TSP pomocí dynamického programování

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

05.02.2026

Problém obchodního cestujícího

Co je TSP?

TSP = Travelling Salesman Problem

Co je TSP?

TSP = Travelling Salesman Problem

„Problém obchodního cestujícího“

Co je TSP?

TSP = Travelling Salesman Problem

„Problém obchodního cestujícího“

Úloha: Obchodní cestující musí navštívit n měst a vrátit se zpět.

Co je TSP?

TSP = Travelling Salesman Problem

„Problém obchodního cestujícího“

Úloha: Obchodní cestující musí navštívit n měst a vrátit se zpět. Jaká je nejkratší možná trasa?

Co je TSP?

TSP = Travelling Salesman Problem

„Problém obchodního cestujícího“

Úloha: Obchodní cestující musí navštívit n měst a vrátit se zpět. Jaká je nejkratší možná trasa?

Formálně:

- Dán úplný graf K_n s n vrcholy

Co je TSP?

TSP = Travelling Salesman Problem

„Problém obchodního cestujícího“

Úloha: Obchodní cestující musí navštívit n měst a vrátit se zpět. Jaká je nejkratší možná trasa?

Formálně:

- Dán úplný graf K_n s n vrcholy
- Hrany mají ohodnocení (vzdálenosti)

Co je TSP?

TSP = Travelling Salesman Problem

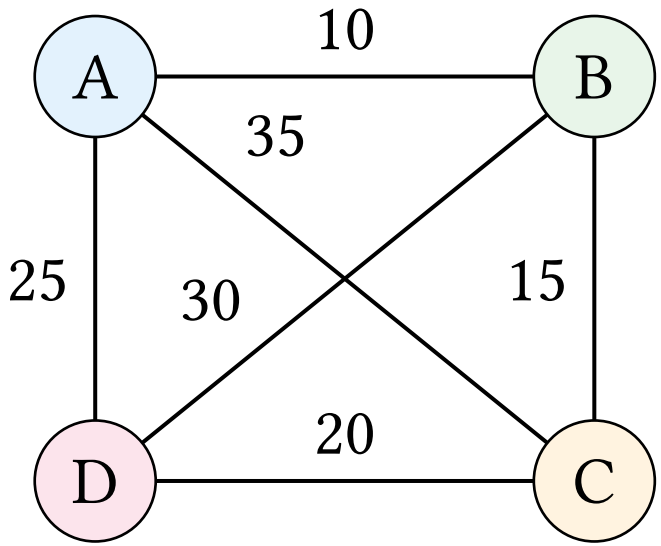
„Problém obchodního cestujícího“

Úloha: Obchodní cestující musí navštívit n měst a vrátit se zpět. Jaká je nejkratší možná trasa?

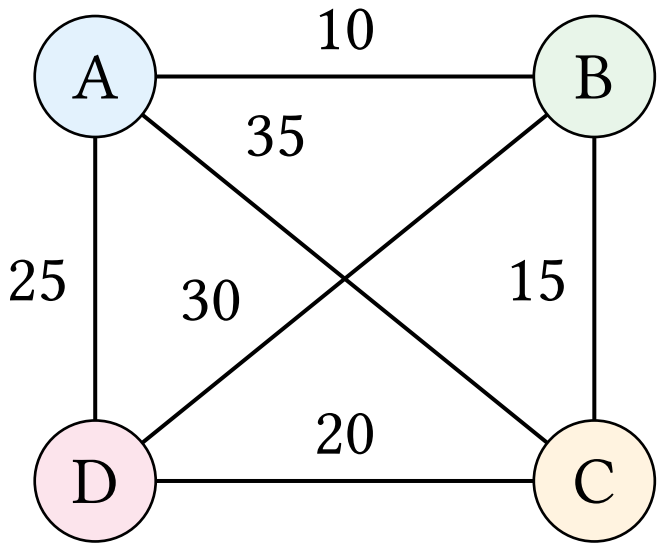
Formálně:

- Dán úplný graf K_n s n vrcholy
- Hrany mají ohodnocení (vzdálenosti)
- Najdi Hamiltonovskou kružnici s minimální délkou

Příklad TSP



Příklad TSP



Cíl: Začít v A, navštívit všechny vrcholy a vrátit se do A s minimální cenou

Proč je TSP důležitý?

Praktické aplikace:

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek
- **Výroba** - optimalizace pohybu robotů

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek
- **Výroba** - optimalizace pohybu robotů
- **Návrh čipů** - minimalizace délky spojů

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek
- **Výroba** - optimalizace pohybu robotů
- **Návrh čipů** - minimalizace délky spojů
- **DNA sekvenování** - skládání fragmentů

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek
- **Výroba** - optimalizace pohybu robotů
- **Návrh čipů** - minimalizace délky spojů
- **DNA sekvenování** - skládání fragmentů
- **Plánování cest** - turistické okruhy

Proč je TSP důležitý?

Praktické aplikace:

- **Logistika** - plánování tras zásilek
- **Výroba** - optimalizace pohybu robotů
- **Návrh čipů** - minimalizace délky spojů
- **DNA sekvenování** - skládání fragmentů
- **Plánování cest** - turistické okruhy

Problém: TSP je **NP-těžký** - neznáme efektivní (polynomiální) algoritmus

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost:

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

n	Počet permutací
5	120

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

n	Počet permutací
5	120
10	3 628 800

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

n	Počet permutací
5	120
10	3 628 800
15	1 307 674 368 000

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

n	Počet permutací
5	120
10	3 628 800
15	1 307 674 368 000
20	2 432 902 008 176 640 000

Naivní přístup - hrubá síla

Idea: Vyzkoušej všechny možné permutace měst

Složitost: $\mathcal{O}(n!)$

n	Počet permutací
5	120
10	3 628 800
15	1 307 674 368 000
20	2 432 902 008 176 640 000

Pro $n = 20$ by výpočet trval **tisíce let**!

