Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava 4

Bc. Peter Bartalos

Simulované žíhanie pre riešenie pohybu koňa po šachovnici

Evolučné algoritmy
Case study

Odbor: Softvérové inžinierstvo
Ročník: 1
Semester: letný
Jún, 2006

Obsah

1 Úvod **2**

2 Zadanie **2**

3 Teoretický úvod **2**

4 Riešenie pohybu koňa po šachovnici **4**

4.1 Metóda riešenia 4

4.2 Výsledky 6

5 Implementácia **9**

Použitá literatúra **11**

1 Úvod

Dokument obsahuje opis riešenia zadania *Použite simulované žíhanie pre riešenie pohybu koňa na šachovnici*, vypracovaného v rámci skúšky z predmetu *Evolučné algoritmy*[1]. Členenie dokumentu je nasledovné. V prvej kapitole sa nachádza teoretický úvod do simulovaného žíhania, vhodný pre tých, ktorý o tejto problematike nemajú dostatočné znalosti (zdroj textu - [1]). V druhej kapitole je opísaná metóda použitá pri riešení a dosiahnuté výsledky. V poslednej kapitole sa nachádza opis softvérovej implementácie, použitej pri riešení problému.

2 Zadanie

Použite simulované žíhanie pre riešenie pohybu koňa na šachovnici, kde zo zadaného poľa máte prebehnúť všetky ostatné poľa, pričom na každé pole vstúpite iba raz. Ako druhú časť úlohy riešte rovnaký problém s podmienkou, že sa kôň musí vrátiť naspäť. Pri riešení použite perturbáciu (mutáciu) toho druhu, že počiatok postupnosti polí do náhodne vybraného počtu polí skopírujete do nového riešenia a zvyšok vygenerujete. Úloha sa dá riešiť aj spätným prehľadávaním, ale len pre malé prípady.

3 Teoretický úvod

Metóda simulovaného žíhania [1, 2, 3, 4] patrí medzi stochastické optimalizačné algoritmy. Svoj základ má vo fyzike, na rozdiel od iných stochastických optimalizačných algoritmov, ktoré majú svoj základ väčšinou v biológii. Algoritmus simulovaného žíhania je založený na analógii medzi žíhaním tuhých telies a optimalizačným problémom.

Počiatkom 80-tych rokov Kirkpatrick, Gelatt a Vecchi a nezávisle Černý dostali geniálny nápad, že problém hľadania globálneho minima môže byť realizovaný podobným spôsobom ako žíhanie tuhého telesa.

Na obrázku 1 je znázornená fyzikálna realizácia simulovaného žíhania. Teleso sa vloží do pece (ľavý blok), ktorá je vyhriata na vysokú teplotu T_{max}. Teplota sa programovacím zariadením (pravý blok) pomaly znižuje na teplotu T_{min}. Týmto spôsobom sa odstránia štruktúrne defekty vyskytujúce sa v telese.

Predpokladajme, že proces ochladzovania je dostatočne pomalý, potom pre každú teplotu T žíhané teleso je v tepelnej rovnováhe, ktorá je opísaná Boltzmanovským rozdelením pravdepodobnosti toho, že pri teplote T je teleso v stave i s energiou E_i.

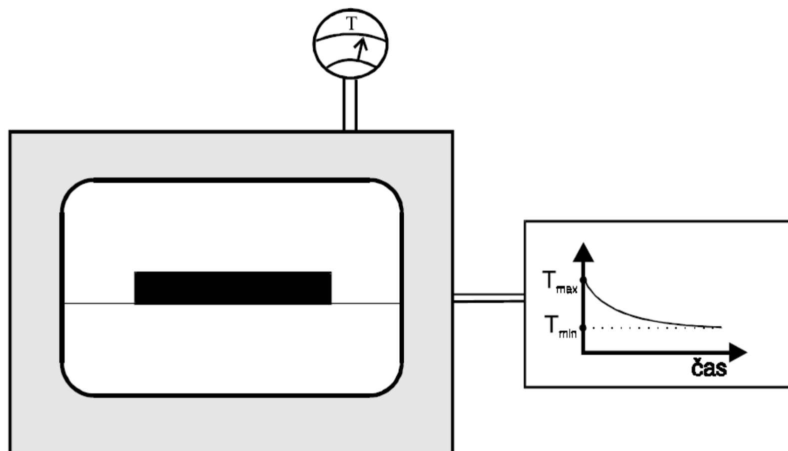
$$w_T(E_i) = \frac{1}{Q(T)} \exp(-\frac{E_i}{k \cdot T})$$

kde k je Botzmanova konštanta a $Q(T)$ je normalizačný faktor nazývaný *partičná funkcia*

$$Q(T) = \sum_i \exp(-\frac{E_i}{k \cdot T})$$

kde sumácia obsahuje všetky stavy i telesa.

