Riešenie sokobana evolučným prístupom

Matej Pavla

2. prezentácia

Stav projektu

- Podarilo sa mi implementovať základné časti evolučného algoritmu
  1. Vytvorenie generácie
  2. Výpočet fitness jedincov
  3. Vytvorenie novej generácie z predošlej
     - Ruleta
     - Elitárstvo
     - Mutácie
- Vykresľovanie pomocou Java Slick2D

1. Vytvorenie novej generácie

- **EvolutionSokoSolver.java:531-generateBrandNewGeneration()**
  - Vytvoríme GENERATION_SIZE jedincov
  - Každý jedinec je reprezentovaný reťazcom min. dĺžky MIN_INDIVIDUM_SIZE a maximálnej MAX_INDIVIDUM_SIZE
  - Bity reťazca sú čísla 0-3, každé predstavuje jeden smer pohybu figúrky v mapke

2. Výpočet fitness

- Fitness počítame najprv ako penalty:
  - Za každý krok získa jedinec STEP_PENALTY bodov ako penalty
  - Ďalej je použitá heurestika: Dávame penalty za to, ako ďaleko je vzdialená každá krabica od najbližšieho cieľa manhattanskou vzdialenosťou
    - Za každé políčko MANHATTAN_PENALTY bodov ako penalty
- Z penalty urobíme fitness, tak aby mal najlepší jedinec najväčšiu fitness (len prevrátime hodnoty penalty)
- Toto všetko sa jede paralelne, funkcia EvolutionSokoSolver.java:182:computeFitnessOfGeneration() rozdelí výpočet do THREADS_NUMBER paralelných vlákien
- Každá fitness sa zvlásť počíta pre každého jedinca vo funkcii EvolutionSokoSolver.java:210: individumFitnessCompute()
  - Urobí si´vlastnú kópiu mapy
  - Pohybuje sa v nej podľa reprezentácie jedinca:
    - 0 – pohyb nahor prípadne tlačenie debny (ak sa dá)
    - 1 – pohyb vľavo prípadne tlačenie debny (ak sa dá)
    - 2 – pohyb nahor prípadne tlačenie debny (ak sa dá)
    - 3 – pohyb vľavo prípadne tlačenie debny (ak sa dá)
- Z výslednej mapy sa podľa počtu krokov a pozície krabíc voči dieram na konci vypočítaju penalty -> fitness

3. Vytvorenie novej generácie

- Deje sa vo funkcii EvolutionSokoSolver.java:26: createNextGeneration()
  - Vytvorí novú generáciu zo starej:
  - CROSSOVER_COUNT jedincov sa vytvorí podľa rulety (jedinec s najväčším fitness má najväčšiu šancu sa krížiť)
    - Vo funkcii EvolutionSokoSolver.java:85:doRoulleteCrossover() nájdeme dvoch jedincov
    - Z prvého zobereme prvých náhodný počet bitov, z druhého náhodný počet posledných bitov
    - Tieto spojíme a vznikne nový jedinec
  - Na týchto jedincov aplikujeme mutácie
    - Max MUTATION_COUNT jedincov bude mať náhodne zmutovaných max MUTATION_BITS bitov (bit sa náhodne zmení)
  - ELITARISM_COUNT jedincov prežije nezmenených do ďalšej generácie
  - Ostatný jedinci pribudnú úplne nový (rovnakým náhodným spôsobom ako tomu bolo v prvej generácii)

4. Mapy a grafické rozhranie

- Mapy som stiahol zo stránky http://users.bentonrea.com/~sasquatch/sokoban/
- Vytvoril som si vlastný parser: LevelDraw.java:49: readMapFromFileById()
    - - načítava súbor mapFilePath
- Načítanú mapu potom vykreslím s pomocou Slick2D vo funkcii LevelDraw.java:76: drawLvlAt()
- Do jednoduchého grafického rozhrania vykresľujem:
  - Načítanú mapu (graficky)
  - Najlepšie riešenie podľa fitness (graficky)
  - Číslo, max a priemerný fitness súčasnej generácie (zatiaľ nápis)
- V metóde SokobanEvo.java:65:update() prebieha výpočet každej generácie, po ktorej sa zavolá metóda render, ktorá podľa výsledkou aktualizuje grafické rozhranie

5. Vyhodnotenie a pokračovanie

- Zatiaľ som implementoval najjednoduchší prístup k evolučnému riešeniu sokobana
  - Jednoduchá heurestika (manhattanské vzdialenosti do najbližšieho cieľa)
  - Tým pádom jednoduchý výpočet fitness
- Na čo som narazil
  - Algoritmus sa po pár riešeniach „zasekne“, tzn. jedinci sa prestanú zlepšovať. Vykazujú vždy rovnaké chovanie a to zasekávanie krabíc v rohoch z ktorých sa už nedá pohnúť
- Návrh zlepšenia algoritmu
  - Do ďalšej prezentácie by som sa chcel sústrediť na tunenie výpočtu fitness:
    - Kvôli vyššie opísanému problému je potrebné zaviesť vysoké penalty za to, keď krabica skončí v mieste odkiaľ sa nedá pohnúť
      - Zvýši sa čas výpočtu fitness
    - Možno aj zmeniť heurestiku (ak by nepomohlo zavedenie penalty za deadlock)
    - Popremýšľať nad pohybom agenta
      - Ak sa nemôže pohnúť smerom ktorý udáva reťazec, pohol by sa opačným smerom?
      - Zostaneme na základnom pohybe po mape alebo budeme evolučne riešiť iba pohyb krabíc (pathfinding by sme riešili napríklad A*, o zvyšok by sa staral evolučný algoritmus...)
      - Toto riešenie by pravdepodobne stálo ale veľa času, preto ho nechávam ako poslednú alternatívu ak hore uvedené nebudú fungovať