Rekapitulace DP

Dynamické programování - rychlé připomenutí

Dynamické programování - rychlé připomenutí

Dynamické programování = ukládáme si výsledky podproblémů

Dynamické programování - rychlé připomenutí

Dynamické programování = ukládáme si výsledky podproblémů

Kdy použít DP:

Dynamické programování - rychlé připomenutí

Dynamické programování = ukládáme si výsledky podproblémů

Kdy použít DP:

1. **Optimální podstruktura** - řešení se skládá z optimálních řešení menších částí

Dynamické programování - rychlé připomenutí

Dynamické programování = ukládáme si výsledky podproblémů

Kdy použít DP:

1. **Optimální podstruktura** - řešení se skládá z optimálních řešení menších částí
2. **Překrývající se podproblémy** - stejné výpočty se opakují

Dynamické programování - rychlé připomenutí

Dynamické programování = ukládáme si výsledky podproblémů

Kdy použít DP:

1. **Optimální podstruktura** - řešení se skládá z optimálních řešení menších částí
2. **Překrývající se podproblémy** - stejné výpočty se opakují

Výsledek: Místo opakovaného počítání použijeme uloženou hodnotu

Proč DP pomůže u TSP?

Klíčové pozorování:

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Záleží pouze na:

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Záleží pouze na:

1. **Která města** jsme již navštívili

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Záleží pouze na:

1. **Která města** jsme již navštívili
2. **Kde** se právě nacházíme

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Záleží pouze na:

1. **Která města** jsme již navštívili
2. **Kde** se právě nacházíme

Podproblém:

Proč DP pomůže u TSP?

Klíčové pozorování:

Když hledáme optimální trasu přes města, nezáleží na **pořadí**, v jakém jsme je navštívili

Záleží pouze na:

1. **Která města** jsme již navštívili
2. **Kde** se právě nacházíme

Podproblém: Jaká je minimální cena dostat se do města j, pokud jsme navštívili množinu měst S?

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Cesty $0 \to 1 \to 2 \to 3$ a $0 \to 2 \to 1 \to 3$

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Cesty $0 \to 1 \to 2 \to 3$ a $0 \to 2 \to 1 \to 3$

obě řeší stejný podproblém:

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Cesty $0 \to 1 \to 2 \to 3$ a $0 \to 2 \to 1 \to 3$

obě řeší stejný podproblém: „Jsme v městě 3, navštívili jsme $\{0, 1, 2, 3\}$“

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Cesty $0 \to 1 \to 2 \to 3$ a $0 \to 2 \to 1 \to 3$

obě řeší stejný podproblém: „Jsme v městě 3, navštívili jsme $\{0, 1, 2, 3\}$“

$\to$ Stačí spočítat jednou a uložit!

Proč DP pomůže u TSP?

Příklad opakujícího se podproblému:

Cesty $0 \to 1 \to 2 \to 3$ a $0 \to 2 \to 1 \to 3$

obě řeší stejný podproblém: „Jsme v městě 3, navštívili jsme $\{0, 1, 2, 3\}$“

$\to$ Stačí spočítat jednou a uložit!

Potřebujeme: Efektivně reprezentovat **množinu navštívených měst**

Bitová reprezentace množin

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]`

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]` - neefektivní pro ukládání stavů

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]` - neefektivní pro ukládání stavů
2. **Množina:** `set<int>`

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]` - neefektivní pro ukládání stavů
2. **Množina:** `set<int>` - pomalé porovnávání

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]` - neefektivní pro ukládání stavů
2. **Množina:** `set<int>` - pomalé porovnávání
3. **Bitová maska:** Celé číslo!

Proč bitová maska?

Problém: Jak reprezentovat podmnožinu n měst?

Možnosti:

1. **Pole:** `bool visited[n]` - neefektivní pro ukládání stavů
2. **Množina:** `set<int>` - pomalé porovnávání
3. **Bitová maska:** Celé číslo! ✓

Proč bitová maska?

Výhoda bitové masky:

- Kompaktní reprezentace

Proč bitová maska?

Výhoda bitové masky:

- Kompaktní reprezentace
- Rychlé operace

Proč bitová maska?

Výhoda bitové masky:

- Kompaktní reprezentace
- Rychlé operace
- Lze použít jako index do pole

Bitmaska jako množina

Klíčová myšlenka:

Bitmaska jako množina

Klíčová myšlenka: Každý **bit** reprezentuje jedno **město**

Bitmaska jako množina

Klíčová myšlenka: Každý **bit** reprezentuje jedno **město**

Bit	3	2	1	0
Město	D	C	B	A

Bitmaska jako množina

Klíčová myšlenka: Každý **bit** reprezentuje jedno **město**

Bit	3	2	1	0
Město	D	C	B	A

- Bit = 1 $\rightarrow$ město je v množině

Bitmaska jako množina

Klíčová myšlenka: Každý **bit** reprezentuje jedno **město**

Bit	3	2	1	0
Město	D	C	B	A

- Bit = 1 $\rightarrow$ město je v množině
- Bit = 0 $\rightarrow$ město není v množině

Bitmaska jako množina

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	{}

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	$\{\}$
1	0001	$\{A\}$

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	$\{\}$
1	0001	$\{A\}$
5	0101	$\{A, C\}$

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	$\{\}$
1	0001	$\{A\}$
5	0101	$\{A, C\}$
7	0111	$\{A, B, C\}$

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	$\{\}$
1	0001	$\{A\}$
5	0101	$\{A, C\}$
7	0111	$\{A, B, C\}$
10	1010	$\{B, D\}$

