Meno: Martin Schnūrer
Predmet: Umelá Inteligencia
Problém: D (puzzle pažravý-GREEDY algoritmus)

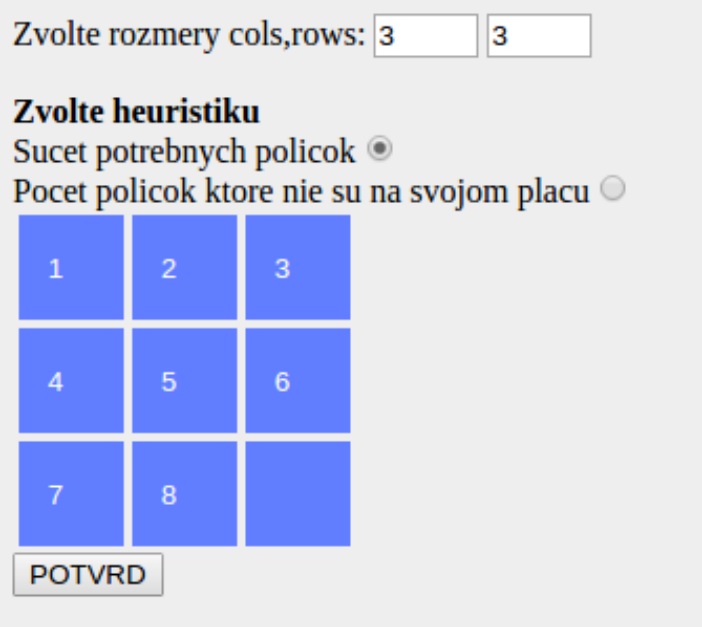
Na riešenie som použil dátove štruktúry:

- Prioritný front
- Hash Tabuľka

Reprezentácia Puzzle:
8 puzzle – pole – [1,2,3,4,5,6,7,8,0]
15 puzzle – pole – [1,2,3,...,14,15,0]

- medzera je reprezentovaná ako „m“ (medzera) .
- je možné použiť aj iné možnosti ako N*N rozmerné , napríklad 5*3 rozmer

Návod na použitie



Najprv zadajte rozmer m*n , potom vyplňte políčka a nechajte jedno políčko prázne.

Program bude fungovať správne iba za podmienky, že bude <u>práve jedno</u> políčko voľné !

VÝPIS (manual)

ťah DOWN znamená, že políčko, ktoré je nad blank políčkom, mám dať dole.
Ťah RIGHT znamená, že políčko, ktoré je naľavo od blank políčka, mám dať doprava.

Dôležité funkcie v programe

- **getInvCount** a **isSolvable**: funkcie zisťujúce, či má dané rozloženie puzzle možné riešenie, ak funkcia isSolvable vráti **false**, puzzle nemá riešenie
- **changeHeuristic**,**recreate**,**changeValue** – funkcie interagujúce s grafickým rozhraním, ak nastane zmena zo strany používateľa – resp. zmení sa voľba, zavolajú sa tieto funkcie. Menia globálne premenné.
- **Puzzle** – Objekt, uchováva svoje pole reprezentujúce stav puzzle. Ďalej si v sebe uchováva pozíciu prázdneho políčka; svoj súčet (heuristiku) ; odkaz na predchodcu a vykonaný krok, ktorý viedol k vytvoreniu daného rozloženia puzzle.
- **InsertToTable –** funkcia vloženia prvku (šachovnice) do hash tabulky.
  Vráti 0 ak sa prvok v tabuľke už nachádza.
  Vráti 1 ak prvok nebol v tabuľke, a bol úspešne vložený do tabuľky
  Vráti 2 ak sa tabuľka premazávala – premazávali sa iba prvky, ktoré sa mali vymazať (neskôr vysvetlené, ktoré prvky sa maju vymazať)
- **porovnajPolia** – porovna 2 polia, vrati 1 ak sa rovnaju, inak vrati 0 (nerovnaju sa)
- **getValid** – vrati pole stringov s valid ťahmi, napriklad ['down','left']
- down,up,left,right – funkcie na vytvorenie nových puzzle možností, podľa toho, ktorým ťahom sa vyberieme
- **findSolution** – hlavná funkcia hľadajúca správne riešenie
- **vypisPredchodcov** – vypis všetkych nodes a ťahov po vyhľadaní správného riešenia
- **startSimulation** – táto funkcia sa spustí ak je stlačený button.

