhoda: prioritní fronta jako univerzální datová struktura pro otevřené uzly (frontier)
(Přesto, zásobník(LIFO) a fronta (FIFO) jsou konceptuálně perfektní datové struktury pro DFS a BFS.)

Nevýhoda: všechny uvedené $f(n)$ odpovídají součtu cen ze startu do n , cost_from_start . Co nám schází?

21 / 33

Notes

- DFS: $f(n) = -n.\text{depth}$
- BFS: $f(n) = n.\text{depth}$
- UCS: $f(n) = n.\text{path_cost}$

Dívají se lidé zpátky, když plánují cestu? Je pohled zpátky vůbec důležitý? Pokud ano, kdy?

Výběr uzlu pomocí $f(n)$. Prohledávaný uzel: $n = (p, s, \text{cost_value})$

Výběr dalšího uzlu k prozkoumání (operace pop):

$$\text{node} \leftarrow \underset{n \in Q}{\operatorname{argmin}} f(n)$$

Jakou funkci $f(n)$ používají algoritmy DFS, BFS a UCS? Spárujte je!

- | | |
|--------|---------------------------------------|
| ▶ DFS: | ▶ $f(n) = n.\text{cost_from_start}$ |
| ▶ BFS: | ▶ $f(n) = n.\text{depth}$ |
| ▶ UCS: | ▶ $f(n) = -n.\text{depth}$ |

Výhoda: prioritní fronta jako univerzální datová struktura pro otevřené uzly (frontier)
(Přesto, zásobník(LIFO) a fronta (FIFO) jsou konceptuálně perfektní datové struktury pro DFS a BFS.)

Nevýhoda: všechny uvedené $f(n)$ odpovídají součtu cen ze startu do n , `cost_from_start` . Co nám schází?

21 / 33

Notes

- DFS: $f(n) = -n.\text{depth}$
- BFS: $f(n) = n.\text{depth}$
- UCS: $f(n) = n.\text{path_cost}$

Dívají se lidé zpátky, když plánují cestu? Je pohled zpátky vůbec důležitý? Pokud ano, kdy?

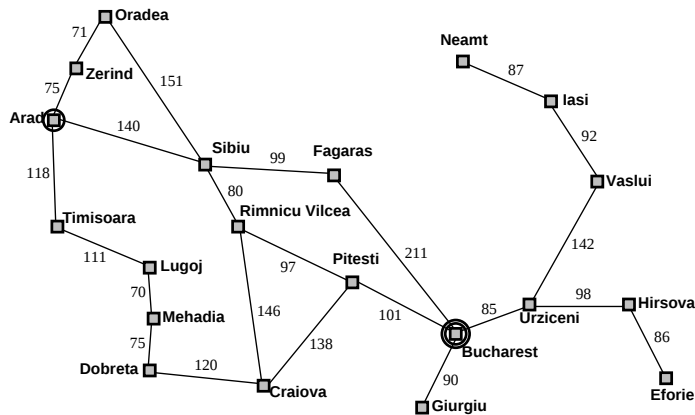
Jak blízko jsme k cíli, **cost-to-go** ? – Heuristika

- ▶ Funkce, která odhaduje, jak blízko je *stav* od nejbližšího cíle.
(„Jakou cenu musíme ještě zaplatit, abychom se dostali do některého cíle?“)
- ▶ Navržena pro konkrétní problém.
- ▶ $h(s)$ – je to funkce stavu (její hodnota se stává atributem uzlu v prohledávacím stromu)
- ▶ Aplikována na *uzel* n , $h(n)$ je heuristická hodnota stavu uloženého v uzlu n .

Notes

Co se stane, když $h(s)$ = skutečná cena nejlevnější cesty do cíle?

Příklad heuristiky



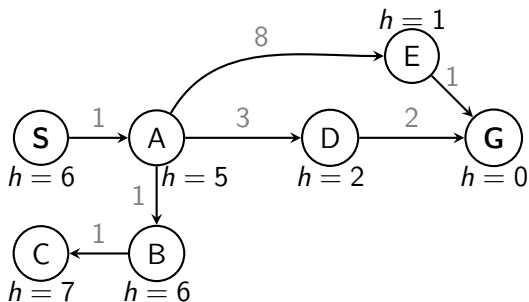
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Notes

Vzdálenost leteckou čarou do Bukurešti.