Bitmaska jako množina

Maska	Binárně	Množina měst
0	0000	$\{\}$
1	0001	$\{A\}$
5	0101	$\{A, C\}$
7	0111	$\{A, B, C\}$
10	1010	$\{B, D\}$
15	1111	$\{A, B, C, D\}$

Bitové operace

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1
OR	|	Bit je 1, pokud alespoň jeden je 1

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1
OR	|	Bit je 1, pokud alespoň jeden je 1
XOR	^	Bit je 1, pokud jsou různé

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1
OR	|	Bit je 1, pokud alespoň jeden je 1
XOR	^	Bit je 1, pokud jsou různé
NOT	~	Negace všech bitů

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1
OR	|	Bit je 1, pokud alespoň jeden je 1
XOR	^	Bit je 1, pokud jsou různé
NOT	~	Negace všech bitů
Posun vlevo	<<	Násobení mocninou 2

Bitové operace

Operace	Operátor	Popis
AND	&	Bit je 1, pokud oba jsou 1
OR	|	Bit je 1, pokud alespoň jeden je 1
XOR	^	Bit je 1, pokud jsou různé
NOT	~	Negace všech bitů
Posun vlevo	<<	Násobení mocninou 2
Posun vpravo	>>	Dělení mocninou 2

Operace AND (&)

AND vrací 1 pouze když **oba** bity jsou 1

Operace AND (&)

AND vrací 1 pouze když **oba** bity jsou 1

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Operace AND (&)

Příklad: `5 & 3`

Operace AND (&)

Příklad: `5 & 3`

```
  0101  (5)
& 0011  (3)
------
  0001  (1)
```

Operace OR (|)

OR vrací 1 když **alespoň jeden** bit je 1

Operace OR (|)

OR vrací 1 když **alespoň jeden** bit je 1

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Operace OR (|)

Příklad: `5 | 3`

Operace OR (|)

Příklad: `5 | 3`

```
  0101  (5)
| 0011  (3)
------
  0111  (7)
```

Posun bitů (<<, >>)

Posun vlevo (<<) - posouvá bity doleva, zprava doplňuje nuly

Posun bitů (<<, >>)

Posun vlevo (<<) - posouvá bity doleva, zprava doplňuje nuly

```
1 << 0  =  1  // 0001 - bit na pozici 0
1 << 1  =  2  // 0010 - bit na pozici 1
1 << 2  =  4  // 0100 - bit na pozici 2
1 << 3  =  8  // 1000 - bit na pozici 3
```

Posun bitů (<<, >>)

Posun vlevo (<<) - posouvá bity doleva, zprava doplňuje nuly

```
1 << 0  =  1  // 0001 - bit na pozici 0
1 << 1  =  2  // 0010 - bit na pozici 1
1 << 2  =  4  // 0100 - bit na pozici 2
1 << 3  =  8  // 1000 - bit na pozici 3
```

Obecně: `1 << i` vytvoří číslo s jedničkou na i-té pozici

Posun bitů (<<, >>)

Posun vlevo (<<) - posouvá bity doleva, zprava doplňuje nuly

```
1 << 0  =  1  // 0001 - bit na pozici 0
1 << 1  =  2  // 0010 - bit na pozici 1
1 << 2  =  4  // 0100 - bit na pozici 2
1 << 3  =  8  // 1000 - bit na pozici 3
```

Obecně: `1 << i` vytvoří číslo s jedničkou na i-té pozici

$$1 \ll i = 2^i$$

Praktické operace s množinami

1. Vytvoření prázdné množiny:

Praktické operace s množinami

1. Vytvoření prázdné množiny:

```
int mask = 0;  // 0000 = {}
```

Praktické operace s množinami

1. Vytvoření prázdné množiny:

```
int mask = 0;  // 0000 = {}
```

2. Přidání města i do množiny:

Praktické operace s množinami

1. Vytvoření prázdné množiny:

```
int mask = 0;  // 0000 = {}
```

2. Přidání města i do množiny:

```
mask = mask | (1 << i);  // Alternatively: mask |= (1 << i)
```

Praktické operace s množinami

1. Vytvoření prázdné množiny:

```
int mask = 0;  // 0000 = {}
```

2. Přidání města i do množiny:

```
mask = mask | (1 << i);  // Alternatively: mask |= (1 << i)
```

Příklad: Přidej město B (index 1) do $\{A\}$ (maska 1)

Praktické operace s množinami

1. Vytvoření prázdné množiny:

```
int mask = 0;  // 0000 = {}
```

2. Přidání města i do množiny:

```
mask = mask | (1 << i);  // Alternatively: mask |= (1 << i)
```

Příklad: Přidej město B (index 1) do $\{A\}$ (maska 1)

```
  0001  (mask = 1, set {A})
| 0010  (1 << 1)
------
  0011  (mask = 3, set {A, B})
```

Praktické operace s množinami

3. Odebrání města i z množiny:

Praktické operace s množinami

3. Odebrání města i z množiny:

```
mask = mask & ~(1 << i);  // Alternatively: mask &= ~(1 << i)
```

Praktické operace s množinami

3. Odebrání města i z množiny:

```
mask = mask & ~(1 << i);  // Alternatively: mask &= ~(1 << i)
```

Příklad: Odeber město B (index 1) z $\{A, B, C\}$ (maska 7)

Praktické operace s množinami

3. Odebrání města i z množiny:

```
mask = mask & ~(1 << i);  // Alternatively: mask &= ~(1 << i)
```

Příklad: Odeber město B (index 1) z $\{A, B, C\}$ (maska 7)

```
  0111  (mask = 7, set {A, B, C})
& 1101  (~(1 << 1) = ~0010 = 1101)
------
  0101  (mask = 5, set {A, C})
```

Praktické operace s množinami

4. Test, zda město i je v množině:

Praktické operace s množinami

4. Test, zda město i je v množině:

```
if (mask & (1 << i)) {
    // city i is in the set
}
```

Praktické operace s množinami

4. Test, zda město i je v množině:

```
if (mask & (1 << i)) {
    // city i is in the set
}
```

Příklad: Je město C (index 2) v $\{A, C\}$ (maska 5)?

