

Paralelní maticové operace

B4B36PDV – Paralelní a distribuované výpočty

B4B36PDV – Paralelní a distribuované výpočty
FEL ČVUT

- Paralelní maticové operace
- Semestrální úloha

Paralelní maticové operace

Skalární součin vektorů u a v délky n lze spočítat jako

$$u \times v = \sum_{i=0}^{n-1} u[i] \cdot v[i]$$

Co když je mnoho prvků v obou vektorech nulových?

Skalární součin vektorů u a v délky n lze spočítat jako

$$u \times v = \sum_{i=0}^{n-1} u[i] \cdot v[i]$$

Co když je mnoho prvků v obou vektorech nulových?

⚠ Potom je neefektivní jak samotné násobení, tak i vektorová reprezentace.

Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí
 $v = \{(4, 3), (6, 1), (10, 2)\}$

Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí
 $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?

●
(1, 1) (2, 5) (6, 2) (8, 2)
(2, 3) (4, 3) (6, 1) (7, 2)
●

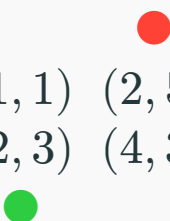
Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí
 $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?



(1, 1)	(2, 5)	(6, 2)	(8, 2)
(2, 3)	(4, 3)	(6, 1)	(7, 2)

Vektory s málo nenulovými hodnotami lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?

(1, 1) (2, 5) (6, 2) (8, 2)
(2, 3) (4, 3) (6, 1) (7, 2)

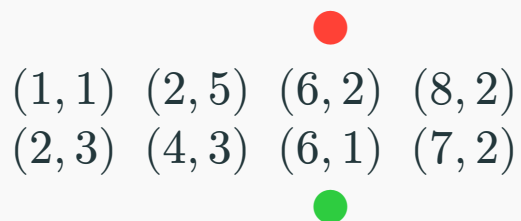
Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí
 $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?



(1, 1) (2, 5) (6, 2) (8, 2)
(2, 3) (4, 3) (6, 1) (7, 2)

Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?

(1, 1)	(2, 5)	(6, 2)	(8, 2)
(2, 3)	(4, 3)	(6, 1)	(7, 2)

Vektory s **málo nenulovými hodnotami** lze reprezentovat kompaktněji!

Místo toho, abychom si pamatovali všechny (i nulové) prvky, tak si pamatujeme pouze:

- na jakých indexech jsou nenulové prvky
- jaké hodnoty jsou na těchto indexech

Např. vektor $v = (0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 2)$ reprezentujeme pomocí
 $v = \{(4, 3), (6, 1), (10, 2)\}$

Jak spočteme skalární součin vektorů u a v ?

(1, 1)	(2, 5)	(6, 2)	(8, 2)
(2, 3)	(4, 3)	(6, 1)	(7, 2)

Součin matice A o rozměrech $m \times n$ a vektoru u délky n lze spočítat jako

$$A_u = (A_1 \times u, A_2 \times u, \dots, A_m \times u)$$

kde $A_i, i \in [m]$ jsou jednotlivé řádky matice A .

Součin matice A o rozměrech $m \times n$ a vektoru u délky n lze spočítat jako

$$A_u = (A_1 \times u, A_2 \times u, \dots, A_m \times u)$$

kde $A_i, i \in [m]$ jsou jednotlivé řádky matice A .

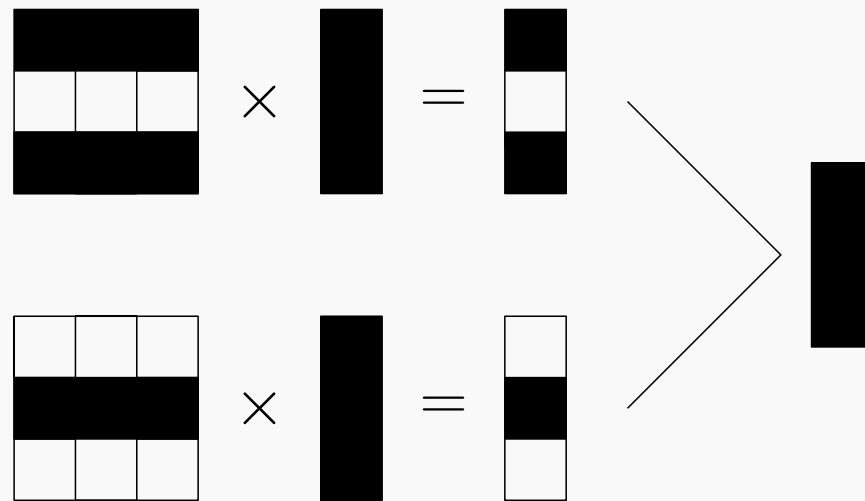
Jak tento výpočet zparalelizovat?

Součin matice A o rozměrech $m \times n$ a vektoru u délky n lze spočítat jako

$$A_u = (A_1 \times u, A_2 \times u, \dots, A_m \times u)$$

kde $A_i, i \in [m]$ jsou jednotlivé řádky matice A .

Jak tento výpočet zparalelizovat?



Je nutné slévat částečné výsledky jednotlivých vláken. Jak na to?

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Deklarujeme si vlastní redukci!

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Deklarujeme si vlastní redukci!

```
#pragma omp declare reduction(name:type:expression) \  
    initializer(expression)
```

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Deklarujeme si vlastní redukci!

```
#pragma omp declare reduction(name:type:expression) \  
    initializer(expression)
```

- name = název vlastní redukce
- type = typ, nad kterým je redukce definována (např. std::vector)
- expression = funkce, která se má vykonávat nad dvěma částečnými výsledky

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Deklarujeme si vlastní redukci!

```
#pragma omp declare reduction(name:type:expression) \  
    initializer(expression)
```

- name = název vlastní redukce
- type = typ, nad kterým je redukce definována (např. std::vector)
- expression = funkce, která se má vykonávat nad dvěma částečnými výsledky

→ Částečné výsledky jsou uloženy v proměnných `omp_in` a `omp_out`

→ Výsledek redukce uložíme zpět do proměnné `omp_out`

Standardní redukce jsou definované na jednoduchých proměnných.

Co když chceme agregovat výsledky ve složitější datové struktuře?

Deklarujeme si vlastní redukci!

```
#pragma omp declare reduction(name:type:expression) \  
    initializer(expression)
```

- name = název vlastní redukce
- type = typ, nad kterým je redukce definována (např. std::vector)
- expression = funkce, která se má vykonávat nad dvěma částečnými výsledky

→ Částečné výsledky jsou uloženy v proměnných `omp_in` a `omp_out`

→ Výsledek redukce uložíme zpět do proměnné `omp_out`

- initializer = jaká má být počáteční hodnota lokální kopie redukované proměnné v každém vlákne (lokální proměnná = `omp_priv`)

Příklad vlastní redukce

```
void merge_elements(element_t& dest, element_t& in) {
    dest = gcd(dest, in);
}

...
#pragma omp declare reduction(merge : element_t : merge_elements)
initializer(omp_priv = 0)
element_t result;
#pragma omp parallel for reduction(merge : result)
for(int i = 0; i < size; i++) {
    // do something with result
}
```

```
void merge_elements(element_t& dest, element_t& in) {
    dest = gcd(dest, in);
}

...
#pragma omp declare reduction(merge : element_t : merge_elements)
initializer(omp_priv = 0)
element_t result;
#pragma omp parallel for reduction(merge : result)
for(int i = 0; i < size; i++) {
    // do something with result
}
```

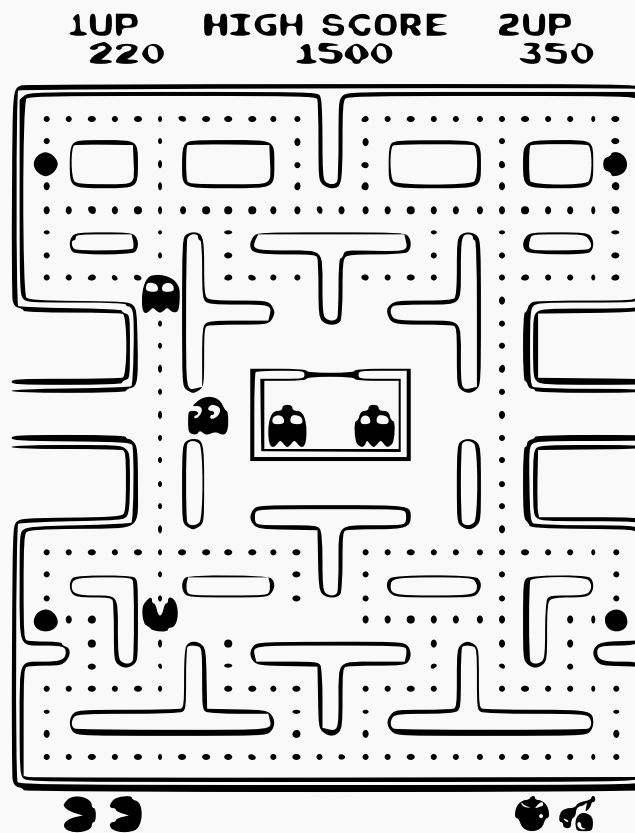
Doimplementujte násobení matice s vektorem

Doimplementujte násobení matice s vektorem Doimplementujte tělo metody `multiply_parallel` v souboru `multiply.cpp`. Vlastní redukci jsme vám již deklarovali. Redukce využívá funkci `merge`. Doimplementujte i tělo redukce (funkce `merge`). Všechny vektory se kterými pracujete jsou řídové!

Semestrální úloha

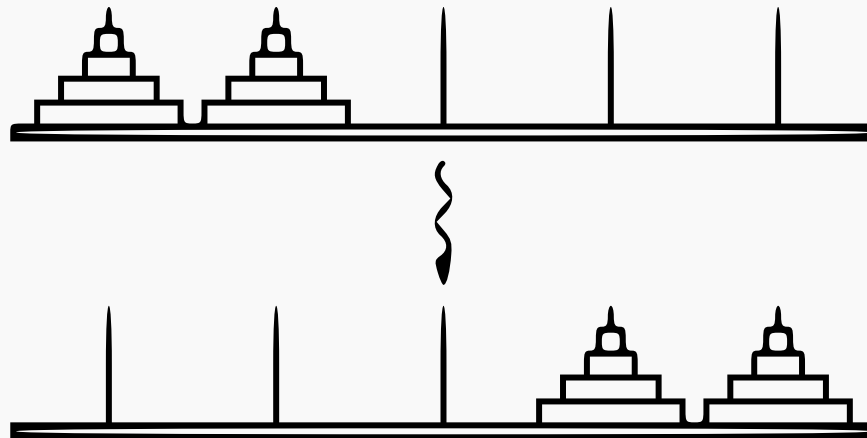
Diskrétní dynamické systémy mají různé **konfigurace** Mezi konfiguracemi lze přecházet pomocí akcí

Jak takové systémy vypadají?



Přesouváme věže z disků z **počátečních** kolíků na **koncové**.