Ilustrace selhání *greedy* algoritmu: Představte si, že chceme z Iasi do Fagaras. Neamt bude expandován, čímž se znovu otevře Iasi. A protože je blíž k Fagaras než Vaslui, bude expandován znovu. Nekonečná smyčka. . . (3.5.1. v [2])

Hladový alg., vyber $n^* = \operatorname{argmin}_{n \in Q} h(n)$



Co je špatné (a dobré) na hladovém algoritmu?

24 / 33

Notes

Také nazýván "Greedy best-first search" [2].

Co se stane v tomto příkladu?

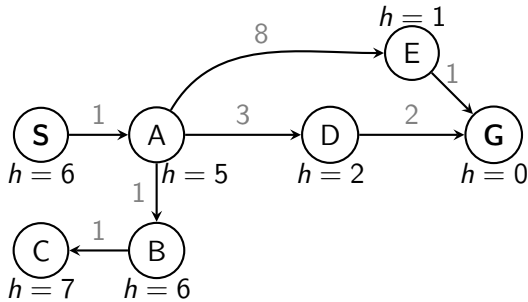
1. Expanduj "S". Přidej "A" do frontie.
2. Expanduj "A". Přidej "B", "D", "E".
3. Expanduj "E" ($h = 1$). Získej "G".

Špatné:

- není optimální
- není úplný (stromová verze; lze ukázat na grafu Rumunská – viz minulý slide)
- grafová verze je úplná pouze na konečném stavovém prostoru

Dobré: je jednoduchý.

Hladový alg., vyber $n^* = \operatorname{argmin}_{n \in Q} h(n)$



Co je špatné (a dobré) na hladovém algoritmu?

24 / 33

Notes

Také nazýván “Greedy best-first search” [2].

Co se stane v tomto příkladu?

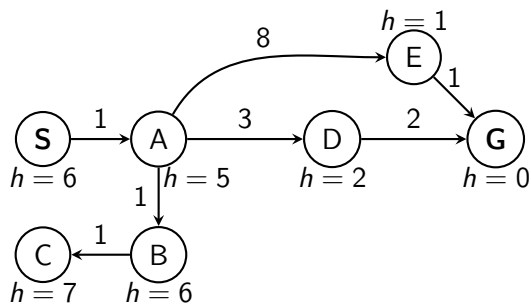
1. Expanduj “S”. Přidej “A” do frontie.
2. Expanduj “A”. Přidej “B”, “D”, “E”.
3. Expanduj “E” ($h = 1$). Získej “G”.

Špatné:

- není optimální
- není úplný (stromová verze; lze ukázat na grafu Rumunska – viz minulý slide)
- grafová verze je úplná pouze na konečném stavovém prostoru

Dobré: je jednoduchý.

A* kombinuje UCS a hladový algoritmus



UCS řadí uzly podle $g(n)$ (cost_from_start)

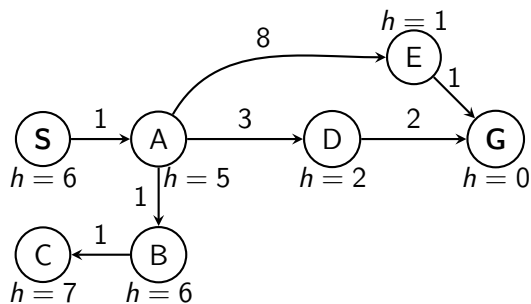
Hladový alg. řadí uzly podle $h(n)$ (heuristika, cost_to_go)

A* řadí uzly podle: $f(n) = g(n) + h(n)$

Notes

Odkrojujte si prohlédávací algoritmus na papíře. Najde nejkratší cestu?