Praktické operace s množinami

4. Test, zda město i je v množině:

```
if (mask & (1 << i)) {
    // city i is in the set
}
```

Příklad: Je město C (index 2) v $\{A, C\}$ (maska 5)?

```
  0101  (mask = 5, set {A, C})
& 0100  (1 << 2)
------
  0100  (non-zero => YES)
```

Praktické operace s množinami

5. Počet všech podmnožin:

Praktické operace s množinami

5. Počet všech podmnožin:

Pro n měst existuje 2^n možných podmnožin

Praktické operace s množinami

5. Počet všech podmnožin:

Pro n měst existuje 2^n možných podmnožin

```
int all_subsets = 1 << n;  // 2^n

// Iterate over all subsets:
for (int mask = 0; mask < (1 << n); mask++) {
    // mask represents one subset
}
```

Praktické operace s množinami

5. Počet všech podmnožin:

Pro n měst existuje 2^n možných podmnožin

```
int all_subsets = 1 << n;  // 2^n

// Iterate over all subsets:
for (int mask = 0; mask < (1 << n); mask++) {
    // mask represents one subset
}
```

Příklad: Pro $n = 3$ máme $2^3 = 8$ podmnožin (0 až 7)

Praktické operace s množinami

6. Množina všech měst:

Praktické operace s množinami

6. Množina všech měst:

```
int all_cities = (1 << n) - 1;
```

Praktické operace s množinami

6. Množina všech měst:

```
int all_cities = (1 << n) - 1;
```

Proč to funguje?

Praktické operace s množinami

6. Množina všech měst:

```
int all_cities = (1 << n) - 1;
```

Proč to funguje?

```
1 << 4      = 10000  (16)
(1 << 4) - 1 = 01111  (15 = {A, B, C, D})
```

Praktické operace s množinami

6. Množina všech měst:

```
int all_cities = (1 << n) - 1;
```

Proč to funguje?

```
1 << 4      = 10000  (16)
(1 << 4) - 1 = 01111  (15 = {A, B, C, D})
```

Všechny bity od 0 do $n - 1$ jsou nastaveny na 1

TSP s bitmaskou - Held-Karp algoritmus

top-down přístup

Definice podproblému

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme
- mask = množina již navštívených měst (bitmaska)

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme
- mask = množina již navštívených měst (bitmaska)

Definice: dp[curr][mask] = minimální cena cesty, která:

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme
- mask = množina již navštívených měst (bitmaska)

Definice: dp[curr][mask] = minimální cena cesty, která:

1. Začíná v městě curr

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme
- mask = množina již navštívených měst (bitmaska)

Definice: dp[curr][mask] = minimální cena cesty, která:

1. Začíná v městě curr
2. Navštíví všechna města, která ještě **nejsou** v mask

Definice podproblému

Stav: (curr, mask) kde:

- curr = aktuální město, kde se nacházíme
- mask = množina již navštívených měst (bitmaska)

Definice: dp[curr][mask] = minimální cena cesty, která:

1. Začíná v městě curr
2. Navštíví všechna města, která ještě **nejsou** v mask
3. Vrátí se do města 0

Definice podproblému

Rekurzivní myšlenka:

Definice podproblému

Rekurzivní myšlenka:

Jsme v městě curr, navštívili jsme města v mask.

Definice podproblému

Rekurzivní myšlenka:

Jsme v městě curr, navštívili jsme města v mask.

Co teď?

Definice podproblému

Rekurzivní myšlenka:

Jsme v městě curr, navštívili jsme města v mask.

Co teď? Zkusíme jít do každého nenavštíveného města i a rekurzivně vyřešíme zbytek.

Definice podproblému

Rekurzivní myšlenka:

Jsme v městě curr, navštívili jsme města v mask.

Co teď? Zkusíme jít do každého nenavštíveného města i a rekurzivně vyřešíme zbytek.

$$\mathrm{dp}[\mathrm{curr}][\mathrm{mask}] = \min_{i \notin \mathrm{mask}} (\mathrm{dist}[\mathrm{curr}][i] + \mathrm{dp}[i][\mathrm{mask} \cup \{i\}])$$

Ukončovací podmínka

Kdy končíme?

Ukončovací podmínka

Kdy končíme?

Když jsme navštívili **všechna** města: mask $=$

Ukončovací podmínka

Kdy končíme?

Když jsme navštívili **všechna** města: mask $= 2^n - 1$

Ukončovací podmínka

Kdy končíme?

