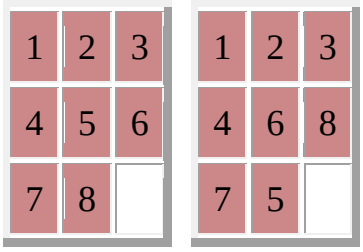
Definovanie problému 2

Našou úlohou je nájsť riešenie 8-hlavolamu. Hlavolam je zložený z 8 očíslovaných políčok a jedného prázdneho miesta. Políčka je možné presúvať hore, dole, vľavo alebo vpravo, ale len ak je tým smerom medzera. Je vždy daná nejaká východisková a nejaká cieľová pozícia a je potrebné nájsť postupnosť krokov, ktoré vedú z jednej pozície do druhej.

Príkladom môže byť nasledovná začiatočná a koncová pozícia:

Začiatok: **Koniec:**



Im zodpovedajúca postupnosť krokov je: **VPRAVO, DOLE, VĽAVO, HORE**.

Implementácia 2

Keď chceme túto úlohu riešiť algoritmami prehľadávania stavového priestoru, musíme si konkretizovať niektoré pojmy:

STAV

Stav predstavuje aktuálne rozloženie políčok. Počiatočný stav môžeme zapísať napríklad

```
((1 2 3)(4 5 6)(7 8 m))
```

alebo

```
(1 2 3 4 5 6 7 8 m)
```

Každý zápis má svoje výhody a nevýhody. Prvý umožňuje (všeobecnejšie) spracovať ľubovoľný hlavolam rozmerov m*n, druhý má jednoduchšiu realizáciu operátorov.

Vstupom algoritmov sú práve dva stavy: začiatočný a cieľový. Vstupom programu však môže byť aj ďalšia informácia, napríklad výber heuristiky.

OPERÁTORY

Operátory sú len štyri:

```
VPRAVO, DOLE, VLAVO a HORE
```

Operátor má jednoduchú úlohu – dostane nejaký stav a ak je to možné, vráti nový stav. Ak operátor na vstupný stav nie je možné použiť, výstup nie je definovaný. V konkrétnej implementácii je potrebné výstup buď vhodne

dodefinovať, alebo zabrániť volaniu nepoužiteľného operátora. **Všetky operátory pre tento problém majú rovnakú váhu.**

Príklad použitia operátora DOLE:

Vstup:

```
((1 2 3)(4 5 6)(7 8 m))
```

Výstup:

```
((1 2 3)(4 5 m)(7 8 6))
```

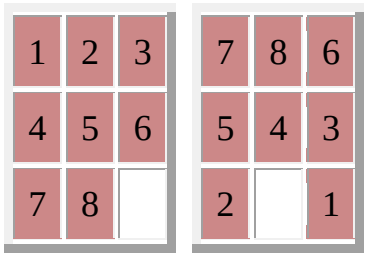
HEURISTICKÁ FUNKCIA

Niektoré z algoritmov potrebujú k svojej činnosti dodatočnú informáciu o riešenom probléme, presnejšie odhad vzdialenosti od cieľového stavu. Pre náš problém ich existuje niekoľko, môžeme použiť napríklad

1. Počet políčok, ktoré nie sú na svojom mieste
2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície
3. Kombinácia predchádzajúcich odhadov

Tieto odhady majú navyše mierne odlišné vlastnosti podľa toho, či medzi políčka počítame alebo nepočítame aj medzeru. Započítavať medzeru však nie je vhodné, lebo taká heuristika nadhodnocuje počet krokov do cieľa.

Príklad:
Heuristika č. 2, bez medzery, odhaduje vzdialenosť nasledujúcich dvoch stavov na




```
4 + 3 + 1 + 1 + 1 + 1 + 2 + 2 = 15
```

UZOL

Stav predstavuje nejaký bod v stavovom priestore. My však od algoritmov požadujeme, aby nám ukázali cestu. Preto musíme zo stavového priestoru vytvoriť graf, najlepšie priamo strom. Našťastie to nie je zložitá úloha. Stavy jednoducho nahradíme uzlami.

Čo obsahuje typický uzol?
Musí minimálne obsahovať

- **STAV** (to, čo uzol reprezentuje) a

- **ODKAZ NA PREDCHODCU** (pre nás zaujímavá hrana grafu, reprezentovaná čo najefektívnejšie).

Okrem toho môže obsahovať ďalšie informácie, ako

- **POSLEDNE POUŽITÝ OPERÁTOR**
- **PREDCHÁDZAJÚCE OPERÁTORY**
- **HĹBKA UZLA**
- **CENA PREJDENEJ CESTY**
- **ODHAD CENY CESTY DO CIEĽA**
- Iné vhodné informácie o uzle

Uzol by však nemal obsahovať údaje, ktoré sú nadbytočné a príslušný algoritmus ich nepotrebuje. Pri zložitých úlohách sa generuje veľké množstvo uzlov a každý zbytočný bajt v uzle dokáže spotrebovať množstvo pamäti a znížiť rozsah prehľadávania algoritmu. Nedostatok informácií môže zase extrémne zvýšiť časové nároky algoritmu. ***Použité údaje zdôvodnite***.

ALGORITMUS

Každé zadanie používa svoj algoritmus, ale algoritmy majú mnohé spoločné črty. Každý z nich potrebuje udržiavať informácie o uzloch, ktoré už kompletne spracoval a aj o uzloch, ktoré už vygeneroval, ale zatiaľ sa nedostali na spracovanie. Algoritmy majú tendenciu generovať množstvo stavov, ktoré už boli raz vygenerované. S týmto problémom je tiež potrebné sa vhodne vysporiadať, zvlášť u algoritmov, kde rovnaký stav neznamená rovnako dobrý uzol.