A* kombinuje UCS a hladový algoritmus



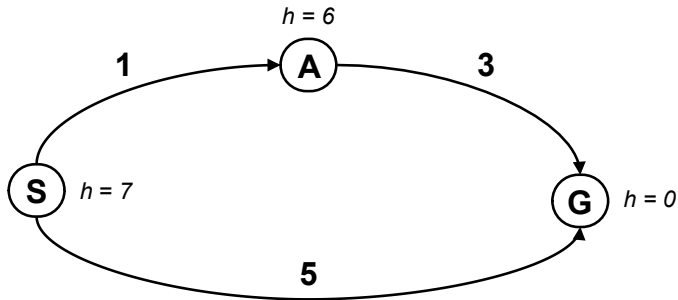
UCS řadí uzly podle $g(n)$ (cost_from_start)

Hladový alg. řadí uzly podle $h(n)$ (heuristika, cost_to_go)

A* řadí uzly podle: $f(n) = g(n) + h(n)$

Odkrojujte si prohlédávací algoritmus na papíře. Najde nejkratší cestu?

Je A* optimální?



2

Co je tu za problém?

²Příklad grafu: Dan Klein a Pieter Abbeel

26 / 33

Notes

Zkuste odpovědět na otázku než budete pokračovat na další slide.

1. S

- $f(S) = g(S) + h(S) = 0 + 7 = 7$
- expandujte uzel a vyškrtněte ho z fronty

2. $S \rightarrow A$

- $f(A) = g(A) + h(A) = 1 + 6 = 7$

3. $S \rightarrow G$

- $f(G) = g(G) + h(G) = 5 + 0 = 5$
- Uzel je nejlevnější ve frontieru. Vyjmeme ho a máme hotovo.

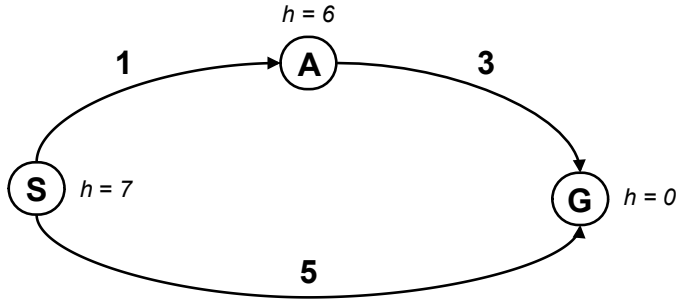
Jejda! To ale není nejlevnější cesta! Co se stalo?

Následující nechte na další slide.

Problém s $h(A) = 6$. Nadhodnocuje skutečnou cenu. (Stejný problém s $h(S)$.)

Odhady musí být $\leq$ skutečné náklady.

Je A^* optimální?



2

Co je tu za problém?

²Příklad grafu: Dan Klein a Pieter Abbeel

26 / 33

Notes

Zkuste odpovědět na otázku než budete pokračovat na další slide.

1. S

- $f(S) = g(S) + h(S) = 0 + 7 = 7$
- expandujte uzel a vyškrtněte ho z fronty

2. $S \rightarrow A$

- $f(A) = g(A) + h(A) = 1 + 6 = 7$

3. $S \rightarrow G$

- $f(G) = g(G) + h(G) = 5 + 0 = 5$
- Uzel je nejlevnější ve frontieru. Vyjmeme ho a máme hotovo.

Jejda! To ale není nejlevnější cesta! Co se stalo?

Následující nechte na další slide.

Problém s $h(A) = 6$. Nadhodnocuje skutečnou cenu. (Stejný problém s $h(S)$.)

Odhady musí být $\leq$ skutečné náklady.

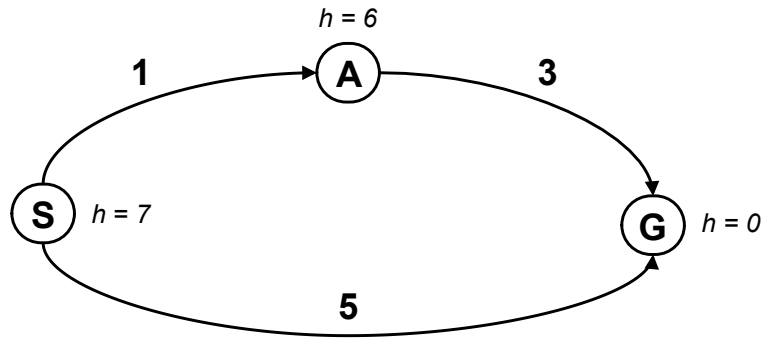
Jaká je správná hodnota $h(A)$?