- Jen jeden disk v jednom kroku
- Větší disk nemůže být na menším

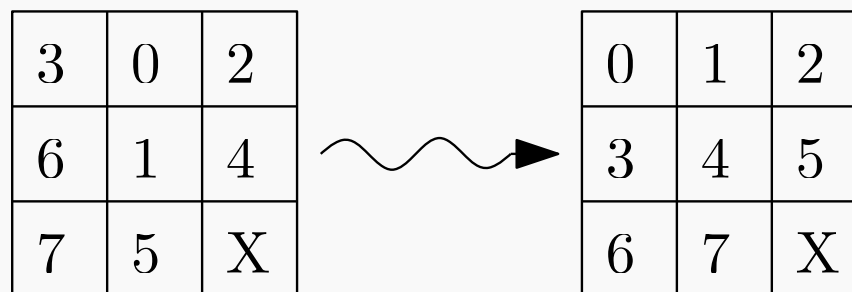


⚠ Doména je korektní, pokud $\text{DISCS} * \text{TOWERS} * \lceil \log_2(\text{RODS}) \rceil \leq 64$

Loydův hlavolam (“sliding puzzle”)

Přesouváme X po poli abychom se dostali z počáteční konfigurace na setříděnou.

- Jen jedno prohození v jednom kroku.
- Prohodíme X s políčkem o jedno nahore, dole, vlevo nebo vpravo.



⚠ Doména je korektní pro rozměry pole 3×3 a 4×4

Pro danou formuli v **konjunktivním normálním tvaru** hledáme ohodnocení, ve kterém bude **splněná**.

$$(\neg x \vee \neg y) \wedge y$$

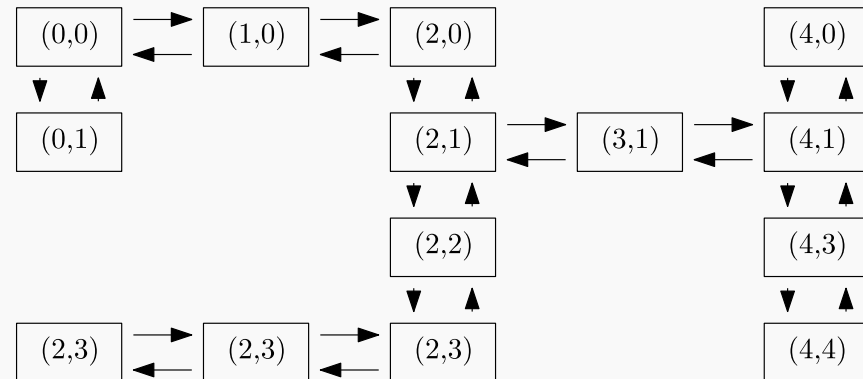
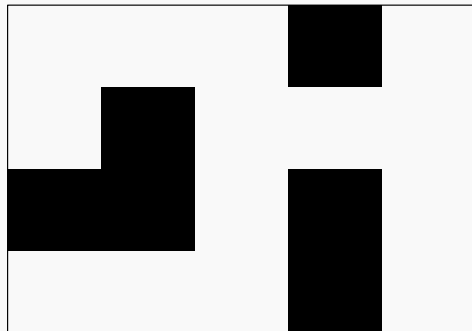
- Přiřazení ohodnocení jen jedné proměnné v jednom kroku.
- Přiřadíme hodnotu jakékoli proměnné s indexem větším než proměnná ohodnocená v minulém kroku.

$$[X, X] \rightsquigarrow [0, 1]$$

⚠ Doména je korektní, pokud $\text{NUM_VARS} \leq 40$

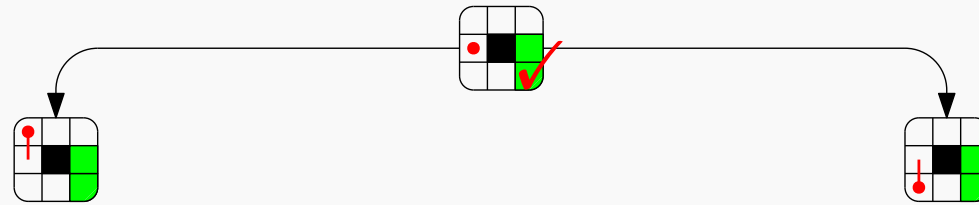
Hledáme cestu z počáteční pozice na koncovou.

- Pohybujeme se o jedno políčko v každém kroku.
- Změníme pozici, pokud nám v cestě nebrání zeď.

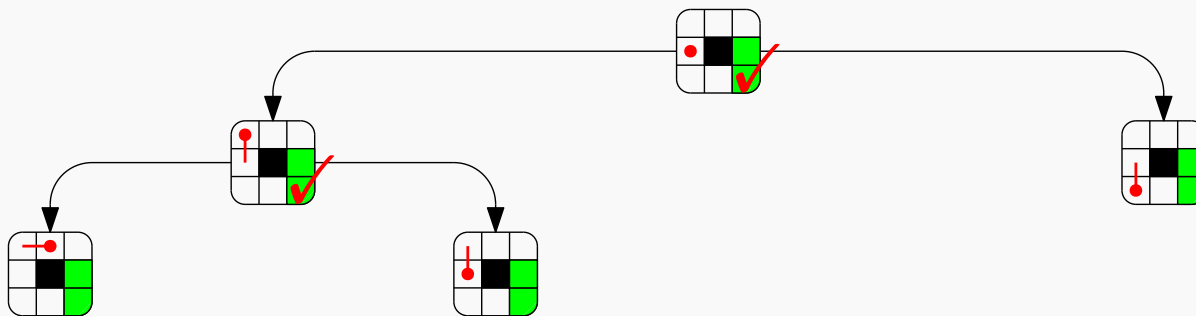


⚠ Doména je korektní, pokud $\log_2(\text{WIDTH} * \text{HEIGHT}) \geq 64$

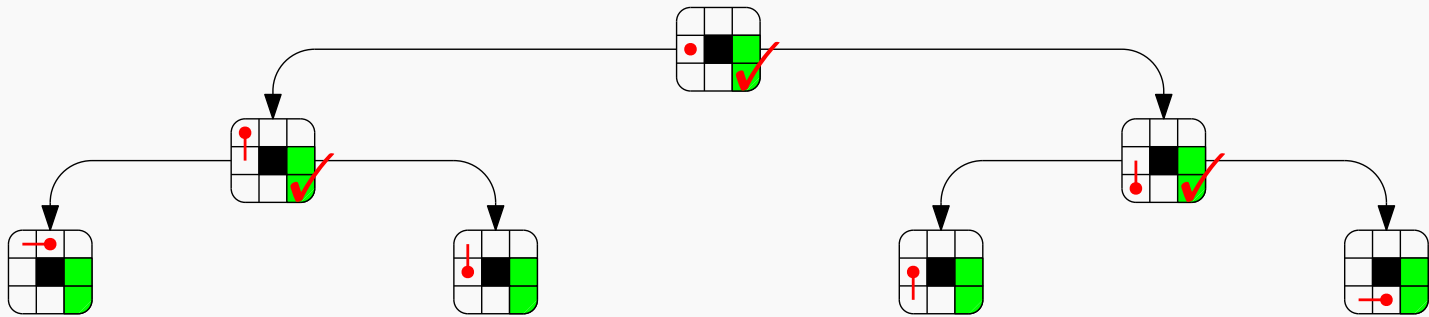




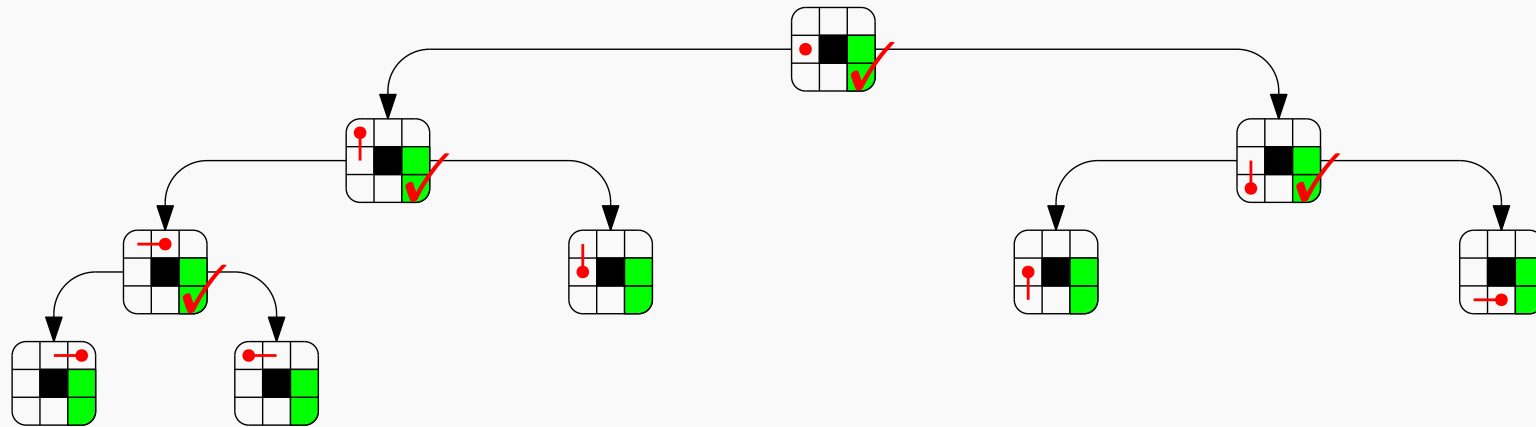
Prohledávání do šířky (BFS)



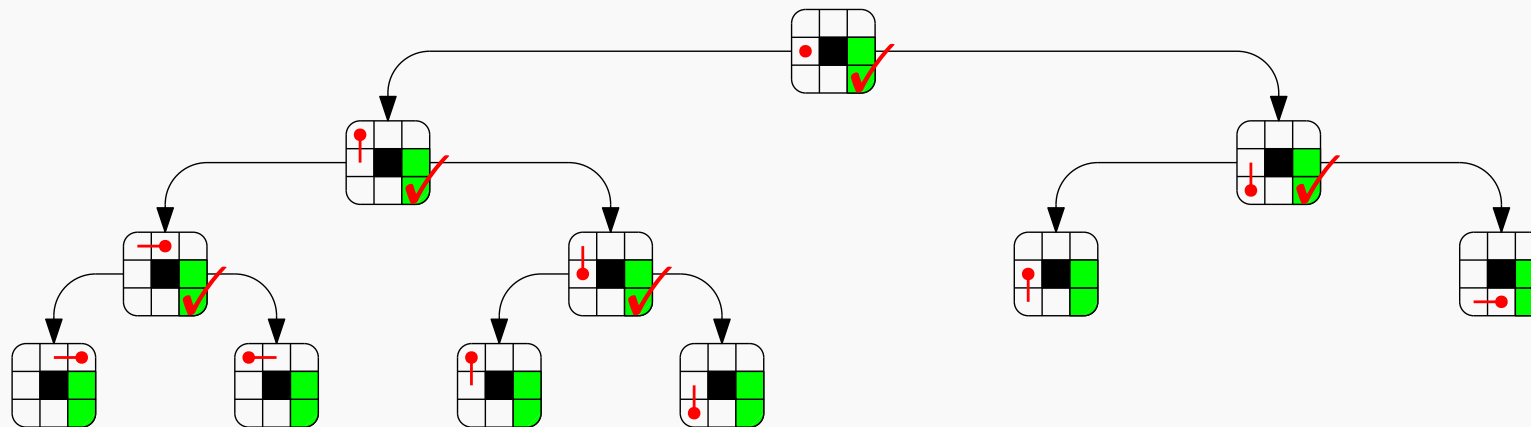
Prohledávání do šířky (BFS)



