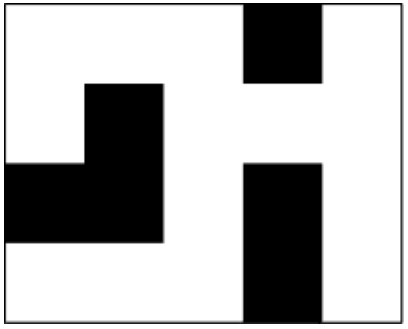
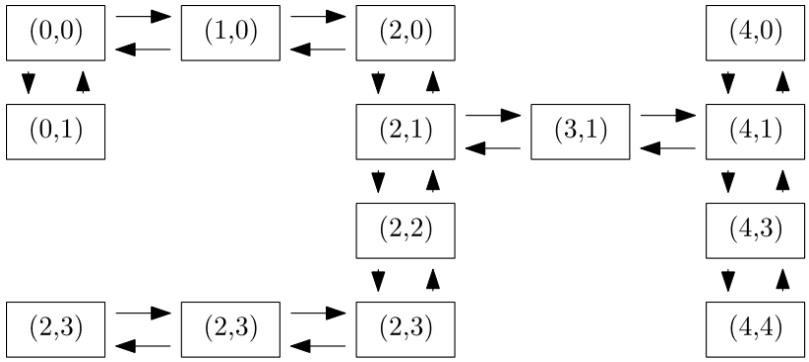
1. semestrální práce - Prohledávání stavového prostoru

Diskrétní dynamický systém je jednoznačně určený svým počátečním stavem, množinou možných akcí a přechodovou funkcí, která pro daný stav a danou akci určí stav následující. Takovým systémem je například následující bludiště.



V tomto případě je počátečním stavem pozice, na které v bludišti začínáme (např. (0,0)) a množinou možných akcí pak { Up, Down, Left, Right } značící pohyb v horizontálním nebo vertikálním směru. Přechodová funkce nám říká, do jakých pozic se můžeme posunout v případě, že směr není zahrazený stěnou. Pokud je zahrazený, zůstaneme na místě.

Stavový prostor pro daný diskrétní dynamický systém je pak množina všech konfigurací, kterých může systém nabývat. Graf stavového prostoru nám na základě přechodové funkce určuje, jakým způsobem se můžeme pomocí vykonávání akcí pohybovat mezi jednotlivými konfiguracemi. Na následujícím obrázku můžete vidět graf stavového prostoru pro naše bludiště.



Vaším úkolem bude naimplementovat paralelní verze dvou základních algoritmů pro prohledávání stavových prostorů (tedy potenciálně nekonečných grafů).

1. **Prohledávání do šířky** (*breadth-first search*) garantuje nalezení nejkratší cesty k cíli v neváženém grafu (za předpokladu, že je cíl dosažitelný v konečné hloubce). V případě, že o problému, který řešíme nemáme žádné další informace, které by nás nasměrovaly k cíli, prohledávání do šířky patří k nejefektivnějším algoritmům. Jeho zásadní nevýhodou je vysoká paměťová složitost, protože si musíme pamatovat všechny uzly, které jsou dosažitelné v daném počtu kroků. To je v případě, že je nejbližší cíl ve větší hloubce a/nebo se graf rychle větví, nepřijatelné.
2. **Prohledávání do hloubky** (*depth-first search*, DFS) v nekonečných grafech negarantuje ani nalezení nějaké cesty v grafu, natož nejkratší. Jeho velkou výhodou je nízká paměťová složitost, protože si potřebujeme pamatovat pouze uzly na aktuální uvažované cestě. Jedním způsobem, jak obejít absenci garancí, je použití iterative deepeningu (algoritmus ID-DFS). Princip iterative deepeningu se dá ve stručnosti shrnout takto: Stavový prostor vždy prohledáváme do určité konečné hloubky, kterou nastavujeme adaptivně. Začneme s nízkou/nulovou hloubkou a prohledáme do hloubky všechny cesty do této délky. Pokud nenalezneme žádné řešení, hloubku zvýšíme a prohledáváme odznova. Cenou za garanci nalezení nejkratší za použití iterative deepeningu je to, že stavový prostor zpravidla musíme prohledat mnohokrát.

Ve Vašem řešení samozřejmě můžete kombinovat prvky obou přístupů, respektive přijít s vlastní strategií prohledávání stavového prostoru. Například můžete prvky prohledávání do šířky používat v rámci ID-DFS (samozřejmě v omezené míře kvůli paměťové náročnosti prohledávání do šířky) a naopak.

Pro Vaše testování máte k dispozici 4 domény:

