Domácí úloha č. 2 – Přelévání vody

Specifikace úlohy

Cílem domácí úlohy přelévání vody bylo navrhnout a implementovat heuristiku pro problém přelévání vody. Tato heuristika se měla srovnat s průchodem do šířky a průchodem do hloubky. Pro jednotlivé strategie řešení je úkolem zjistit délku cesty k jednotlivým řešením a počet navštívených bodů stavového prostoru.

Nástroje řešení

K implementaci jsem využil programovací jazyk **Java** pod prostředím **NetBeans**. Všechny výpočty běželi na procesoru Intel Core 2 Duo 3.00 GHz a pod operačním systémem Microsoft Windows 7. Výsledky byly zpracovány tabulkovým procesorem Microsoft Excel.

Výsledný zdrojový kód je spouštěn ze souboru **Main.java**, zbytek kódu je přehledně rozdělen do tříd.

K měření času jsem použil funkci **System.currentTimeMillis().**

Rozbor jednotlivých variant řešení

Algoritmus průchodu do šířky (BFS)

Tímto algoritmem jsem realizoval průchod stavového prostoru metodou do šířky za využití fronty. Další stavy se (stejně jako u ostatních algoritmu) expandují tak, že nejdříve se zkusí každý z kýblů vylít, poté každý kýbl naplnit a nakonec lze kýble mezi sebou přelévat. Toto vše děláme tak dlouho, dokud máme na zásobníku nějaký stav.

Tento typ průchodu se hodí, pokud se hledaný stav nachází blízko startovního uzlu, tzn. někde nahoře stavového stromu. Jak jsme se měřením přesvědčili a rovněž díky nevhodné implementaci jednotlivých stavů byl tento průchod velice časově náročný. Rychle jsem nalezl pouze stavy, které byly na začátku stavového prostoru.

Algoritmus průchodu do hloubky (DFS)

Tento algoritmus byl v nějakých případech rychlejší jak BFS, ale ve většině případů program skončil přetečením. Proto je tento algoritmus naprosto nevhodným, protože zbytečně expanduje stavy hluboko do stavového prostoru, což nám způsobuje velké paměťové nároky.

Algoritmus s heuristikou

Algoritmus s heuristikou je implementován stejně jako BFS a DFS, akorát místo klasické fronty (nebo zásobníku) se používá prioritní fronta, abychom mohli stavy na zásobníku řadit dle jejich priority. Je to vlastně A* algoritmus s vlastním ohodnocováním stavů. Ke každému stavu je tak nutné dopočítat prioritu, což by mělo být nějaké číslo, které určuje, jak velká je šance, že se přes tento stav dostaneme k řešení.

Jednotlivé priority počítám tak, že pro každý stav spočítám jeho hodnocení. Na začátku má stav hodnocení nulové. Pokud se v aktuálním stavu objeví nějaký kyblík, který má aktuální obsah totožný s nějakým cílovým obsahem, zvednu hodnocení stavu o jedničku. Pokud je tento aktuální obsah dokonce na svém správném místě, zvednu ohodnocení o dvojku.

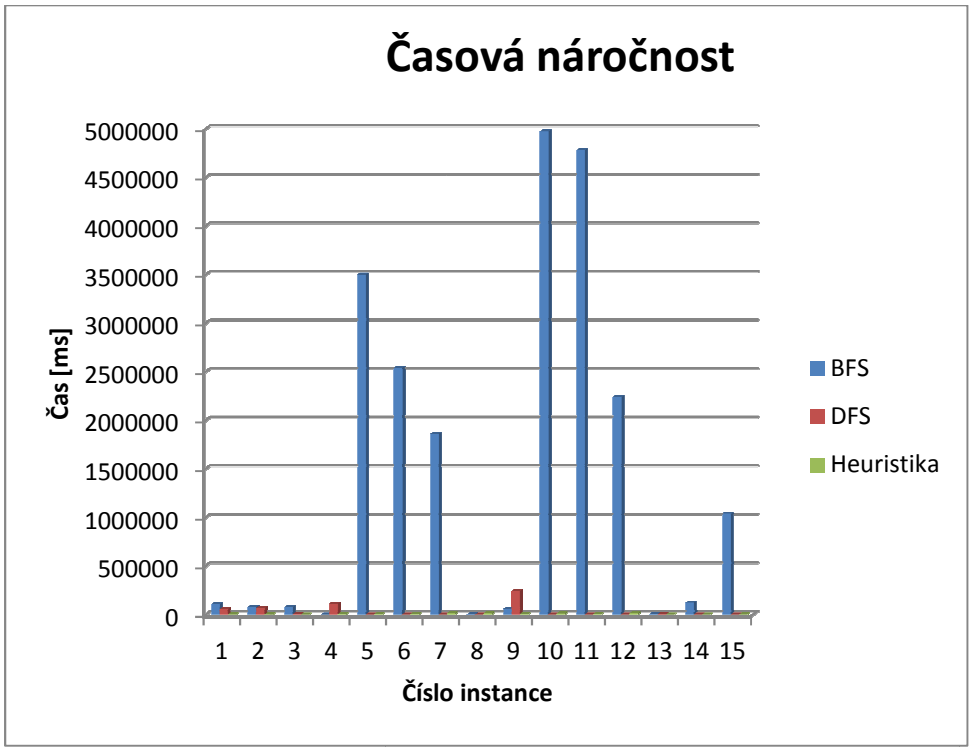
Díky této heuristice se mi povedlo dosáhnout velice nízkého času. V některých případech bylo sice BFS rychlejší, ale obecně vzato je heuristika lepším algoritmem hledání cílové konfigurace.