Když jsme navštívili **všechna** města: mask $= 2^n - 1$

$$\text{dp}[\text{curr}][\text{all}] = \text{dist}[\text{curr}][0]$$

Ukončovací podmínka

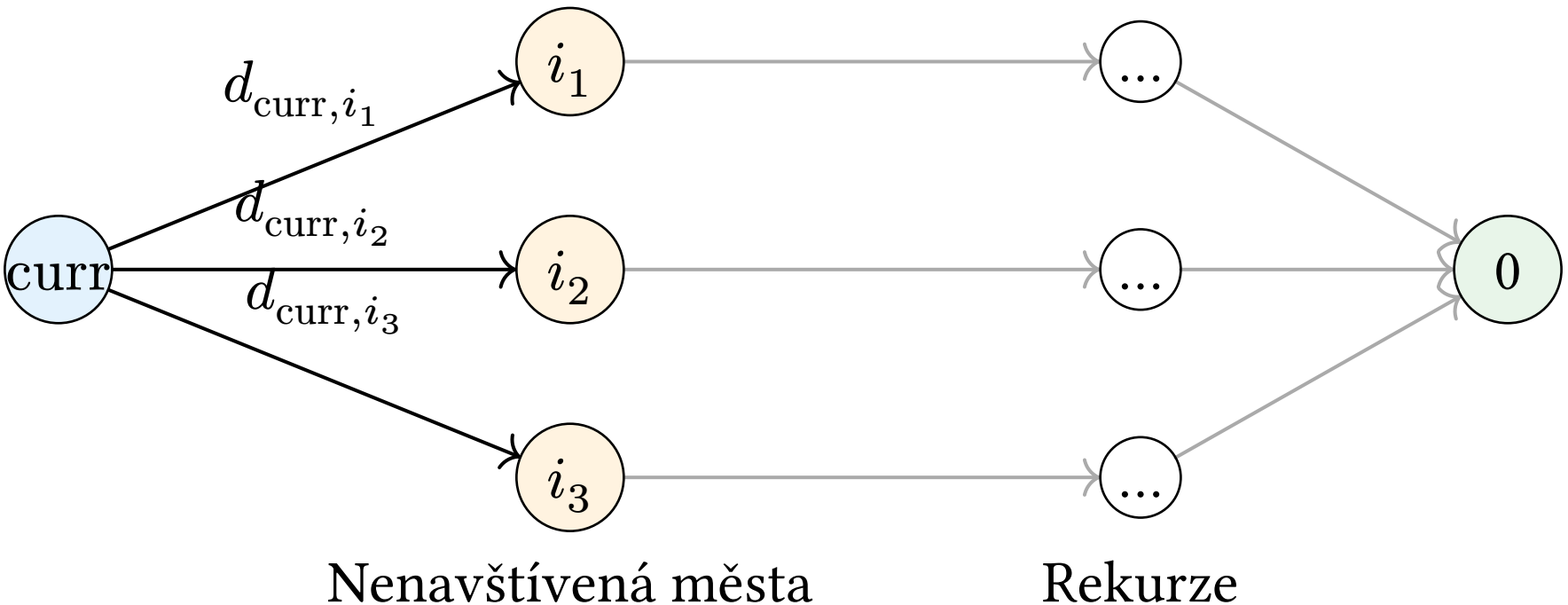
Kdy končíme?

Když jsme navštívili **všechna** města: mask $= 2^n - 1$

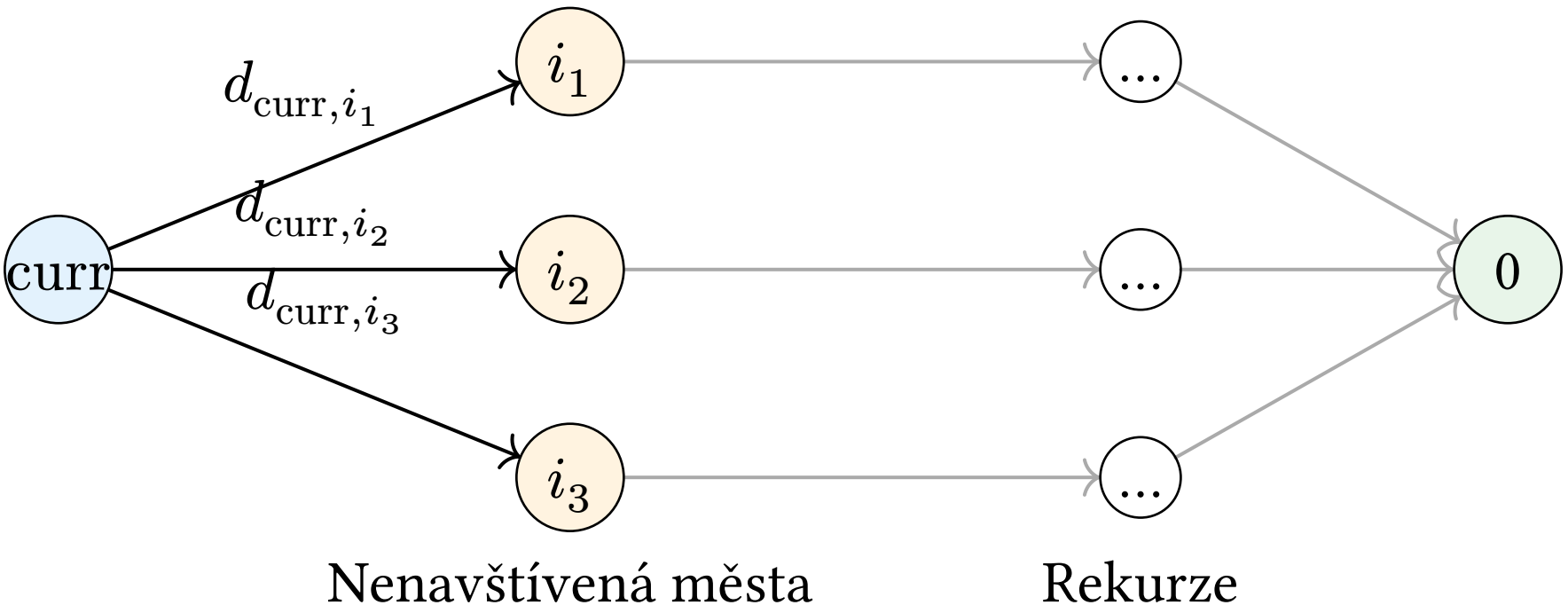
$$\text{dp}[\text{curr}][\text{all}] = \text{dist}[\text{curr}][0]$$

Slovně: Pokud jsme navštívili všechna města, zbývá jen vrátit se do města 0

Ukončovací podmínka



Ukončovací podmínka



Vybíráme nejlepší další město i a rekurzivně řešíme zbytek

Volání

Jak zavoláme algoritmus?