A: $0 \leq h(A) \leq 4$

B: $h(A) \leq 3$

C: $0 \leq h(A) \leq 3$

D: $0 \leq h(A)$



27 / 33

Notes

$h(A) \leq 3$ znamená, že odhad je menší nebo rovný skutečné ceně nejlevnější cesty z A do cíle. Heuristika nesmí být pesimistická.

B je správně.

Záporné $h(n)$ neporušuje podmínku přípustnosti, ale $h(\text{Cíl}) = 0$ musí být dodrženo.

Diskuse např. zde

<https://stackoverflow.com/questions/30067813/are-heuristic-functions-that-produce-negative-values-inadmissible>

Heuristická funkce h je připustná, pokud:

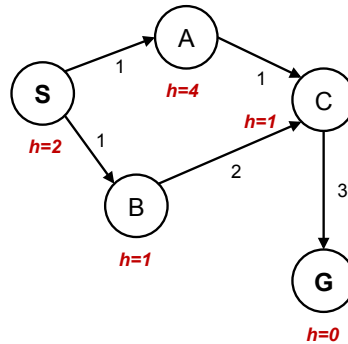
$$\begin{aligned}\forall \text{Goal} : h(n) &\leq \text{cost}(n.\text{state}, \text{Goal}) \\ h(\text{Goal}) &= 0\end{aligned}$$

Notes

Ačkoli připustná heuristika může generovat i záporné hodnoty na cestě k cíli, dává to smysl? Jak byste interpretovali $h(n) = 0$? Je to smysluplné minimum? Proč?

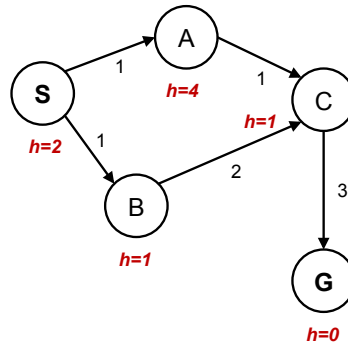
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



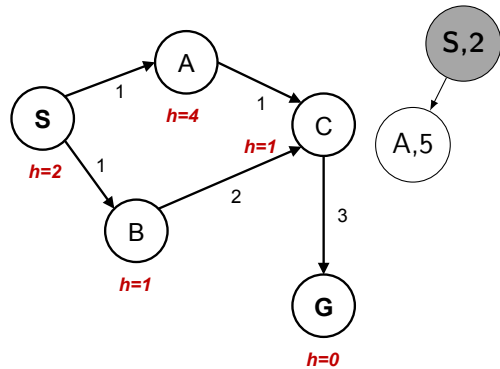
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



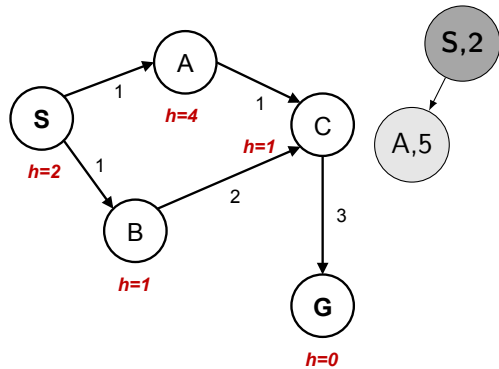
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



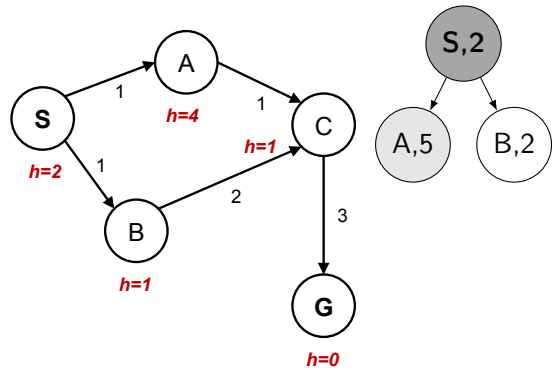
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



