Zadání třetího projektu SUI 2024/25

Karel Beneš

4. prosince 2024

Cílem tohoto projektu v SUI je vytvořit sadu prohledávacích algoritmů a demostrovat jejich funkčnost na hře FreeCell. Prostředí pro vývoj a evaluaci algoritmů je k dispozici na Githubu[1]. Termín odevzdání je 5. ledna 2024 ve 23:59:59, odevzdává se ve Studisu.

V rámci projektu máte vytvořit celkem tři díly:

1. Prohledávání do šířky.
2. Prohledávání do hloubky, s uživatelsky kontrolovatelným limitem hloubky (`--dls-limit DEPTH`). Limit je na délku řešení, která by tak měla být nejvýše `DEPTH` akcí. Ekvivalentně, výchozí stav můžete považovat za hloubku 0.
3. Algoritmus A*, s uživatelsky volitelnou heuristikou (`--heuristic nb_not_home | student`). Heuristika `student` není součástí letošního zadání, metodu `StudentHeuristic::distanceLowerBound` neupravujte, nechť tedy stále vrací konstantu 0. Prohledávání pomocí A* s touto heuristikou tak bude kopírovat chování BFS.

Je přípustné a vítané implementovat možná zrychlení, jako například testování cílových stavů již při jejich rozbalování; tam, kde to nevede ke ztrátě optimálnosti algoritmu. Na druhé straně se držte tradiční formulace algoritmů, takže při prohledávání do šířky uchovávejte – a testujte – již prozkoumané stavy a při prohledávání do hloubky naopak žádnou paměť již prozkoumaným stavům nevěnujte.

Rozhraní řešení Aby se dal pohodlně předávat a konfigurovat, je každý prohledávací algoritmus instancí třídy dědící rozhraní `SearchStrategyItf` (`search-interface.h`), očekává se tedy od něj, že bude metodou `solve(init_state)` hledat řešení z dodaného stavu. Prohledávací stav `SearchState` za tím účelem poskytuje seznam v něm proveditelných akcí (`.actions()`) a test, zda je řešením (`.isFinal()`). Prohledávací akce `SearchAction` má jedinou relevantní metodu, a sice `.execute(state)`, která vrací výsledný stav[2]. Řešením je potom posloupnost akcí.

Náležitosti řešení Implementovat budete v jazyce C++17, odevzdávat budete jediný soubor `sui-solution.cc`, v němž budou implementovány veškeré potřebné metody a funkce. Definice očekávaných metod naleznete v hlavičkovém souboru `search-strategies.h`. Ukázka prohledávacího algoritmu a heuristiky je v souboru `strategies-provided.cc`. Je zapovězeno obcházení typového systému[3], spouštění dalších vláken/procesů a práce se souborovým systémem. Není dovoleno používat jiné než standardní knihovny, cizí implementace používejte nanejvýše pro volnou inspiraci, raději vůbec. Obecně se držte rozhraní dodaného v souboru `search-interface.h`, v případě nejistoty se zeptejte.

Paměťová omezení Jak se v projektu sami přesvědčíte, prohledávání do šířky, ať už s heuristikou nebo bez, je paměťově velmi náročná záležitost. Prohledávací algoritmy jsou proto v tomto projektu omezeny hlídačem (`MemWatcher`), který běží v druhém vlákně. Pokud rezidentní pamět procesu přesáhne uživatelem zadanou hodnotu (`--mem-limit NB_BYTES`), proces bude násilně ukončen. Proto je nezbytné, aby při prohledávání jednoho problému nedocházelo k úniku paměti, dbejte na její uvolňování. Zároveň je silně doporučeno, aby si Vaše řešení hlídala spotřebu paměti (využijte funkci `memusage.h:getCurrentRSS()`) a v případě blízkého přiblížení k limitu[4] vzdala pokusy o řešení a odevzdala řešení neplatné, ideálně prázdné. Pokud dojde k ukončení programu kvůli překročení paměti, bude na Vaše řešení pohlíženo, jako by nevyřešilo všechny následující instance problému. Testování bude ovšem probíhat v mnoha

[1] https://github.com/ibenes/freecell

[2] `SearchState` nelze kopírovat přiřazením, pouze konstrukcí nového. Lze je ale přiřazením přesouvat, což využijete právě při zachycení výsledku provedení akce.

[3] Tedy zejména se vyvarujte přetypování, ať už jako `*_cast<NewType>(value)` nebo, nedej bože, `(NewType) value`.

[4] Limit je prohledávacímu algoritmu poskytnut při konstrukci; 50 MB se jeví jako rozumná rezerva.

oddělených dávkách, takže *ojedinělý* pád způsobený příliš aktivně alokujícím kontejnerem nebude mít fatální dopad na Vaše hodnocení.

Doporučení pro řešení

- Straňte se ruční správy paměti.
- Připomeňte si standardní kontejnery dostupné v STL. Přemýšlejte o tom, jaké operace potřebujete provádět *rychle* a volte kontejnery úměrně tomu. Lineární vyhledávání není správnou volbou prakticky nikdy.
- Kde Vám nestačí standardní kontejnery, použijte chytré ukazatele. Nejspíš se Vám bude hodit `shared_ptr`.
- Při řešení postupujte v navrženém pořadí.
- Při vývoji se nebojte používat `--easy-mode N` s velmi nízkými hodnotami `N`. Na konzistentní řešení plně náhodných instancí FreeCellu byste potřebovali dobrou heuristiku, to není součástí letošního zadání.
- Referenční řešení se se značnou rezervou vešlo do 200 řádků. Pokud směřujete k většímu počtu, máte vertikálně dost upovídaný styl formátování kódu (nevadí) nebo něco děláte špatně.
- Pokud nedokážete najít řešení, vracejte prázdné řešení.
- Pokud chcete testovat instance `SearchState` testovat na rovnost, máte na to deklarován přátelský operátor. Jeho implementace je na Vás, linker si ji najde. Pozn.: `std::set` se na rovnost neptá.

Vyhodnocování řešení Všechny části Vašeho řešení budou vyhodnocovány na různě složitých instancích hry FreeCell, s různými paměťovými omezeními. Při vyhodnocování budou brána v patrnost omezení daná slepotou prohledávání v BFS a DFS, resp. slabiny dodané heuristiky, a není očekáváno, že se vždy podaří vyřešit všechny instance hry. Můžete předpokládat, že dostupná pamět bude vždy alespoň 1 GB.

Přibližně polovina bodů bude udělena za to, že algoritmy fungují, a druhá polovina za to, že fungují rozumně rychle.