Volání

Jak zavoláme algoritmus?

$$\text{Odpověď} = \text{dp}[0][\{0\}]$$

Volání

Jak zavoláme algoritmus?

$$\text{Odpověď} = \text{dp}[0][\{0\}]$$

Slovně: Začínáme v městě 0, navštívili jsme zatím jen město 0

Volání

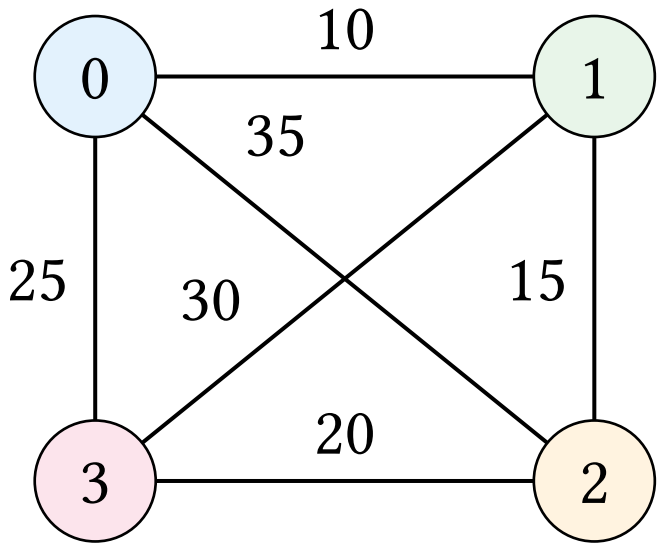
Jak zavoláme algoritmus?

$$\text{Odpověď} = \text{dp}[0][\{0\}]$$

Slovně: Začínáme v městě 0, navštívili jsme zatím jen město 0

Rekurze postupně navštíví všechna města a vrátí se zpět

Příklad - 4 města



Příklad - 4 města

Matice vzdáleností:

	0	1	2	3
0	0	10	35	25
1	10	0	15	30
2	35	15	0	20
3	25	30	20	0

Příklad - 4 města

Rekurzivní volání: dp[0][0001] (začínáme v 0, navštívili jsme {0})

Příklad - 4 města

Rekurzivní volání: dp[0][0001] (začínáme v 0, navštívili jsme {0})

Zkoušíme jít do nenavštívených měst:

Příklad - 4 města

Rekurzivní volání: $\mathrm{dp}[0][0001]$ (začínáme v 0, navštívili jsme $\{0\}$)

Zkoušíme jít do nenavštívených měst:

$$\mathrm{dp}[0][0001] = \min \begin{cases} \mathrm{dist}[0][1] + \mathrm{dp}[1][0011] \\ \mathrm{dist}[0][2] + \mathrm{dp}[2][0101] \\ \mathrm{dist}[0][3] + \mathrm{dp}[3][1001] \end{cases}$$

Příklad - 4 města

Rekurzivní volání: $\mathrm{dp}[0][0001]$ (začínáme v 0, navštívili jsme $\{0\}$)

Zkoušíme jít do nenavštívených měst:

$$\mathrm{dp}[0][0001] = \min \begin{cases} \mathrm{dist}[0][1] + \mathrm{dp}[1][0011] \\ \mathrm{dist}[0][2] + \mathrm{dp}[2][0101] \\ \mathrm{dist}[0][3] + \mathrm{dp}[3][1001] \end{cases}$$

Každé $\mathrm{dp}[\ldots][\ldots]$ se dále rekurzivně rozvíjí…

Příklad - 4 města

Příklad větve: dp[1][0011] (jsme v 1, navštívili jsme $\{0, 1\}$)

Příklad - 4 města

Příklad větve: $\mathrm{dp}[1][0011]$ (jsme v 1, navštívili jsme $\{0, 1\}$)

$$\mathrm{dp}[1][0011] = \min \begin{cases} \mathrm{dist}[1][2] + \mathrm{dp}[2][0111] \\ \mathrm{dist}[1][3] + \mathrm{dp}[3][1011] \end{cases}$$

Příklad - 4 města

Příklad větve: $\mathrm{dp}[1][0011]$ (jsme v 1, navštívili jsme $\{0, 1\}$)

$$\mathrm{dp}[1][0011] = \min \begin{cases} \mathrm{dist}[1][2] + \mathrm{dp}[2][0111] \\ \mathrm{dist}[1][3] + \mathrm{dp}[3][1011] \end{cases}$$

$$= \min \begin{cases} 15 + \mathrm{dp}[2][0111] \\ 30 + \mathrm{dp}[3][1011] \end{cases}$$

Příklad - 4 města

Další úroveň: dp[2][0111] (jsme v 2, navštívili jsme $\{0, 1, 2\}$)

Příklad - 4 města

Další úroveň: $\mathrm{dp}[2][0111]$ (jsme v 2, navštívili jsme $\{0, 1, 2\}$)

$\mathrm{dp}[2][0111] = \min\{\mathrm{dist}[2][3] + \mathrm{dp}[3][1111]$

Příklad - 4 města

Další úroveň: $\mathrm{dp}[2][0111]$ (jsme v 2, navštívili jsme $\{0, 1, 2\}$)

$$\begin{aligned}\mathrm{dp}[2][0111] &= \min\{\mathrm{dist}[2][3] + \mathrm{dp}[3][1111] \\ &= 20 + \mathrm{dp}[3][1111]\end{aligned}$$