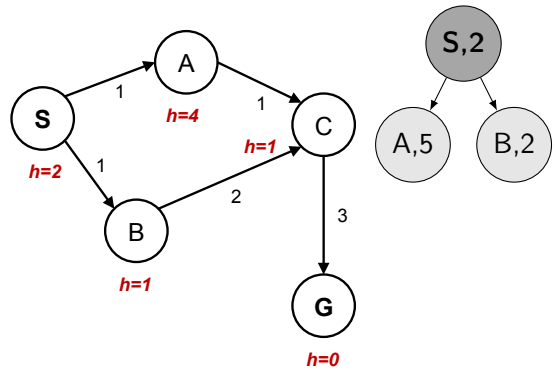
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



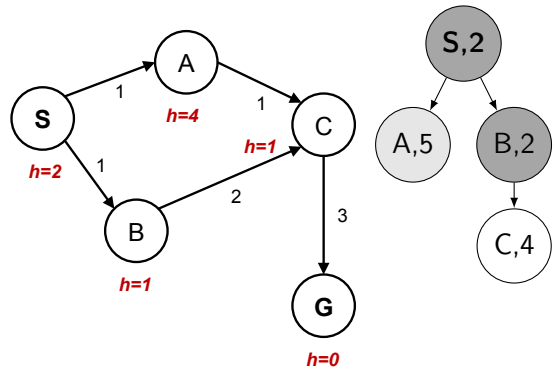
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



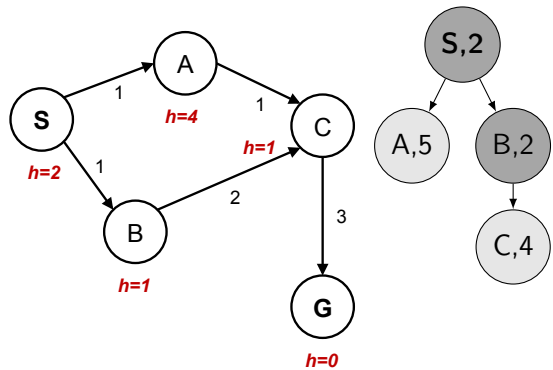
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



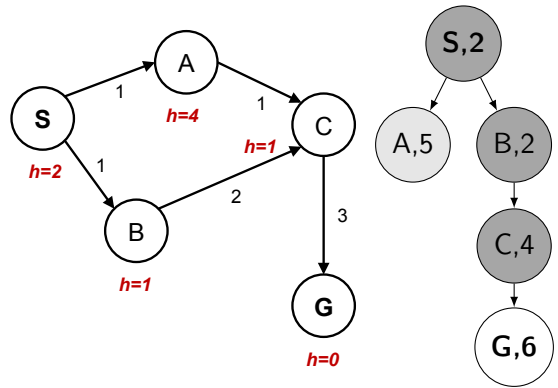
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



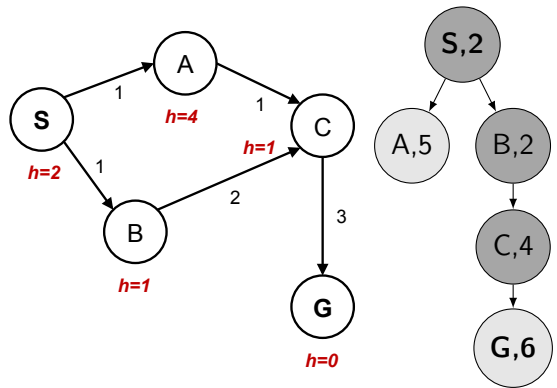
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



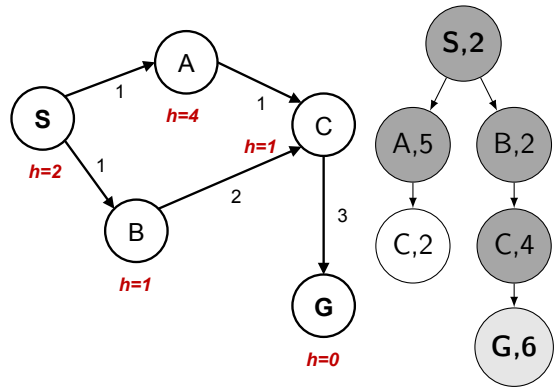
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