Popis fungovania

Do funkcie findSolution vstupujú polia – začiatočný a koncový stav. Začiatočný stav sa uloží do frontu neprehľadaných stavov. Ďalším krokom je postupné vyberanie neprehľadaných stavov z frontu. Ak je front prázdny, tak sme neúspešný (chyba) . Ak sa z frontu vybral náš koncový stav, vyhlásime úspech – našli sme riešenie pre koncový stav. Pre každý vybraný neprehľadaný stav vygenerujeme valid ťahy a z nich následne vytvoríme dalšie možné rozloženia puzzle. Hneď pri vytvorení puzzle sa do objektu puzzle vypočíta heuristika – podľa voľby užívateľa. V objekte sa uchováva číslo heuristiky. Novo vytvorené stavy sa pokúsime uložiť do hash

tabuľky.Pri úspešnom vložení nové puzzle rozloženie uložíme do frontu neprehľadaných stavov. Pri neúspešnom vložení do hash tabuľky – to znamená, že tento stav sme už prehľadali, tak tento stav zahodíme. Na začiatok frontu pôjdu stavy, ktoré majú heuristiku menšiu ako prvý člen frontu. Inak pôjdu až na koniec. Po vygenerovaní opakujeme výber z frontu a cyklus sa opakuje.

Na to aby sme negenerovali stavy, ktoré už predtým boli vygenerované, sme si ich zapamätávali v hash tabuľke.

Pre väčšie problémy – viac ako 3x3 – môže nastať rýchle zaplnenie tabuľky.

Preto som použil pri určitom zaplnení tabuľky jej premazanie, pričom som si pamätal najlepšiu dosiahnutú cestu – s najnižšou heuristikou. Po premazaní tabuľky som do nej znova vložil stavy, ktoré určovali najlepšiu nájdenu cestu – sú to vlastne potomkovia (alebo predchodcovia) najlepšieho stavu.

Po premazaní a pridaní najlepšej cesty pokračuje simulácia od najlepšieho stavu.

Ak by za po ďalšom zaplnení nenašiel lepší stav s lepšou heuristikou od posledne lepšieho stavu, tak program uviazne v lokálnom minime – tento problem je riešený tak, že po premazaní tabuľky sa heuristika najlepšieho stavu (s najmenšou heuristikou) inkrementuje. Týmto sa zabezpečí, že program neuviazne v lokálnom minime, ale bude hľadať inú cestu – podobne dobrú ako bola predtým – týmto zabezpečíme odchod z lokálneho minima.

Testovanie (špeciálnych prípadov)

prípad keď začiatočný stav == koncový

2x2 šachovnicu

3x3 šachovnicu (10 rôznych prípadov)

3x3 šachovnicu

4x4 šachovnicu (5 rôznych prípadov)

MxN rôzne stavy

NOT VALID ŠACHOVNICA -> je prítomná implementácia zisťovania. Funguje iba pre N*N rozmery, kde N>2 . Nie je zahrnutá do bežiacého kódu, pretože pri m*n rozmeroch odpovedá, že nie je valid, pričom valid rozloženie to je.

Čas

1x1	Joke
2x2	0-1 ms
3x3	5-10 ms
2x5	3-8 ms
3x4	20-500 ms
4x4	300 ms – 10000 ms
5x5	20 – 100 s

Porovnanie heuristík

heurstika č. 1 – súčet potrebných prejdených pozícií jednotlivých políčok aby sa dostali na svoje miesto – Manhattan

heuristika č.2 – počet políčok, ktoré nie sú na svojích miestach

Zhodnotenie

Podľa môjho názoru vyhráva jednoznačne heuristika č.1 Manhattan.

Odôvodnenie

Pri Puzzle 2x2,3x3 si rozdiel medzi heuristikami takmer nevšimneme. Výrazný rozdiel nastáva pri väčších rozmeroch puzzle, kde je rýchlejšia heuristika Manhattan. Je to spôsobené tým, že heuristika pri počte políčok, ktoré nie su na svojom mieste sa môže ľahko mýliť. Keď je heuristika napr. 2 (2 políčka nie su na svojom mieste) , môže sa nám zdať že už sme takmer v cieli. Ale to nie je pravda, pretože môžu byť vymenené políčka na opačných koncoch a to nám značí, že k správnemu vyriešeniu sme ešte naozaj ďaleko. Napr. 8 na mieste 1 , a 1 na mieste 8.

Preto je možné, že pri väčších rozmeroch heuristika č.2 ľahko uviazne v lokálnom minime. Z tohto dôvodu vidím heuristiku č.1 ako lepšie a optimálnejšie riešenie pre puzzle.