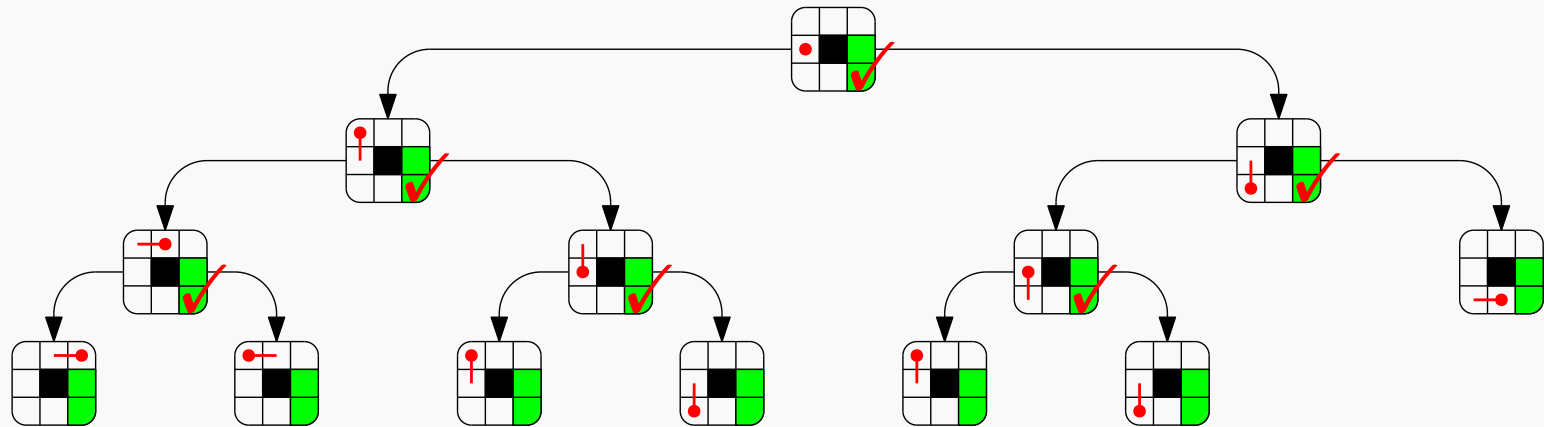
Prohledávání do šířky (BFS)



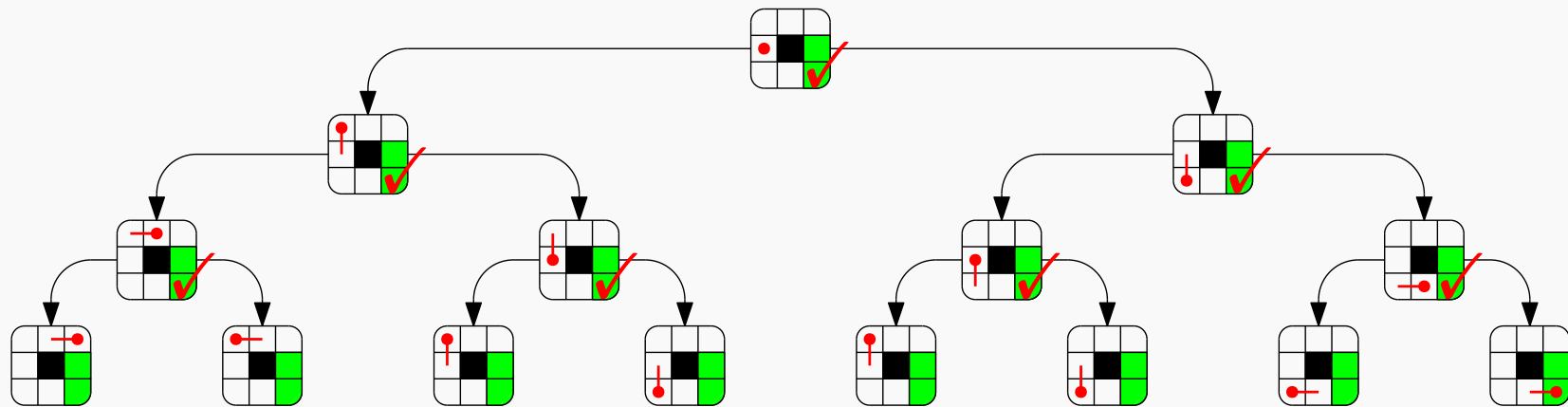
Prohledávání do šířky (BFS)



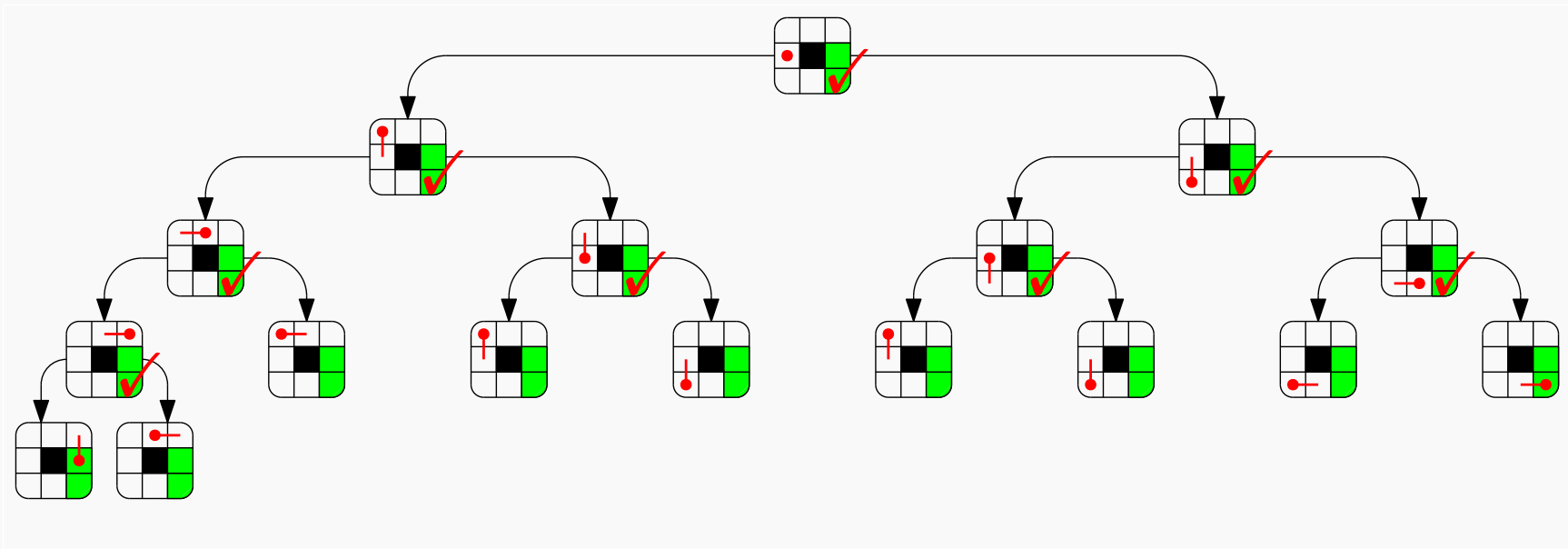
Prohledávání do šířky (BFS)



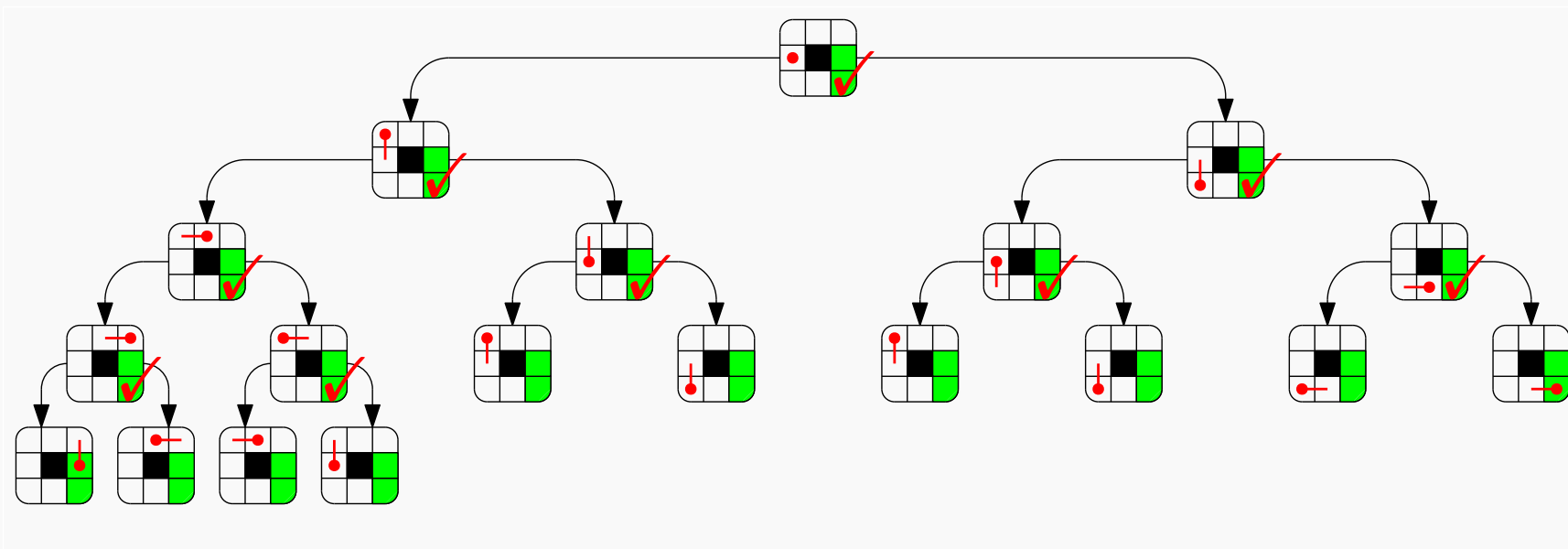
Prohledávání do šířky (BFS)