Naměřené výsledky

Níže výpis naměřených hodnot tabulkově:

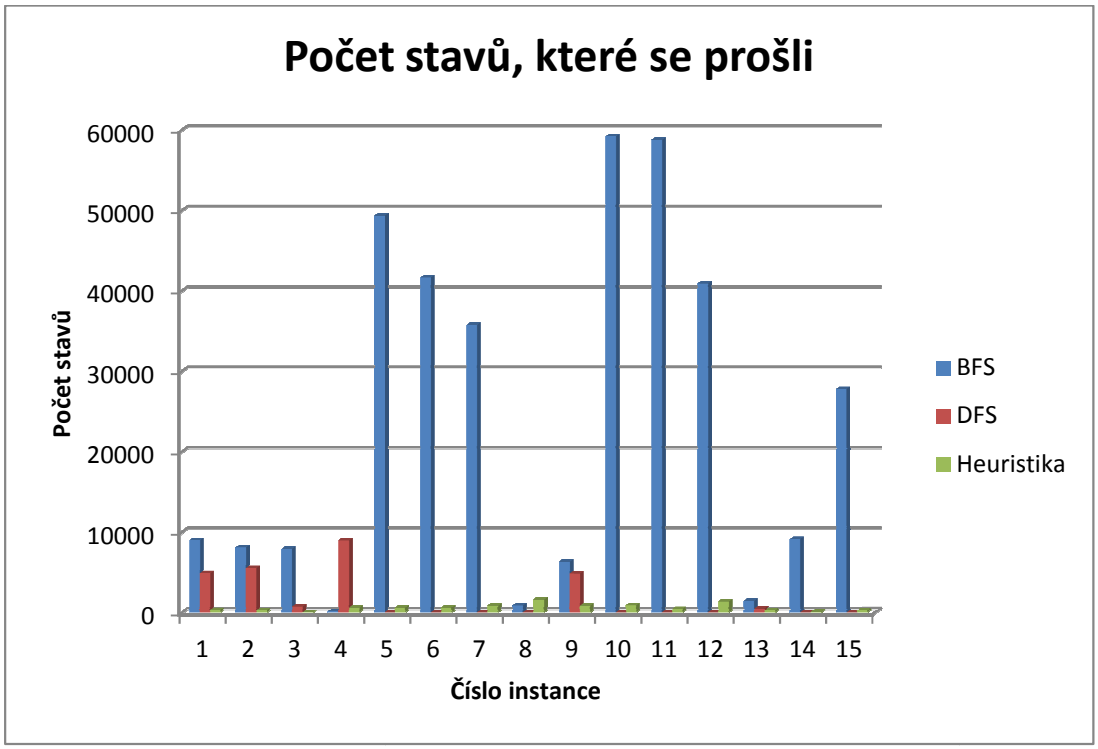
	BFS			DFS			Heuristic 1		
Č.	Hloubka	# uzlů	Čas[ms]	Hloubka	# uzlů	Čas[ms]	Hloubka	# uzlů	Čas[ms]
11	10	8992	106125	1125	4870	57563	19	295	1061
12	8	8084	80165	836	5521	68987	9	268	530
13	8	7914	78721	318	742	2400	23	35	16
14	3	170	117	5	8953	109536	11	597	2340
21	16	49350	3499577	-	-	-	42	586	2402
22	12	41669	2540323	-	-	-	52	618	6755
23	11	35750	1861169	-	-	-	38	827	12340
24	5	872	1513	-	-	-	18	1604	12090
25	7	6324	58427	4692	4848	241919	37	839	4929
31	14	59200	4975088	-	-	-	20	884	10889
32	12	58772	4781442	-	-	-	30	423	3713
33	10	40908	2240414	-	-	-	23	1377	14664
34	5	1461	4555	475	476	1758	6	264	359
35	7	9155	122913	-	-	-	20	100	156
36	9	27773	1039404	-	-	-	24	279	717