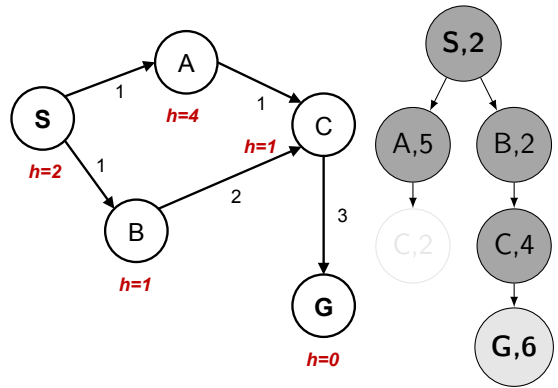
Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



Heuristika: Stačí přípustnost?

```
1: function FORWARD_SEARCH
2:   Q.insert(_, s0, 0, h(s0))
3:   Označ s0 jako navštívený
4:   while Q není prázdná do
5:     p, s, g, _ ← Q.pop()
6:     parent[s] ← p
7:     if s ∈ SG then return Success
8:     for all a ∈ A(s) do
9:       s', c ← result(s, a)
10:      if s' není navštívený then
11:        Označ s' jako navštívený
12:        Q.insert(s, s', g + c, g + c + h(s'))
13:      else
14:        Vyřeš duplikaci s' v Q
return Failure
```



Jaká je správná hodnota $h(A)$?

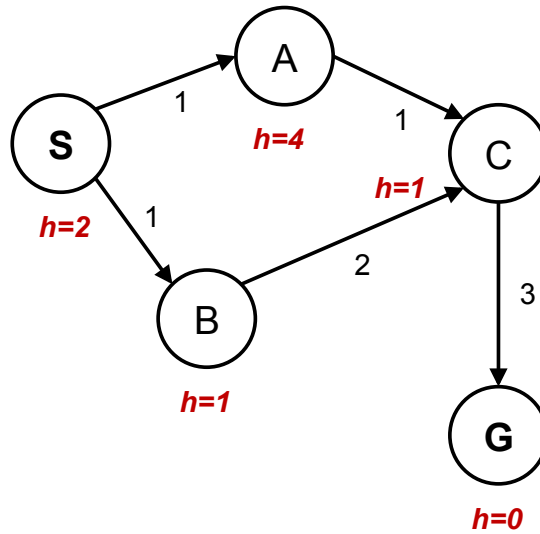
Předpokládejte, že ostatní $h(s)$ se nezmění.

A: $h(A) = 1$

B: $h(A) = 2$

C: $1 \leq h(A) \leq 2$

D: $0 \leq h(A) \leq 1$



30 / 33

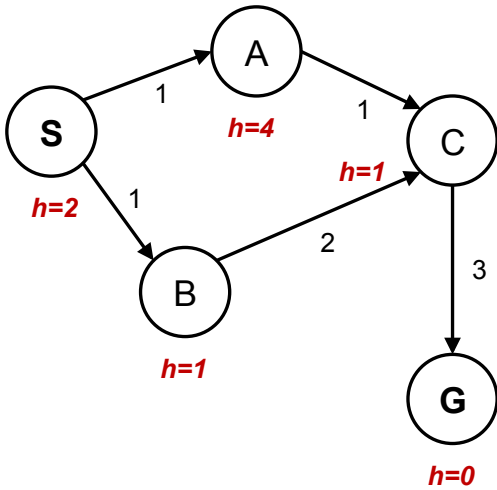
Notes

Jak bude vysvětleno na dalších slidech:

$$h(A) \leq c(A, C) + h(C) = 2$$

$$h(S) \leq c(S, A) + h(A) \text{ it means } h(A) \geq h(S) - c(A, S) = 1$$

Konzistentní heuristika



Připustná h :

$$h(A) \leq \text{skutečná nejnížší cena } A \rightarrow G$$

$$h(\text{Cil}) = 0$$

Konzistentní h :

$$h(\text{Cil}) = 0 \text{ pro všechny cile}$$

$$h(A) - h(C) \leq \text{skutečná nejnížší } A \rightarrow C$$

obecně:

$$h(p) - h(s) \leq \text{skutečná nejnížší cena } p \rightarrow s \text{ pro}$$

každý pár rodiče p a jeho následníka s

$$f(n) = g(n) + h(n) \text{ se podél cesty ze startu do cíle nikdy nesnižuje!}$$

Notes

Naše heuristika byla přípustná.