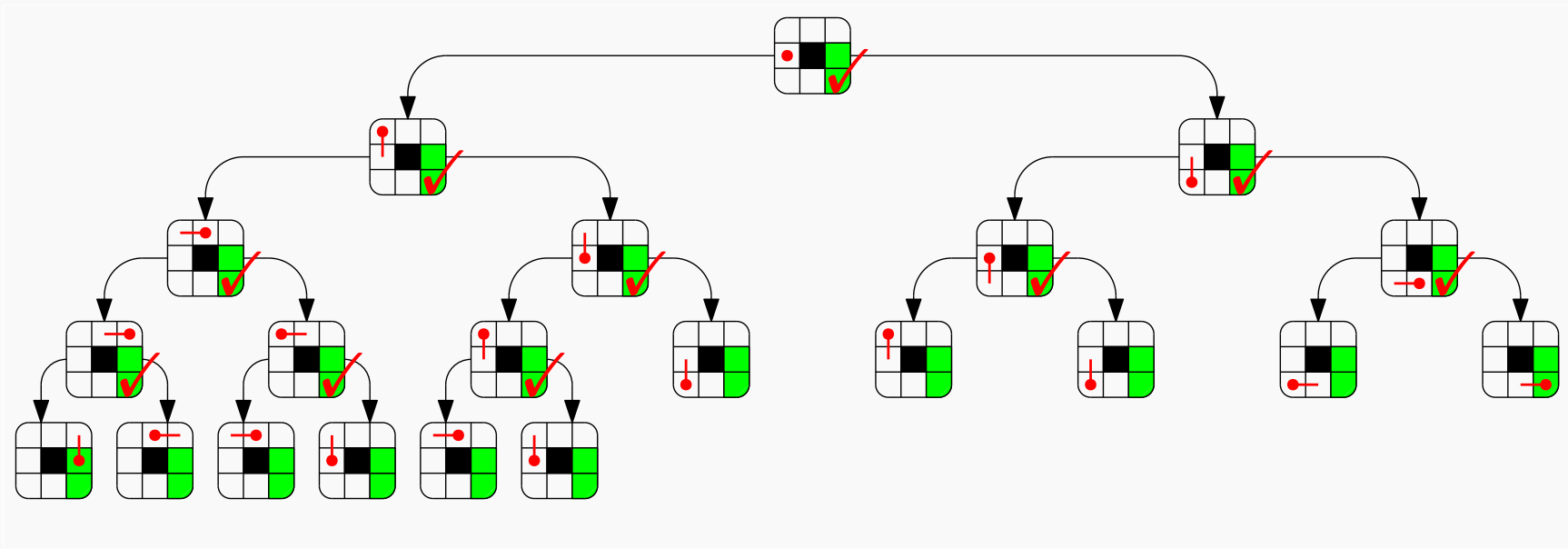
Prohledávání do šířky (BFS)



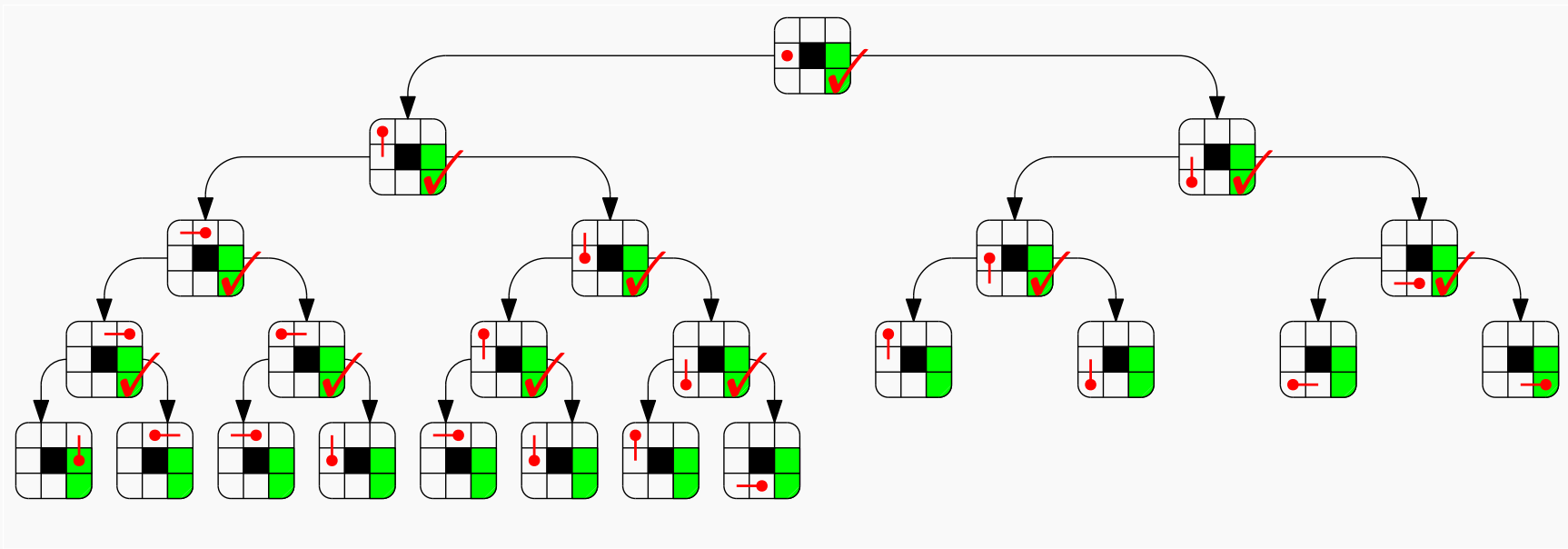
Prohledávání do šířky (BFS)



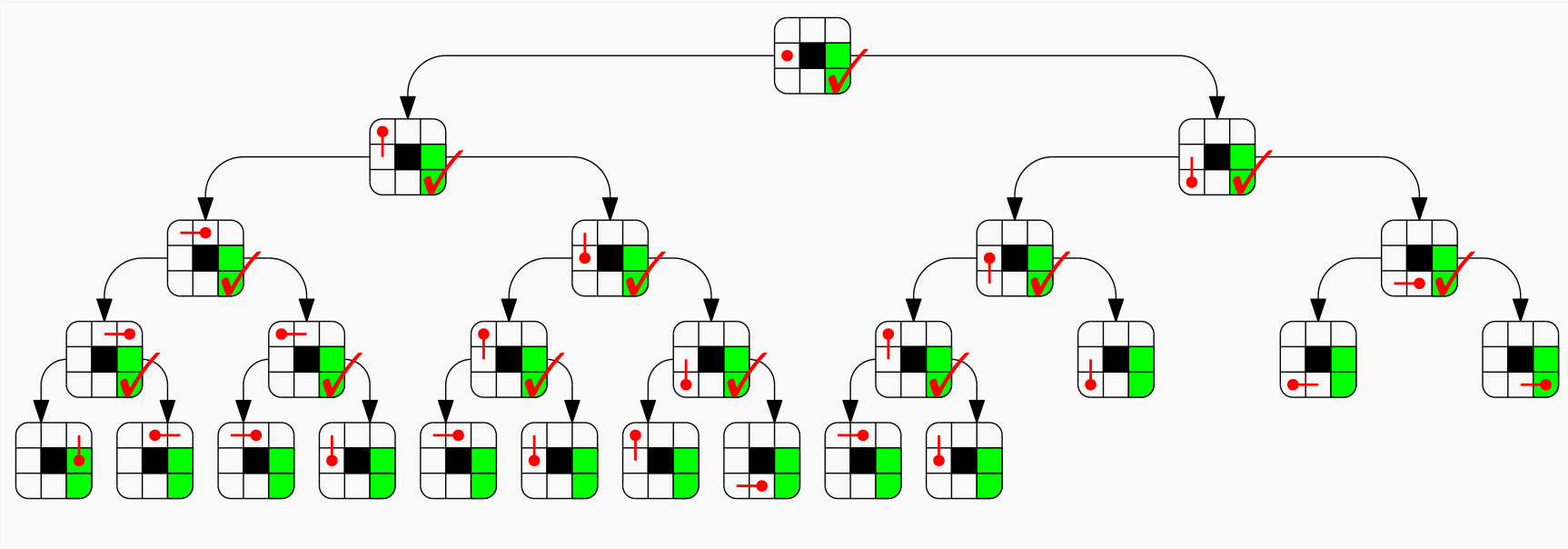
Prohledávání do šířky (BFS)



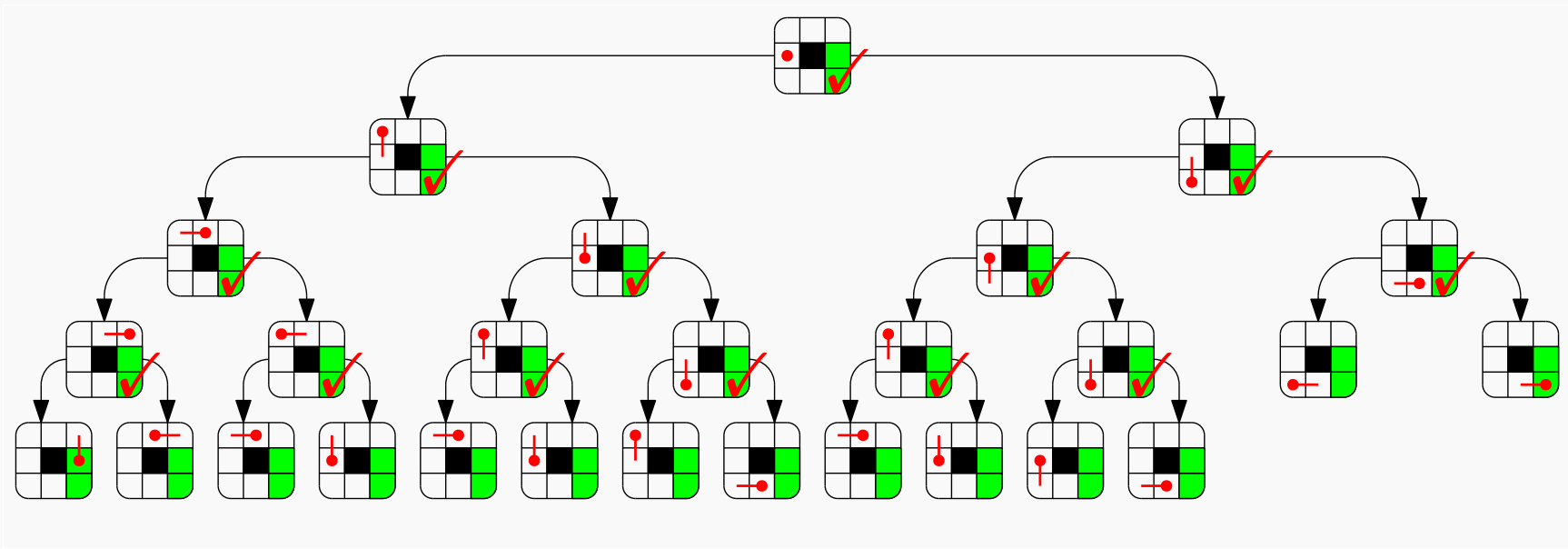
Prohledávání do šířky (BFS)