Závislost času na jednotlivých instancích



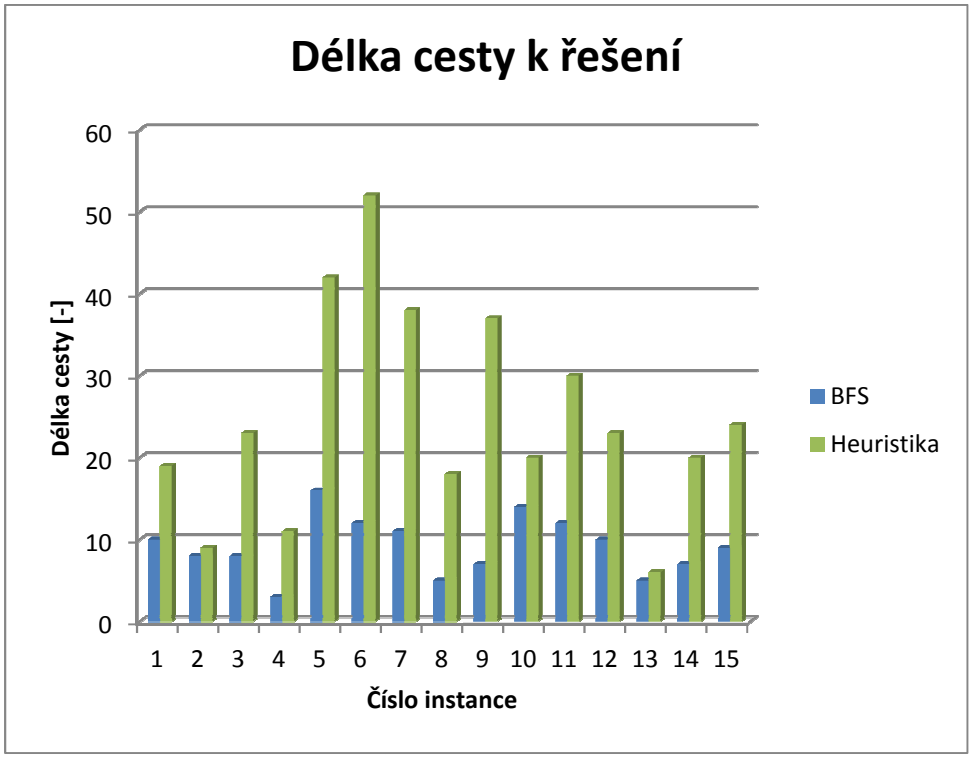
Jak je vidět z grafu, BFS je velice časově náročné. DFS je zase paměťově náročně a některé instance se vůbec nepodařilo dokončit.

Závislost počtu procházených stavů na jednotlivých instancích



DFS se u některých instancí vůbec nepodařilo naměřit, proto v grafu není vidět, ale pokud bych měl lepší implementaci programu, myslím si, že by DFS mělo úplně nejvíce expandovaných stavů ze všech algoritmů.

Délka cesty u jednotlivých instancí



Z grafu je vidět, že delších cest dosahuje heuristika, což odpovídá její strategii průchodu stromu. Nejde tak řešení rychleji, protože neprochází systematicky, ale snaží se přiblížit co nejvíce k řešení.

Stručný popis implementace

- Main.java – spustitelný soubor, kde se načítá soubor s instancemi, vytváří se vlastní výpočet, nastavuje strategie výpočtu
- Nalevna.java – kontext daného řešení, design pattern Strategy, uchovávají se zde otevřené stavy
- IAlgorithm.java – rozhraní, které implementuje každý algoritmus řešení
- BaseAlgorithm – základní algoritmus průchodu, poskytuje metodu ziskejNoveStavy, která provede expanzi a uložení na zásobník
- BfsAlgorithm, DfsAlgorithm – konkrétní strategie průchodu pro BFS a DFS
- AstarAlgorithm – průchod stavového prostoru pomocí A* algoritmu se zvolenou heuristikou
- StavyKybliku.java – objekt jednoho stavu kyblíků, který uchovává jednotlivé kyblíky
- Kyblik.java – objekt jednoho kyblíku, má atributy kapacita, aktuální objem, cílový objem
- StavyKyblikuComparator – komparátor potřebný pro řazení stavů v prioritní frontě

Závěr

Pokud bychom od programu vyžadovali řešení, které půjde nejlépe rekonstruovat, tzn. reálně si provést, i přes svoji časovou náročnost, by byl nejlepší průchod do šířky, protože dosahuje nejkratších cest do cíle.

Pokud upřednostňujeme rychlost výpočtu všech instancí, je jednoznačně nejlepší použitá heuristika. Stávající heuristika by šla určitě vylepšit, např. zahrnutím vzdálenosti od počátku. Je ale velice zajímavé pozorovat, jak mírná úprava implementace (změna z klasické fronty na prioritní a implementace komparátoru) nám pomůže dosáhnout řádově nižších časů.