Pro *stromovou* variantu algoritmu by to stačilo. Alg. by expandoval C a našel alternativní, levnější cestu.

V *grafové* variantě je problémem podgraf $A \rightarrow C \rightarrow G$, kde je porušena podmínka na *konzistenci* heuristiky.

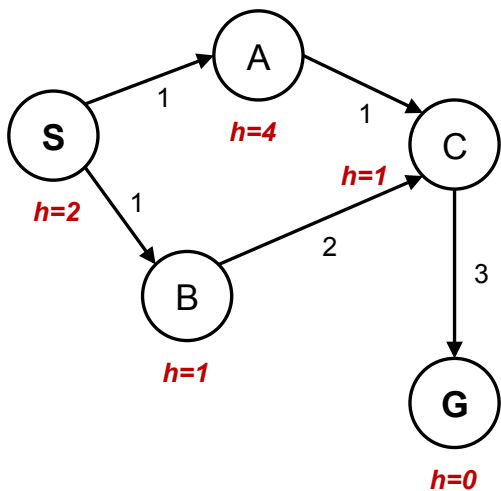
Ze struktury grafu plynou dvě omezení na hodnotu $h(A)$:

$$h(S) - h(A) \leq c(S, A)$$

$$h(A) - h(C) \leq c(A, C)$$

Ano, všechny konzistentní heuristiky jsou přípustné. Mimochodem, není vůbec snadné vymyslet heuristickou funkci, která je přípustná, ale není konzistentní.

Konzistentní heuristika



Připustná h :

$h(A) \leq$ skutečná nejnižší cena $A \rightarrow G$

$h(\text{Cíl}) = 0$

Konzistentní h :

$h(\text{Cíl}) = 0$ pro všechny cíle

$h(A) - h(C) \leq$ skutečná nejnižší $A \rightarrow C$

obecně:

$h(p) - h(s) \leq$ skutečná nejnižší cena $p \rightarrow s$ pro každý pár rodiče p a jeho následníka s

$f(n) = g(n) + h(n)$ se podél cesty ze startu do cíle nikdy nesnižuje!

31 / 33

Notes

Naše heuristika byla přípustná.

Pro *stromovou* variantu algoritmu by to stačilo. Alg. by expandoval C a našel alternativní, levnější cestu.

V grafové variantě je problémem podgraf $A \rightarrow C \rightarrow G$, kde je porušena podmínka na *konzistenci* heuristiky.

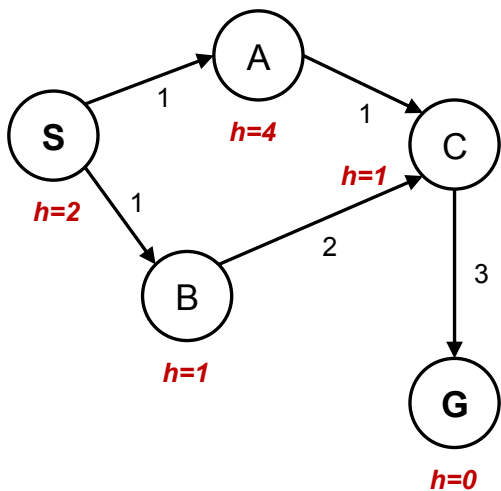
Ze struktury grafu plynou dvě omezení na hodnotu $h(A)$:

$$h(S) - h(A) \leq c(S, A)$$

$$h(A) - h(C) \leq c(A, C)$$

Ano, všechny konzistentní heuristiky jsou přípustné. Mimochodem, není vůbec snadné vymyslet heuristickou funkci, která je přípustná, ale není konzistentní.