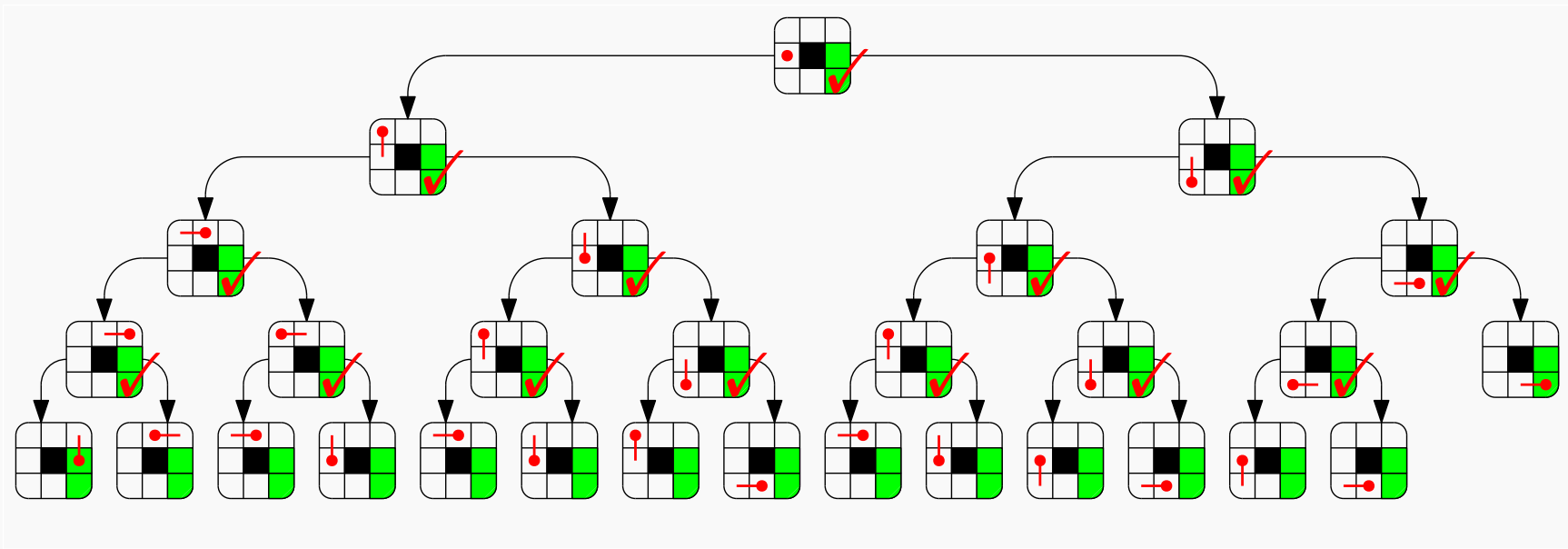
Prohledávání do šířky (BFS)



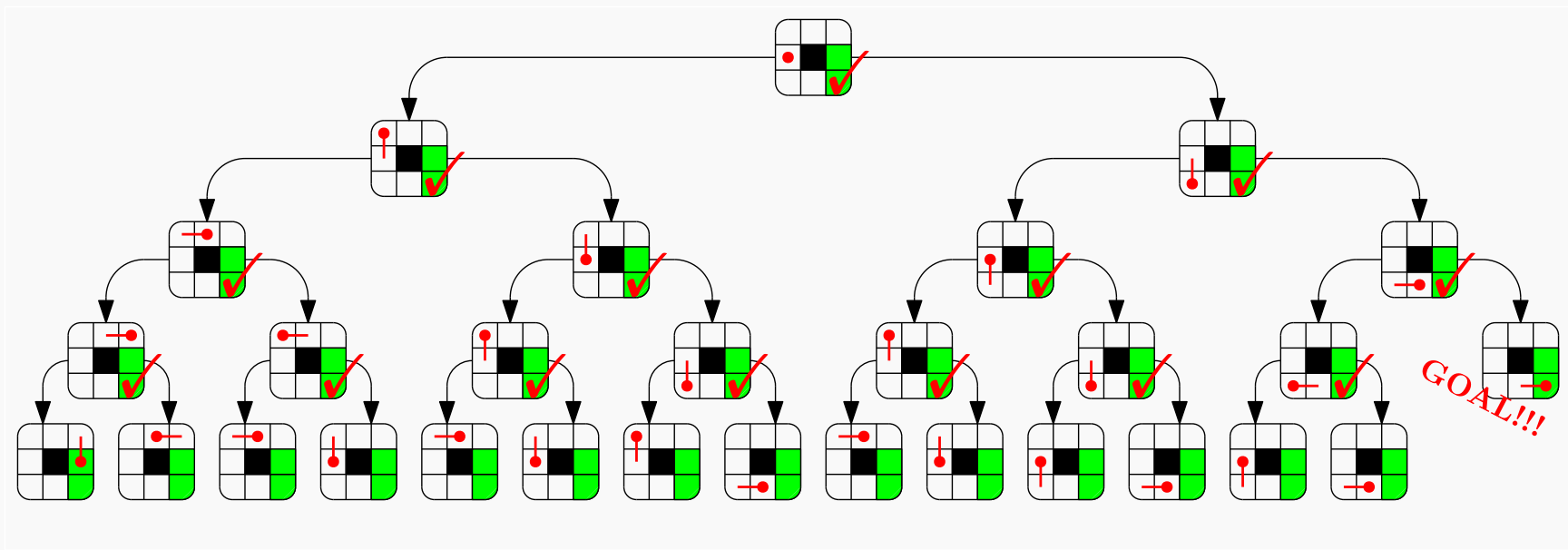
Prohledávání do šířky (BFS)



Prohledávání do šířky (BFS)



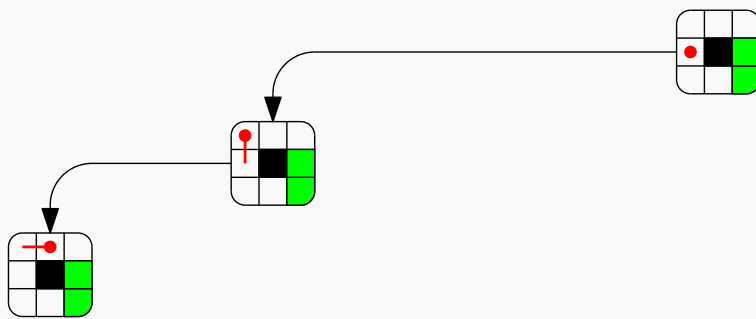
Prohledávání do šířky (BFS)

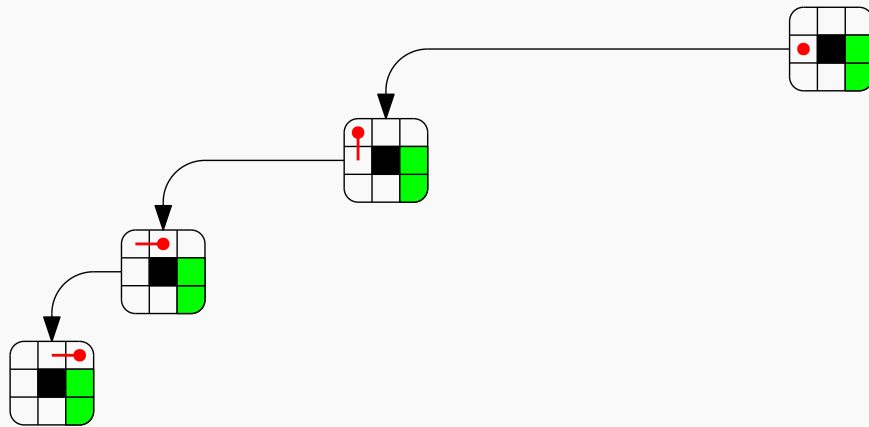


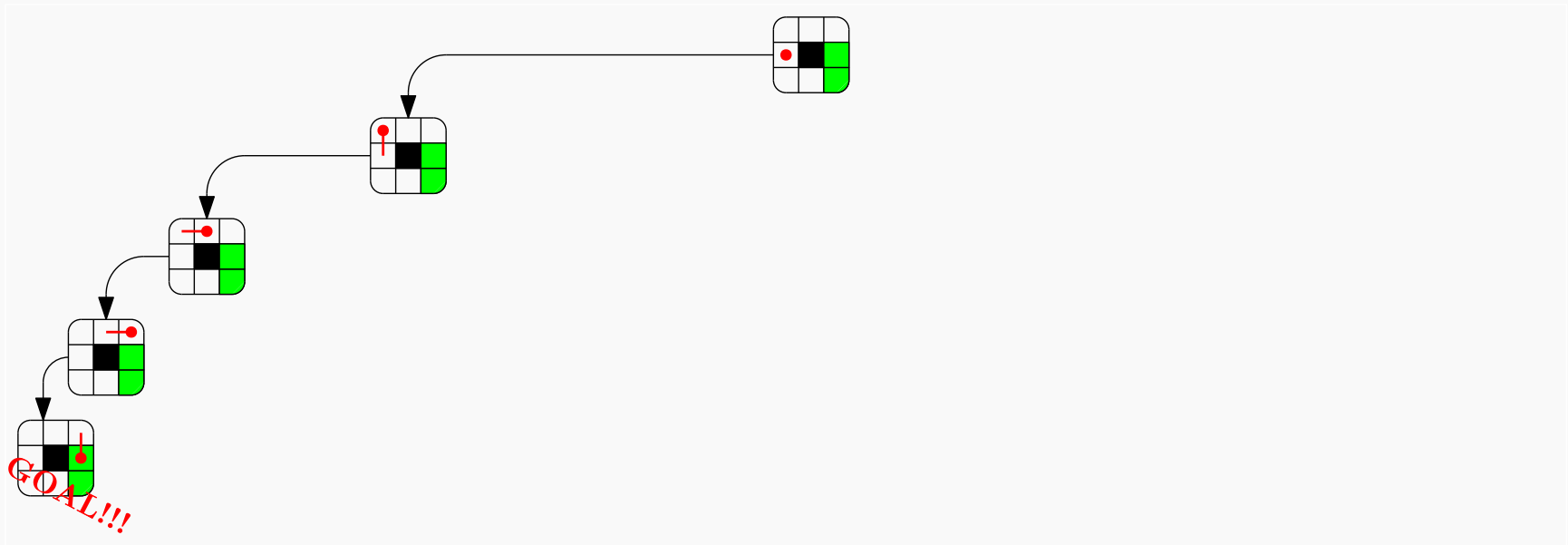
Optimální, ale (potenciálně) s exponenciální pamětí!