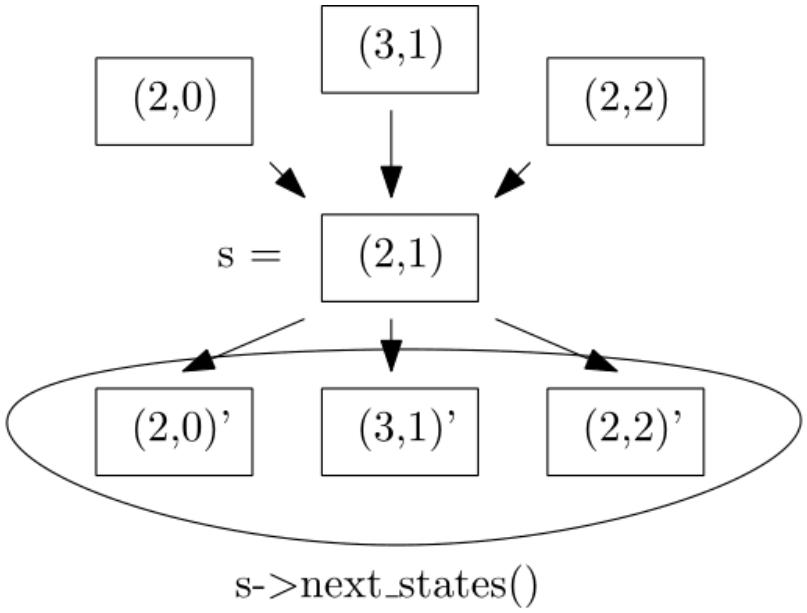
- *Hanojské věže* parametrizované počtem kolíků (`RODS`), počtem věží (na začátku obsazených kolíků, `TOWERS`) a počtem kotoučů v každé věži (`DISCS`). Doména je korektní (funkce `get_identifier()` vrací unikátní identifikátory), pokud platí `DISCS*TOWERS*ceil(log2(RODS)) ⇐ 64` a zároveň `TOWERS == 1` .

- *Splňování booleovských funkcí (SAT)*. Stavový prostor je tvořen stromem možných rozhodnutí o hodnotě každého literálu. Doména je parametrizovaná počtem literálů (`NUM_VARS`), počtem termů ke splnění (`NUM_CLAUSES`) a maximální velikostí termu (`MAX_CLAUSE_SIZE`). Doména je korektní pokud `NUM_VARS ⇐ 40` . V této doméně je i možnost vygenerovat zadání, kde je cena za různé zvolené proměnné různá (`UNIFORM=false`) - v opačném případě mají všechny přechody cenu 1.
- *Sliding puzzle* je parametrizované rozměrem hracího plánu (`SIZE` x `SIZE`) a počtem náhodných tahů, které jsou použity pro vygenerování iniciální konfigurace (`SOLUTION_DEPTH`). Doména je korektní pro `SIZE=3` a `SIZE=4` .
- *Bludiště*, které je parametrizované rozměrem hracího plánu (`WIDTH` x `HEIGHT`). Doména je korektní, pokud `log2(WIDTH*HEIGHT) ⇐ 64` . Obdobně jako u SATu je zde možnost proměnlivých cen za přechod při volbě parametru `UNIFORM=false` .

Zda je stav řešením (a tedy jedním z cílů) můžete zjistit voláním funkce `is_goal()` .

Správnost Vašeho řešení můžete testovat na malých instancích jednotlivých domén, například na Hanojských věžích s třemi kolíky, jednou věží a dvěma kotouči.

Stáhněte si balíček sem_01.zip. Prohledávání do šířky implementujte do souboru `bfs.cpp` . Paměťově efektivní prohledávání (které budeme spouštět na velkých problémech, na kterých prohledávání do šířky selže kvůli paměťové náročnosti!) implementujte do souboru `iddfs.cpp` . Každá z prohledávacích metod vrací ukazatel (typu `std::shared_ptr<const state>`) na cílový stav, který byl dosažen pomocí optimální cesty. Na začátku prohledávání dostanete na vstupu pouze ukazatel na počáteční stav (typu `std::shared_ptr<const state>`). Následující stavy můžete získat voláním metody `next_states()` na daném stavu. Pro více informací si pročtěte komentáře v souboru `state.h` . **Pokud optimálních řešení existuje více, vracejte stav s minimálním identifikátorem** (viz metoda `state::get_identifier()`).



Hodnocení úlohy

Za úlohu můžete získat maximálně **12 bodů** v závislosti na rychlosti vašeho řešení. Body se Vám budou nasčítávat podle následujících podúloh.

1. *Nalezení jakékoliv cesty pomocí prohledávání do šířky.* Pokud Vaše implementace nalezne libovolné řešení problému (ne nezbytně optimální) a přitom dosáhnete díky paralelizaci zrychlení oproti referenční sekvenční implementaci BFS, získáte **2 body**.
2. *Nalezení nejkratší cesty pomocí prohledávání do šířky.* Pokud Vaše implementace nalezne optimální řešení v neváženém grafu (tj., nejkratší cestu), můžete získat až **3 body** (podle rychlosti Vašeho řešení).
3. *Nalezení nejkratší cesty pomocí paměťově efektivního algoritmu (ID-DFS).* Pokud Vaše implementace ID-DFS nalezne nejkratší cestu v neváženém grafu, můžete získat až **5 bodů** (dle rychlosti Vašeho řešení). Počítejte s tím, že Váš ID-DFS budeme spouštět s omezenou pamětí (například pouze nízké stovky MB).
4. *Nalezení nejlevnější cesty ve váženém grafu pomocí paměťově efektivního algoritmu (ID-DFS).* Pokud je Vaše implementace schopná nalézt optimální řešení i ve váženém grafu (tj., nejlevnější cestu), můžete získat až **2 body** (v závislosti na rychlosti Vašeho řešení).

Zazipované soubory `bfs.cpp` a `iddfs.cpp` odevzdávejte do systému BRUTE do **neděle 10.5.2020 23:59 CET**. Před odevzdáním se ujistěte, že jste z kódu odebrali všechny ladící výpisy.

Pokud byste chtěli algoritmus pro nalezení nejlevnější cesty ve váženém grafu odevzdat zvlášť, naimplementujte tento algoritmus v tomto souboru a odevzdejte ho spolu s `bfs.cpp` a `iddfs.cpp`.