Konzistentní heuristika



Připustná h :

$h(A) \leq$ skutečná nejnižší cena $A \rightarrow G$

$h(\text{Cíl}) = 0$

Konzistentní h :

$h(\text{Cíl}) = 0$ pro všechny cíle

$h(A) - h(C) \leq$ skutečná nejnižší $A \rightarrow C$

obecně:

$h(p) - h(s) \leq$ skutečná nejnižší cena $p \rightarrow s$ pro každý pár rodiče p a jeho následníka s

$f(n) = g(n) + h(n)$ se podél cesty ze startu do cíle nikdy nesnižuje!

31 / 33

Notes

Naše heuristika byla připustná.

Pro *stromovou* variantu algoritmu by to stačilo. Alg. by expandoval C a našel alternativní, levnější cestu.

V *grafové* variantě je problémem podgraf $A \rightarrow C \rightarrow G$, kde je porušena podmínka na *konzistenci* heuristiky.

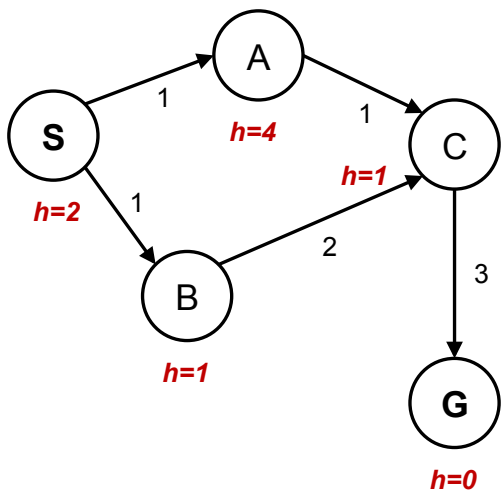
Ze struktury grafu plynou dvě omezení na hodnotu $h(A)$:

$$h(S) - h(A) \leq c(S, A)$$

$$h(A) - h(C) \leq c(A, C)$$

Ano, všechny konzistentní heuristiky jsou připustné. Mimochodem, není vůbec snadné vymyslet heuristickou funkci, která je připustná, ale není konzistentní.

Konzistentní heuristika



Připustná h :

$h(A) \leq$ skutečná nejnižší cena $A \rightarrow G$

$h(\text{Cíl}) = 0$

Konzistentní h :

$h(\text{Cíl}) = 0$ pro všechny cíle

$h(A) - h(C) \leq$ skutečná nejnižší $A \rightarrow C$

obecně:

$h(p) - h(s) \leq$ skutečná nejnižší cena $p \rightarrow s$ pro každý pár rodiče p a jeho následníka s

$f(n) = g(n) + h(n)$ se podél cesty ze startu do cíle nikdy nesnižuje!

31 / 33

Notes

Naše heuristika byla připustná.

Pro *stromovou* variantu algoritmu by to stačilo. Alg. by expandoval C a našel alternativní, levnější cestu.

V grafové variantě je problémem podgraf $A \rightarrow C \rightarrow G$, kde je porušena podmínka na *konzistenci* heuristiky.

Ze struktury grafu plynou dvě omezení na hodnotu $h(A)$:

$$h(S) - h(A) \leq c(S, A)$$

$$h(A) - h(C) \leq c(A, C)$$

Ano, všechny konzistentní heuristiky jsou připustné. Mimochodem, není vůbec snadné vymyslet heuristickou funkci, která je připustná, ale není konzistentní.

- ▶ Graf stavového prostoru vs. strom prohledávání
- ▶ Strategie prohledávání - vlastnosti, složitosti
- ▶ Ohodnocení stavů - `cost_from_start` a `cost_to_go`
- ▶ Efektivita – využití heuristických odhadů hodnot `cost_to_go`
- ▶ Ne všechny heuristiky jsou stejně dobré (přípustnost, konzistence, informovanost)

Odkazy, další čtení

Některé obrázky převzaty z [2]. Kapitola 2 v [1] poskytuje kompaktní/zhuštěný úvod do vyhledávacích algoritmů.

[1] Steven M. LaValle.

Planning Algorithms.

Cambridge, 1st edition, 2006.

Online version available at: <http://planning.cs.uiuc.edu>.

[2] Stuart Russell and Peter Norvig.

Artificial Intelligence: A Modern Approach.

Prentice Hall, 4th edition, 2021.

<http://aima.cs.berkeley.edu/>.