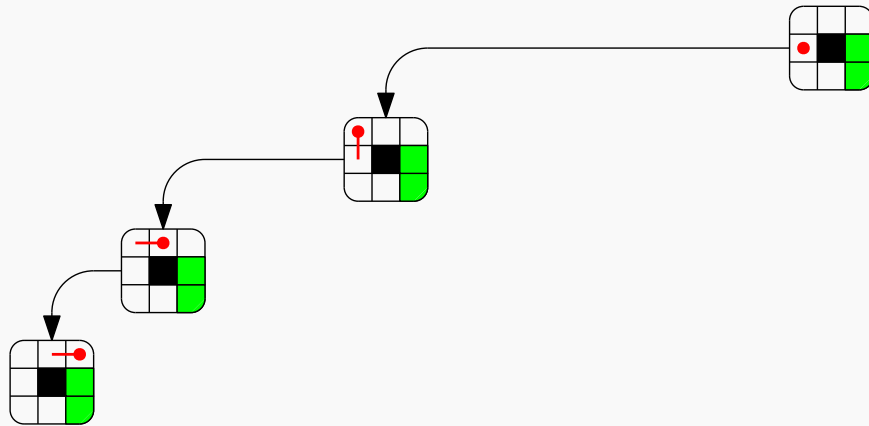


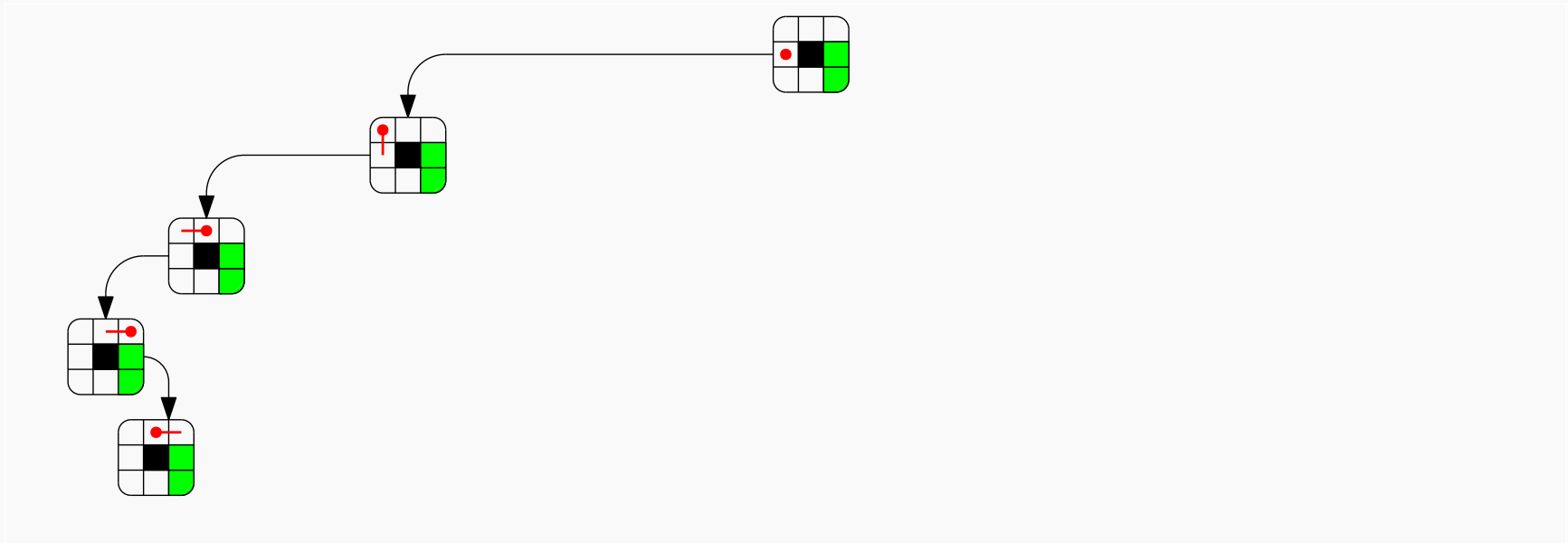


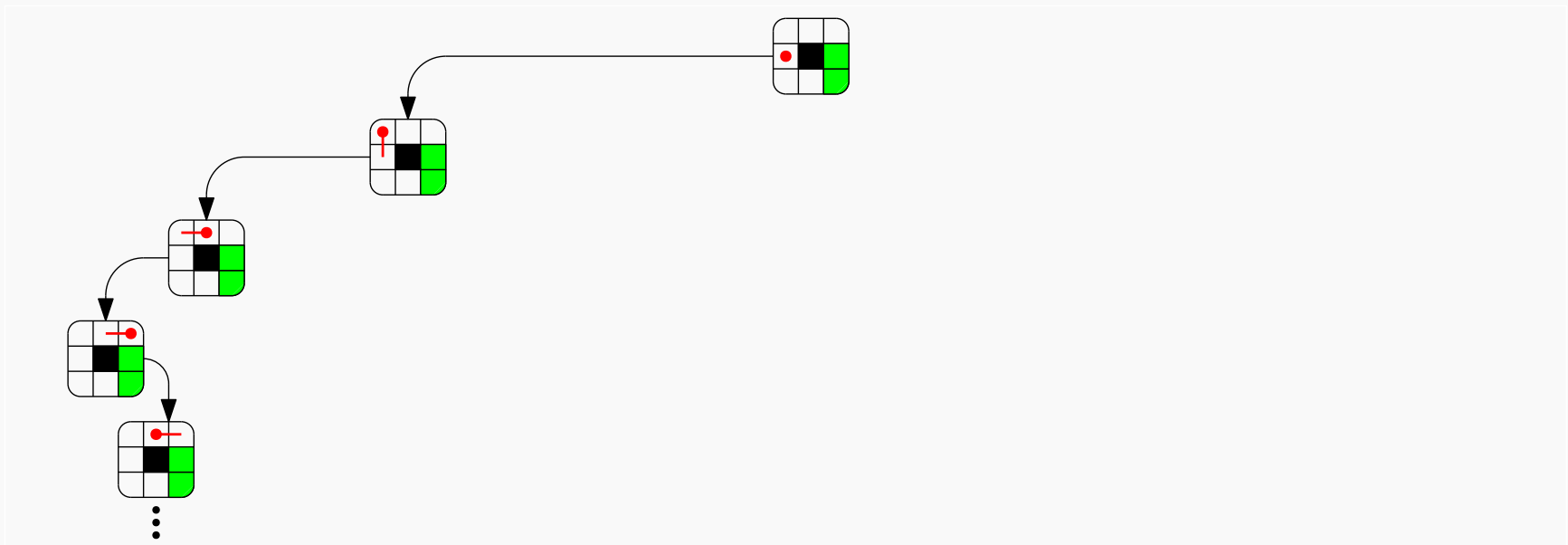












Malá paměťová náročnost, ale bez garancí!

Co když chceme jak garance,
tak malou paměťovou náročnost?