Příklad - 4 města

Další úroveň: $\mathrm{dp}[2][0111]$ (jsme v 2, navštívili jsme $\{0, 1, 2\}$)

$$\begin{aligned}\mathrm{dp}[2][0111] &= \min\{\mathrm{dist}[2][3] + \mathrm{dp}[3][1111] \\ &= 20 + \mathrm{dp}[3][1111]\end{aligned}$$

Ukončovací podmínka: $\mathrm{dp}[3][1111] = \mathrm{dist}[3][0] = 25$

Příklad - 4 města

Další úroveň: $\mathrm{dp}[2][0111]$ (jsme v 2, navštívili jsme $\{0, 1, 2\}$)

$$\mathrm{dp}[2][0111] = \min\{\mathrm{dist}[2][3] + \mathrm{dp}[3][1111]$$
$$= 20 + \mathrm{dp}[3][1111]$$

Ukončovací podmínka: $\mathrm{dp}[3][1111] = \mathrm{dist}[3][0] = 25$

Tedy: $\mathrm{dp}[2][0111] = 20 + 25 = 45$

Příklad - 4 města

Zpětné skládání:

Příklad - 4 města

Zpětné skládání:

$\mathrm{dp}[2][0111] = 45$

Příklad - 4 města

Zpětné skládání:

$$\mathrm{dp}[2][0111] = 45$$

$$\mathrm{dp}[1][0011] = \min(15 + 45, 30 + \ldots) = \min(60, \ldots) = 60$$

Příklad - 4 města

Zpětné skládání:

$\mathrm{dp}[2][0111] = 45$

$\mathrm{dp}[1][0011] = \min(15 + 45, 30 + \ldots) = \min(60, \ldots) = 60$

(po vyhodnocení všech větví)

Příklad - 4 města

Zpětné skládání:

$\mathrm{dp}[2][0111] = 45$

$\mathrm{dp}[1][0011] = \min(15 + 45, 30 + \ldots) = \min(60, \ldots) = 60$

(po vyhodnocení všech větví)

Optimální: $\mathrm{dp}[0][0001] = 10 + 60 = 70$

Příklad - 4 města

Zpětné skládání:

$\mathrm{dp}[2][0111] = 45$

$\mathrm{dp}[1][0011] = \min(15 + 45, 30 + \ldots) = \min(60, \ldots) = 60$

(po vyhodnocení všech větví)

Optimální: $\mathrm{dp}[0][0001] = 10 + 60 = 70$

Optimální trasa: $0 \to 1 \to 2 \to 3 \to 0$ s cenou **70**

Implementace v C

Struktura řešení

```c
#include <stdio.h>
#include <limits.h>

#define N 4  // Number of cities
#define INF INT_MAX

int dist[N][N] = {
    {0, 10, 35, 25},
    {10, 0, 15, 30},
    {35, 15, 0, 20},
    {25, 30, 20, 0}
};
int dp[N][1 << N];  // DP table: N cities × 2^N states
```

Rekurzivní funkce

```
int solve(int curr, int mask) {
    // Base case: all cities visited
    if (mask == (1 << N) - 1) {
        return dist[curr][0];  // Return to start
    }

    // Return memoized result
    if (dp[curr][mask] != -1) {
        return dp[curr][mask];
    }
```

Rekurzivní funkce

```
int result = INF;
// Try visiting every unvisited city
for (int i = 0; i < N; i++) {
    // Skip if city i is already visited
    if (mask & (1 << i)) continue;
    // Visit city i
    int newMask = mask | (1 << i);
    int cost = dist[curr][i] + solve(i, newMask);

    if (cost < result) {
        result = cost;
    }
}
```

Rekurzivní funkce

```
    // Memoize and return
    dp[curr][mask] = result;
    return result;
}
```

Hlavní funkce

```
int tsp() {
    // Initialize DP table to -1 (not computed)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < (1 << N); j++)
            dp[i][j] = -1;

    // Start from city 0, with city 0 visited
    int mask = 1;  // 0001 = {city 0}
    return solve(0, mask);
}
```

Rekonstrukce cesty

Často chceme znát nejen **délku**, ale i **samotnou trasu**

Rekonstrukce cesty

Často chceme znát nejen **délku**, ale i **samotnou trasu**

Metoda: Zpětné sledování z memoizované tabulky

Rekonstrukce cesty

Často chceme znát nejen **délku**, ale i **samotnou trasu**

Metoda: Zpětné sledování z memoizované tabulky

```
void printPath() {
    int mask = 1;  // Start: only city 0 visited
    int curr = 0;

    printf("Path: 0");
```

Rekonstrukce cesty

```
    while (mask != (1 << N) - 1) {
        for (int i = 0; i < N; i++) {
            if (mask & (1 << i)) continue;  // Already visited
            int newMask = mask | (1 << i);
            int expected = dist[curr][i] + dp[i][newMask];
            if (expected == dp[curr][mask]) {
                printf(" -> %d", i); curr = i; mask = newMask;
                break;
            }
        }
    }
    printf(" -> 0\n");
}
```

Analýza složitosti

Časová složitost

Počet stavů: $2^n \times n$

Časová složitost

Počet stavů: $2^n \times n$

- 2^n možných podmnožin (masek)

Časová složitost

Počet stavů: $2^n \times n$

- 2^n možných podmnožin (masek)
- n možných koncových měst

Časová složitost

Počet stavů: $2^n \times n$

- 2^n možných podmnožin (masek)
- n možných koncových měst

Přechody: Pro každý stav zkoušíme $\mathcal{O}(n)$ přechodů

Časová složitost

Počet stavů: $2^n \times n$

- 2^n možných podmnožin (masek)
- n možných koncových měst

Přechody: Pro každý stav zkoušíme $\mathcal{O}(n)$ přechodů

Celková složitost: $\mathcal{O}(2^n \times n^2)$

Časová složitost

Porovnání s hrubou silou:

Časová složitost

Porovnání s hrubou silou:

n	Hrubá síla $\mathcal{O}(n!)$	DP $\mathcal{O}(2^n n^2)$
10	3 628 800	102 400

Časová složitost

Porovnání s hrubou silou:

n	Hrubá síla $\mathcal{O}(n!)$	DP $\mathcal{O}(2^n n^2)$
10	3 628 800	102 400
15	1.3×10^{12}	7 372 800

Časová složitost

Porovnání s hrubou silou:

n	Hrubá síla $\mathcal{O}(n!)$	DP $\mathcal{O}(2^n n^2)$
10	3 628 800	102 400
15	1.3×10^{12}	7 372 800
20	2.4×10^{18}	419 430 400

Časová složitost

Porovnání s hrubou silou:

n	Hrubá síla $\mathcal{O}(n!)$	DP $\mathcal{O}(2^n n^2)$
10	3 628 800	102 400
15	1.3×10^{12}	7 372 800
20	2.4×10^{18}	419 430 400
25	1.5×10^{25}	2.1×10^{10}

Časová složitost

Porovnání s hrubou silou:

n	Hrubá síla $\mathcal{O}(n!)$	DP $\mathcal{O}(2^n n^2)$
10	3 628 800	102 400
15	1.3×10^{12}	7 372 800
20	2.4×10^{18}	419 430 400
25	1.5×10^{25}	2.1×10^{10}

Pro $n = 20$: z **tisíců let** na **sekundy**!

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Pro $n = 25$: $2^{25} \times 25 \times 4$ bytes ≈ 3.2 GB

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Pro $n = 25$: $2^{25} \times 25 \times 4$ bytes ≈ 3.2 GB

Optimalizace:

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Pro $n = 25$: $2^{25} \times 25 \times 4$ bytes ≈ 3.2 GB

Optimalizace:

- Iterovat přes masky podle počtu bitů

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Pro $n = 25$: $2^{25} \times 25 \times 4$ bytes ≈ 3.2 GB

Optimalizace:

- Iterovat přes masky podle počtu bitů
- Ukládat jen předchozí “vrstvu”

Prostorová složitost

Paměťové nároky: $\mathcal{O}(2^n \times n)$

Pro $n = 20$: $2^{20} \times 20 \times 4$ bytes ≈ 80 MB

Pro $n = 25$: $2^{25} \times 25 \times 4$ bytes ≈ 3.2 GB

Optimalizace:

- Iterovat přes masky podle počtu bitů
- Ukládat jen předchozí “vrstvu”
- Komprese stavů

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů
- Paměť i čas se stávají limitujícími

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů
- Paměť i čas se stávají limitujícími

Pro větší instance:

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů
- Paměť i čas se stávají limitujícími

Pro větší instance:

- Aproximační algoritmy

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů
- Paměť i čas se stávají limitujícími

Pro větší instance:

- Aproximační algoritmy
- Heuristiky (genetické algoritmy, simulované žíhání)

Limity algoritmu

Praktické omezení: $n \approx 20 - 25$ měst

- Exponenciální růst 2^n
- Pro $n = 30$: $2^{30} \approx 10^9$ stavů
- Paměť i čas se stávají limitujícími

Pro větší instance:

- Aproximační algoritmy
- Heuristiky (genetické algoritmy, simulované žíhání)
- Speciální případy (metrický TSP, euklidovský TSP)

Shrnutí

Co jsme se naučili

1. **TSP** je NP-těžký problém optimalizace tras

Co jsme se naučili

1. **TSP** je NP-těžký problém optimalizace tras
2. **Dynamické programování** = efektivnější řešení než hrubá síla

Co jsme se naučili

1. **TSP** je NP-těžký problém optimalizace tras
2. **Dynamické programování** = efektivnější řešení než hrubá síla
3. **Bitová maska** je elegantní způsob reprezentace množin
   - Kompaktní: jedno číslo = jedna množina
   - Rychlé operace: přidání, odebrání, test příslušnosti

Co jsme se naučili

1. **TSP** je NP-těžký problém optimalizace tras
2. **Dynamické programování** = efektivnější řešení než hrubá síla
3. **Bitová maska** je elegantní způsob reprezentace množin
   - Kompaktní: jedno číslo = jedna množina
   - Rychlé operace: přidání, odebrání, test příslušnosti
4. **Held-Karp algoritmus** má složitost $\mathcal{O}(2^n n^2)$
   - Významné zlepšení oproti $\mathcal{O}(n!)$
   - Prakticky použitelný pro $n \leq 25$

Klíčové bitové operace

Operace	Kód v C
Prázdná množina	`0`
Všechna města	`(1 << n) - 1`
Přidej město i	`mask | (1 << i)`
Odeber město i	`mask & ~(1 << i)`
Je město i v množině?	`mask & (1 << i)`
Počet podmnožin	`1 << n`

Další kroky

Procvičení:

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C
- Přidejte rekonstrukci optimální cesty

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C
- Přidejte rekonstrukci optimální cesty

Rozšíření:

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C
- Přidejte rekonstrukci optimální cesty

Rozšíření:

- Asymetrický TSP (různé vzdálenosti tam a zpět)

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C
- Přidejte rekonstrukci optimální cesty

Rozšíření:

- Asymetrický TSP (různé vzdálenosti tam a zpět)
- TSP s časovými okny

Další kroky

Procvičení:

- Implementujte TSP pro 5-6 měst ručně
- Napište kompletní program v C
- Přidejte rekonstrukci optimální cesty

Rozšíření:

- Asymetrický TSP (různé vzdálenosti tam a zpět)
- TSP s časovými okny
- Aproximační algoritmy pro velké instance