Činnosť nasledujúcich algoritmov sa dá z implementačného hľadiska opísať nasledujúcimi všeobecnými krokmi:

1. Vytvor počiatočný uzol a umiestni ho medzi vytvorené a zatiaľ nespracované uzly
2. Ak neexistuje žiadny vytvorený a zatiaľ nespracovaný uzol, skonči s neúspechom – riešenie neexistuje
3. Vyber najvhodnejší uzol z vytvorených a zatiaľ nespracovaných, označ ho aktuálny
4. Ak tento uzol predstavuje cieľový stav, skonči s úspechom – vypíš riešenie
5. Vytvor nasledovníkov aktuálneho uzla a zaraď ho medzi spracované uzly
6. Vytrieď nasledovníkov a ulož ich medzi vytvorené a zatiaľ nespracované
7. Choď na krok 2.

Uvedené kroky sú len všeobecné a pre jednotlivé algoritmy ich treba ešte vždy rôzne upravovať a optimalizovať.

Niekoľko výsledkov z prehľadávania do šírky v M*N hlavolame.

Vieme, že tento hlavolam sa môže nachádzať v ľubovoľnom stave, ktorý patrí práve do jednej z dvoch navzájom disjunktných, rovnako veľkých množín stavov. Ľubovoľný stav vyjadrujú permutácie, takže počet možných stavov, patriacich do jednej z týchto množín je polovica z faktoriálu z M*N. Toto číslo neuveriteľne rýchlo rastie, takže nie je problém zahltiť pamäť počítača a dospieť k nepoužiteľne dlhým časom výpočtu. Nasledujúce príklady riešení (prvé tri) ukazujú, k akým časom sa je možné priblížiť pri aspoň trochu slušnej optimalizácii riešenia. Časy boli určené pre AMD Turion 1,4GHz, 500MB operačnej pamäti, priamo z IDE, pri náročnej hudbe na pozadí (asi 15% výkonu – to boli časy!).

Stavy z uvedených zoznamov je tiež vhodné využiť na testovanie vlastného riešenia. (Môžete skontrolovať, či môj algoritmus fungoval správne a neexistuje kratšia cesta medzi vybranými stavmi.) Tiež je možné využiť, že už viete, v akej najväčšej hĺbke môže byť cieľový uzol.

V prostredí NetBeans, v Jazyku JAVA 1.5, bol naprogramovaný algoritmus prehľadávania celého stavového priestoru do šírky. Oproti pôvodnému, všeobecnému postupu, bolo použitých niekoľko optimalizácií:

1. Algoritmus negeneruje „spätný“ ťah, to znamená, že napríklad ak rodičovský uzol bol vytvorený zo svojho predchodcu posunom políčka doľava, nebude sa generovať jeho potomok posunom doprava. Ušetrí sa tým zbytočné vytváranie uzla a kontrola stavu, ktorý už bol spracovaný.
2. Vytvorené uzly s novými stavmi sú vkladané do frontu, aby sa dodržalo spracovanie prehľadávaním do šírky, ale zároveň sa ich stav ukladá do štruktúry HashSet, aby bola zaistená rýchla kontrola už vytvoreného stavu.
3. Existuje len jediný HashSet všetkých stavov uzlov, spracované uzly na rozdiel od vygenerovaných už nie sú vo fronte. Uzly sú však udržiavané v odkaze na rodiča, aby sme mohli na záver vypísať najdlhšiu nájdenú (optimálnu) cestu.

Výstupy pre rozmer:

- 3*2; 6! = 720
- 4*2; 8! = 40 320
- 3*3; 9! = 362 880
- 5*2; 10! = 3 628 800
- 4*3; 12! = 479 001 600

- 6*2; 12! = 479 001 600

Rozmer 5*2 bol vytvorený takmer o 11 rokov neskôr, na notebooku s procesorom core i7 a 16GB pamäti. Desaťnásobný počet uzlov pre tento rozmer zvládol za približne rovnaký čas ako pôvodný notebook pre rozmer 3x3. Rozmer 4x3 je ale zatiaľ pre Javu priveľký. JVM je 32 bitový, zvláda heap do veľkosti 4GB, kde v pôvodnej verzii dokázal uložiť 12 miliónov uzlov a v novej verzii 25 miliónov uzlov. Rozmer 4x3 však obsahuje 240 miliónov uzlov. Na dosiahnutie hĺbky 29 už Java potrebovala asi tri minúty a viac ako polovicu z toho času bežal Garbage Collector.

Na riešenie hlavolamu s dvanástimi políčkami boli použité ďalšie optimalizácie:

1. Stav hlavolamu je udržiavaný v jednej premennej typu long (64 bitov, použité sú 4 bity na číslo, takže skutočne využitých je len 48 bitov). Odkaz na predchodcu je Int a ešte je tam po Byte na hĺbku, pozíciu nuly a smer, takže veľkosť celého uzla je 16 Byte. (Java však aj tak potrebovala okolo 140 Byte na jeden uzol.)
2. Hashtable obsahuje len uvedený stav, teda hodnoty veľkosti 8 Byte. Pôvodný čas pre rozmer 5x2 bol v jazyku C dve sekundy, ad-hoc optimalizáciou hash funkcie a posunu sa mi podarilo čas stiahnuť na 1,5 sekundy. Výraznejšie skrátenie času prišlo až s kompiláciou na 64-bitovú aplikáciu, lebo väčšina práce sa vykonáva so 64-bitovými číslami.
3. V jazyku C sú uzly ukladané do jedného pamäťového bloku (pre 240 miliónov uzlov vo veľkosti 3,84 GB), ktorý slúži zároveň ako front uzlov. Hashtable je rovnakej veľkosti, takže dochádza maximálne k 50% zaplneniu.