Budeme prohledávat do **omezené** hloubky

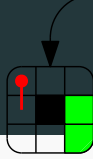
ID-DFS



limit = 1

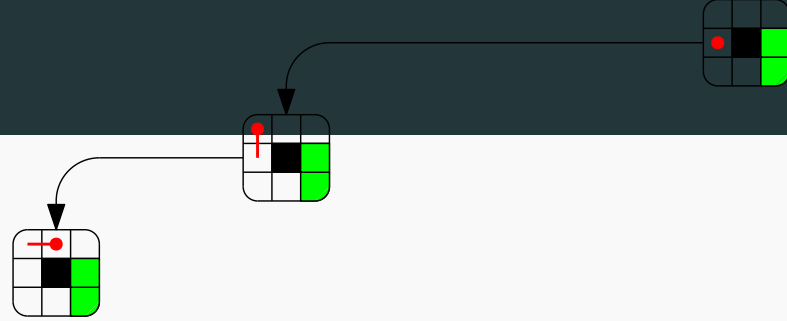
limit = 1

limit = 2

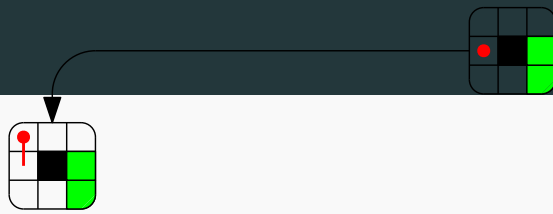


limit = 2

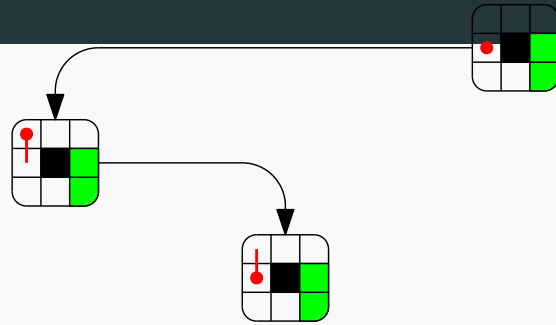
ID-DFS



limit = 2



limit = 2



limit = 2



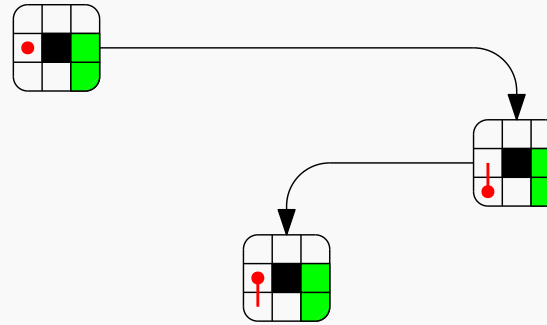
limit = 2



limit = 2



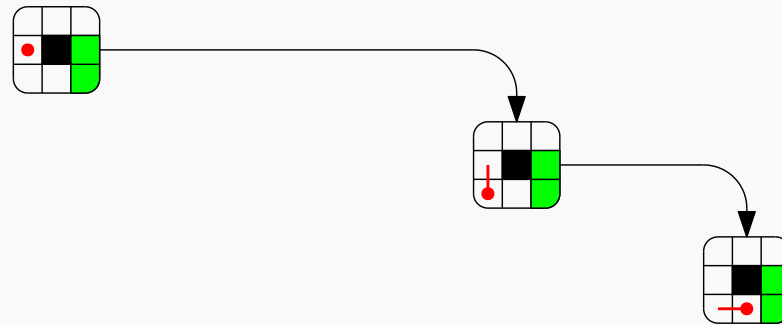
limit = 2



limit = 2



limit = 2



limit = 2



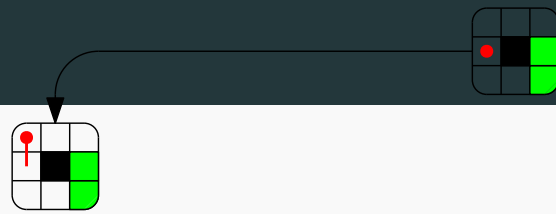
limit = 2



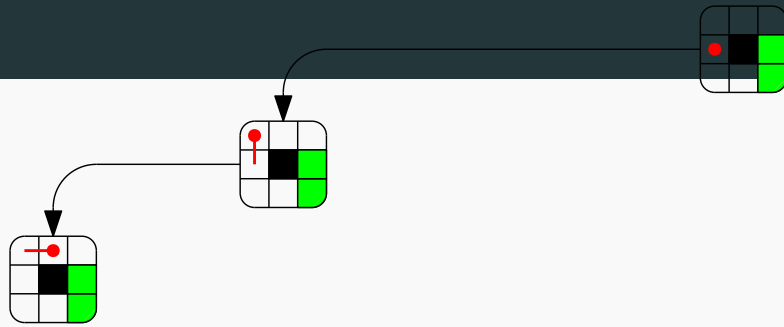
limit = 2



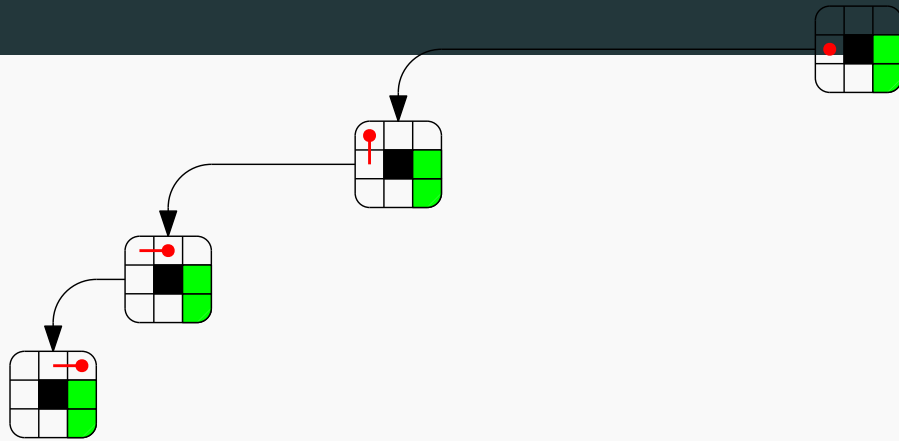
limit = 3



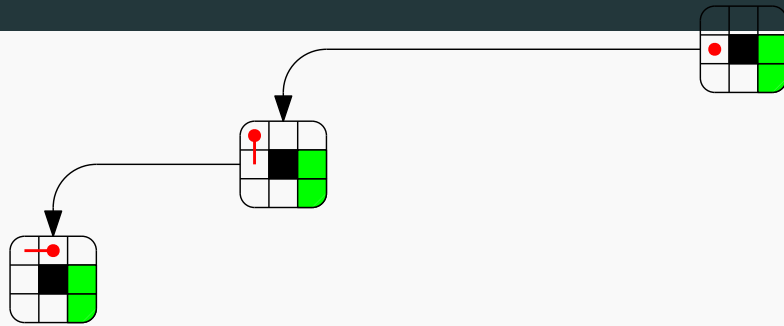
limit = 3



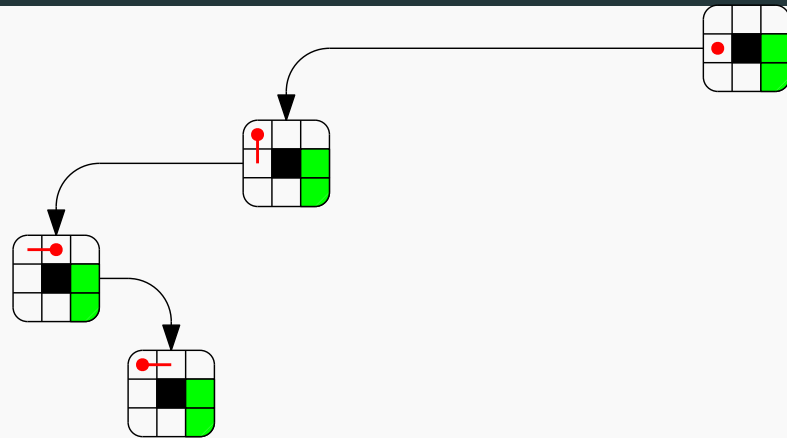
limit = 3



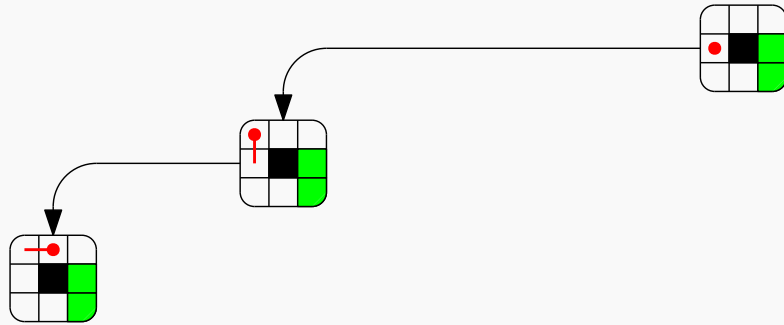
limit = 3



limit = 3



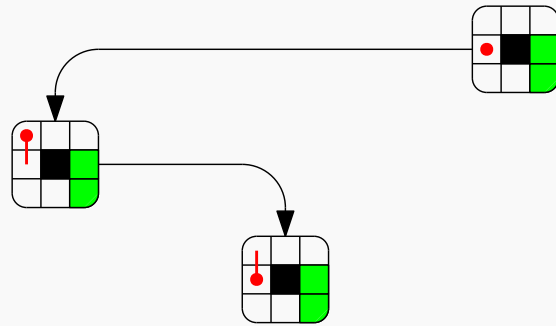
limit = 3



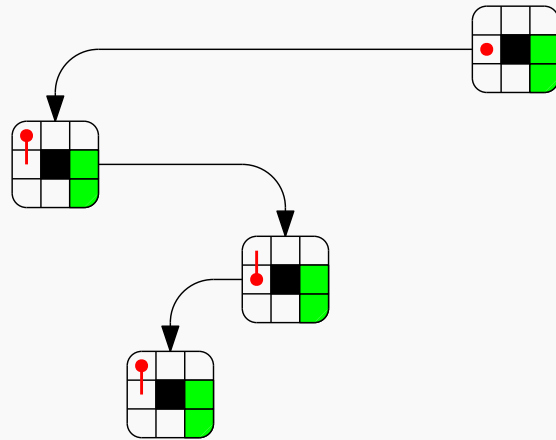
limit = 3



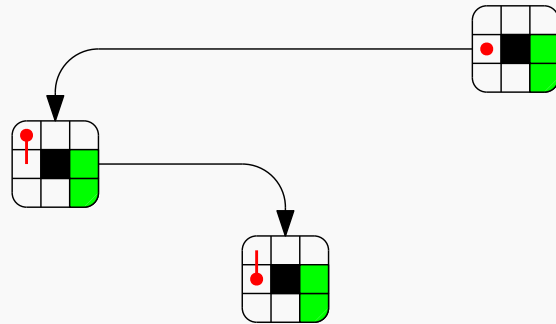
limit = 3



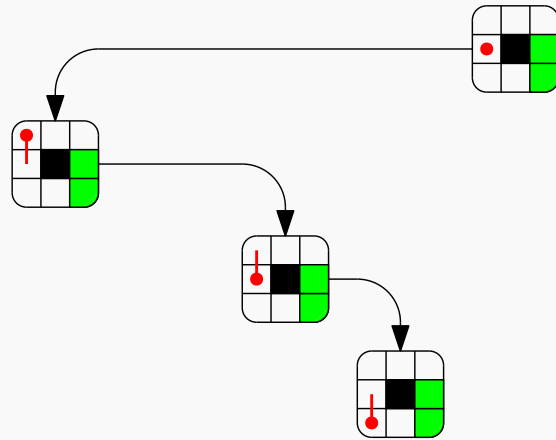
limit = 3



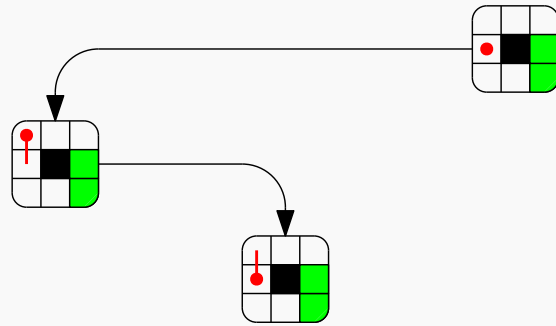
limit = 3



limit = 3



$\text{limit} = 3$



limit = 3



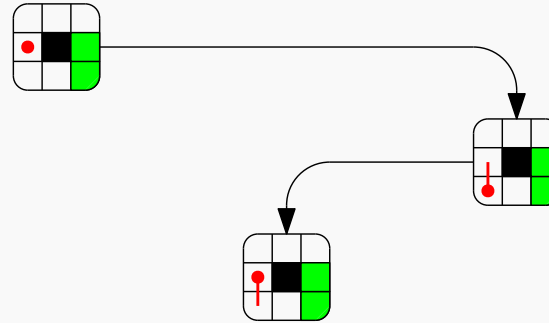
limit = 3



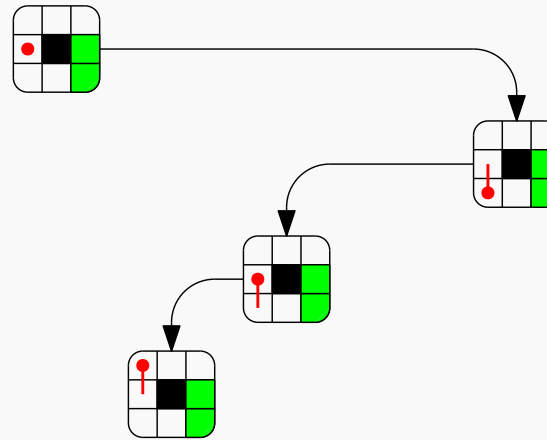
limit = 3



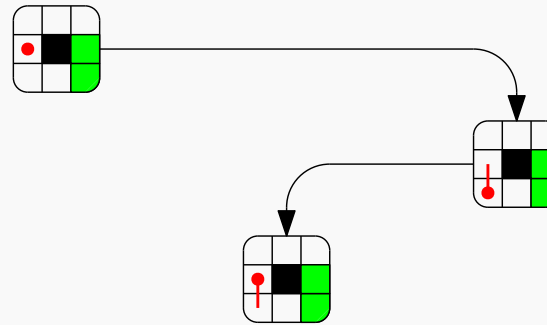
limit = 3



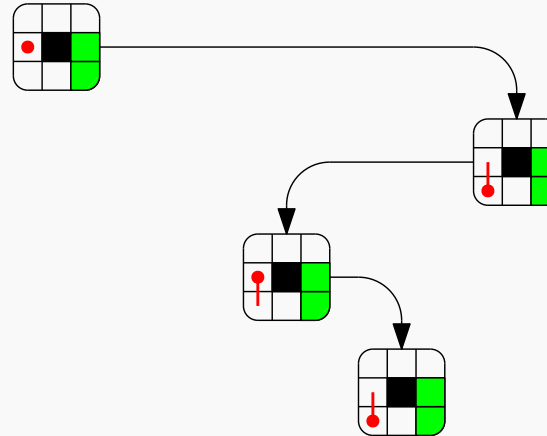
limit = 3



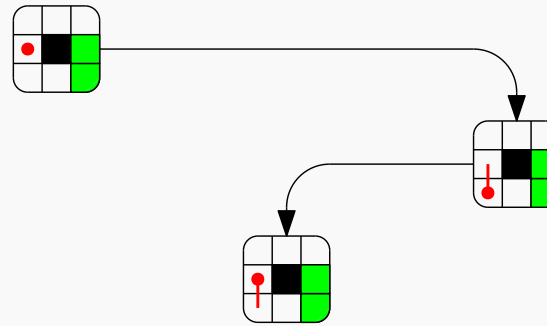
limit = 3



limit = 3



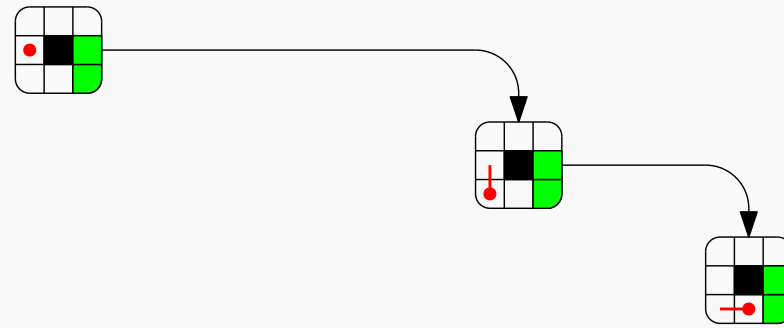
limit = 3



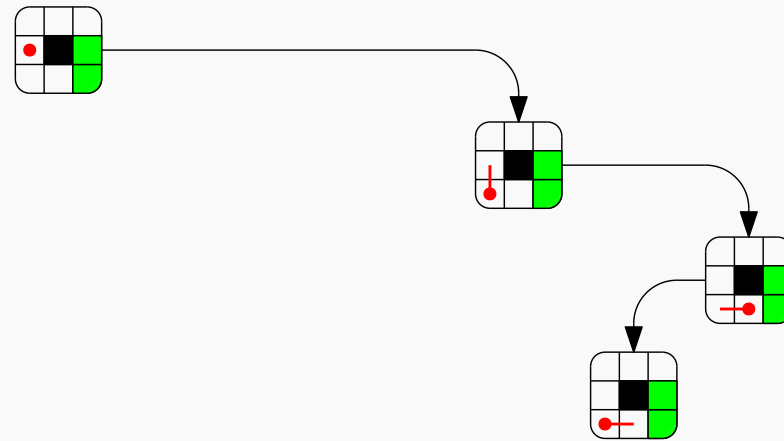
limit = 3



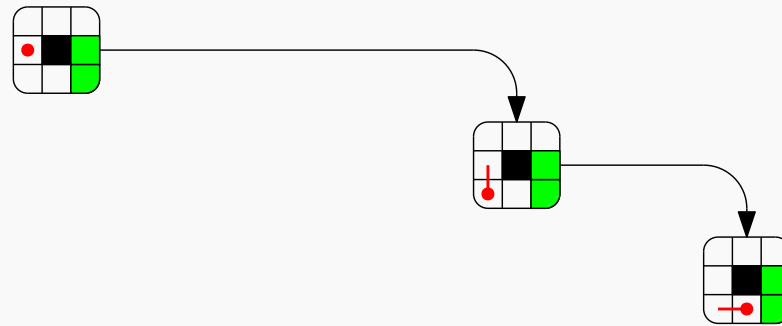
limit = 3



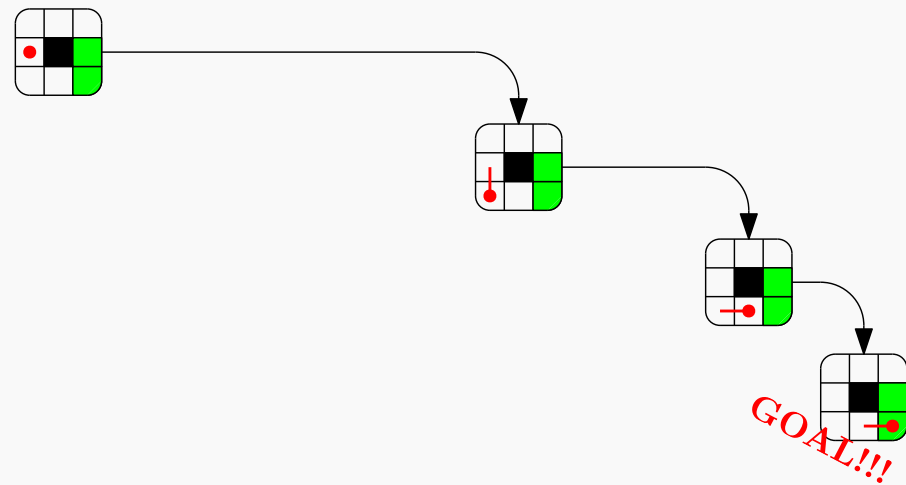
limit = 3



limit = 3



limit = 3



limit = 3

V paměti máme pouze aktuální cestu :-)

Nechceme vás příliš ovlivňovat...

Některé uzly jsme navštěvovali mnohokrát (i na stejné cestě!) (Zkuste zabránit vstupování do stejných stavů – v paralelní verzi možná budete muset dělat kompromisy...)

Nemusíte implementovat přesné verze těchto algoritmů (Například, v ID-DFS si můžete pamatovat o něco víc než jen aktuální cestu. Také můžete malé části stromu procházet pomocí BFS...)

... ale především po vás budeme chtít tyto algoritmy paralelizovat :-)

Zejména v ID-DFS algoritmu je správná správa paměti nutností! (Váš algoritmus musí být schopný běžet v prostředí s omezenou pamětí)

Dosud jste se pravděpodobně setkali zejména s **raw pointery** ; například `state*` `s`;

Veškerá zodpovědnost za správnou správu paměti by byla na vás :-)

Naším cílem ale není zkoušet vás z toho, kdo je lepší programátor v C/C++...

Proto správu paměti (částečně) přebíráme za vás!

Jak to děláme?

C++11 shared pointers

S RAII návrhovým vzorem jsme se už setkali u `std::unique_lock`.

```
template <typename lock_t>
class unique_lock {
private:
    lock_t& mutex;

public:
    unique_lock(lock_t& mutex) : mutex(mutex) {
        mutex.lock();
    }

    ~unique_lock() {
        mutex.unlock();
    }
};
```

S RAII návrhovým vzorem jsme se už setkali u `std::unique_lock`.

```
template <typename lock_t>
class unique_lock {
private:
    lock_t& mutex;

public:
    unique_lock(lock_t& mutex) : mutex(mutex) {
        mutex.lock();
    }

    ~unique_lock() {
        mutex.unlock();
    }
};
```

Vlastnictví zámku je unikátní.

Obdobně funguje i `std::weak_ptr` pro správu pointeru.

Paměť se uvolní okamžitě po zániku instance `std::weak_ptr`!

Co když to ale nechceme?

Obdobně funguje i `std::weak_ptr` pro správu pointeru.

Paměť se uvolní okamžitě po zániku instance `std::weak_ptr`!

Co když to ale nechceme?

- Instanci `std::weak_ptr` uložíme například do vektoru. Dále používáme raw pointer (získaný přes `ptr.get()`). Paměť se ale uvolní po zániku vektoru, raw pointer přestane být validní!

Obdobně funguje i `std::unique_ptr` pro správu pointeru.

Paměť se uvolní okamžitě po zániku instance `std::unique_ptr`!

Co když to ale nechceme?

- Instanci `std::unique_ptr` uložíme například do vektoru. Dále používáme raw pointer (získaný přes `ptr.get()`). Paměť se ale uvolní po zániku vektoru, raw pointer přestane být validní!
- Důsledně budeme spravovat, kdo aktuálně pointer vlastní. Paměť zanikne, když ho nějaká funkce nikomu nepředá (pomocí `std::move`).

To je celkem dost pracné!

Obdobně funguje i `std::unique_ptr` pro správu pointeru.

Paměť se uvolní okamžitě po zániku instance `std::unique_ptr`!

Co když to ale nechceme?

- Instanci `std::unique_ptr` uložíme například do vektoru. Dále používáme raw pointer (získaný přes `ptr.get()`). Paměť se ale uvolní po zániku vektoru, raw pointer přestane být validní!
- Důsledně budeme spravovat, kdo aktuálně pointer vlastní. Paměť zanikne, když ho nějaká funkce nikomu nepředá (pomocí `std::move`).

To je celkem dost pracné!

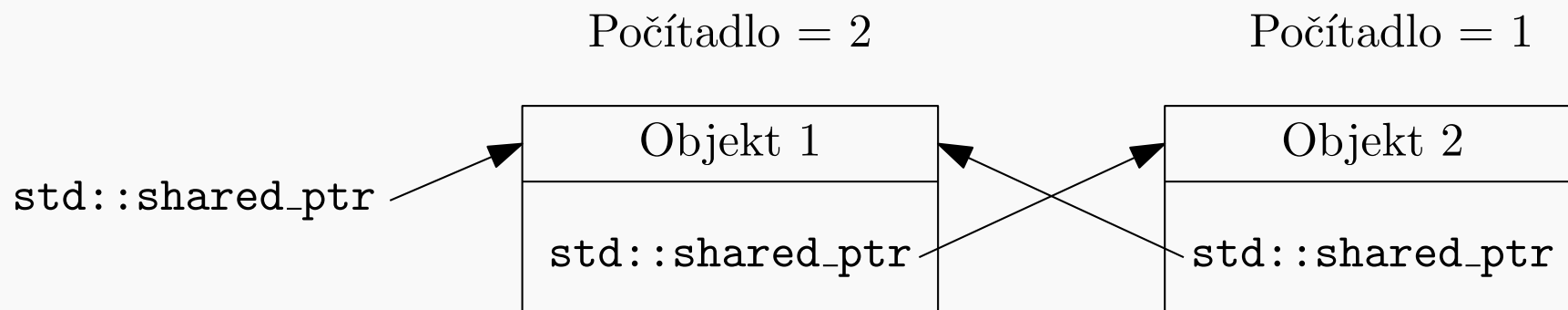
- Použijeme `std::shared_ptr` a pointery předáváme jako kdyby to byly raw pointery.

Shared pointers = jednoduchá automatická správa paměti
(jednoduchý „garbage collector“)

- Každý shared pointer si drží počítadlo, kolik instancí `std::shared_ptr` na dané místo v paměti ukazuje
 - Při kopírování shared pointeru se počítadlo zvýší
 - Při rušení instance se počítadlo sníží
- Když počítadlo dospěje na nulu, paměť se dealokuje

Při rozumné práci fungují výborně, ale...

⚠ Pozor na cykly!!!

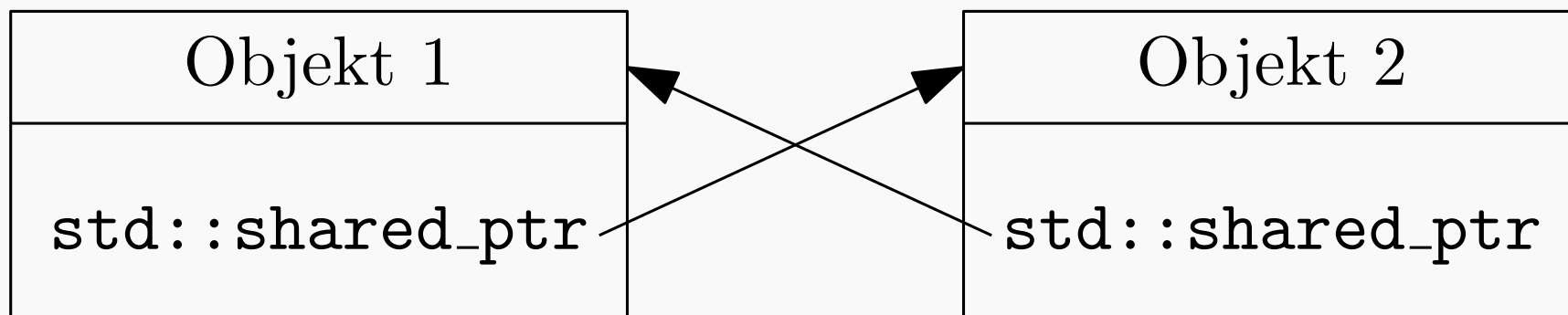


Při rozumné práci fungují výborně, ale...

⚠ Pozor na cykly!!!

Počítadlo = 1

Počítadlo = 1



Se shared pointery dokonce můžete provádět atomické operace:

```
// 'atomicka promenna'
std::shared_ptr<const state>& concurrent_ptr = ... ;

// ocekavana hodnota:
std::shared_ptr<const state> local_copy
    = atomic_load(&concurrent_ptr);

// hodnota k zapsani
std::shared_ptr<const state> new_ptr = ... ;

while( ... ) {
    if(atomic_compare_exchange_strong(
        &concurrent_ptr, &local_copy, new_ptr)) {
        break;
    }
}
```


Doimplementujte metodu `iddfs`

Doimplementujte metodu `iddfs` Doimplementujte tělo metody `iddfs`, která bude vykonávat sekvenční prohledávání do hloubky s definovanou maximální hloubkou (kterou budete iterativně zvyšovat, dokud nenarazíte na cíl). Vyzkoušejte si práci se sdílenými ukazateli a s doménovými metodami `is_goal()` a `next_states()`.