[1] http://www2.fiit.stuba.sk/ pospichal/prednaskaSTU06.htm



Obrázok 1: Fyzikálna realizácia žíhania

Nech teplota T klesá, potom Boltzmanovská distribúcia uprednostňuje stavy s nižšou energiou. V prípade, že teplota sa blíži nule, stav s minimálnou energiou má nenulovú (jednotkovú) pravdepodobnosť.

Ak je ochladzovanie telesa príliš rýchle (telesu nie je umožnené získať tepelnú rovnováhu pre každú teplotu), defekty v telese môžu zamrznúť za vzniku nestabilných štruktúr, ktoré sú vzdialené od mriežkovej štruktúry s najnižšou energiou.

Metropolis a spol. navrhli Monte Carlo metódu, ktorá simuluje evolúciu systému tak, že generuje postupnosť stavov systému nasledujúcim spôsobom.

Nech je daný aktuálny stav systému (určený polohou častíc telesa), potom malá náhodná porucha je generovaná tak, že častice sú jemne posunuté. Ak rozdiel $\Delta E = E_{perturbed-E_current}$ medzi porušeným stavom a aktuálnym stavom je negatívny ($E_{perturbed<E_current}$), potom proces pokračuje s novým porušeným stavom. V opačnom prípade, ak $E_{perturbed>E_current}$, pravdepodobnosť porušenia stavu

$$Pr(perturbed \leftarrow current) = \min(1, \exp(\Delta E/kT))$$

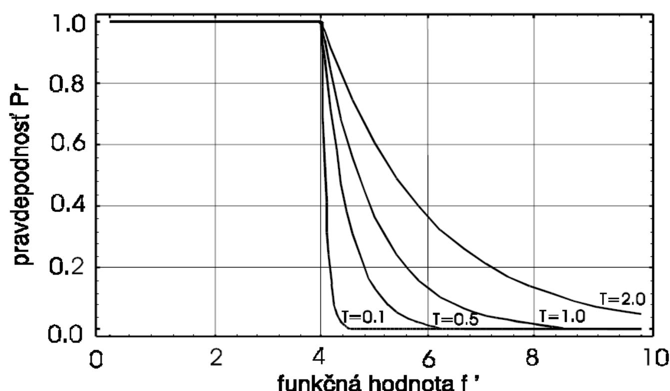
Toto pravidlo akceptovania porušeného stavu sa nazýva *Metropolisovo kritérium.*

Na obrázku 2 je znázornený priebeh Metropolisovho kritéria

$$Pr = \min(1, \exp(-(f' - f)/T))$$

pre rôzne teploty, kde f je fixná funkčná hodnota ($f = 4$) a f' je nezávislá premenná braná z intervalu $[0, 10]$. Pre klesajúce hodnoty teploty T a pre $f' > f$, pravdepodobnosť $Pr \rightarrow 0$ ak $T \rightarrow 0$.

K tomu, aby sme formalizovali Metropolisov algoritmus zavedieme nasledujúci formalizmus. Nech je stav systému určený stavovou premennou x (vo všeobecnosti vektor obsahujúci mnoho stavových premenných) a energiou $f(x)$. Proces poruchy stavu x na iný stav x' je formálne reprezentovaný operátorom, $x' = O_{perturb(x)}$. Stochastický charaker tohto operátora spočíva v náhodnej zmene stavu x na stav x'.



Obrázok 2: Priebeh Metropolisovho kritéria

4 Riešenie pohybu koňa po šachovnici

Táto kapitola je venovaná somotnému riešeniu zadaného problému. V prvej podkapitole obsahuje podrobný opis metódy, ktorou bol problém riešený. Druhá podkapitola sa venuje dosiahnutým výsledkom.

4.1 Metóda riešenia

Definovaný problém sme riešili v súlade so zadaním pomocou simulovaného žíhania. Pri riešení sme sa snažili využiť aj existujúce metódy a aj nami navrhnuté postupy. Hlavnými zdrojmi informácií pre nás boli [1, 4]. V nasledujúcej časti sa budeme zaoberať presným opisom jednotlivých častí riešenia.

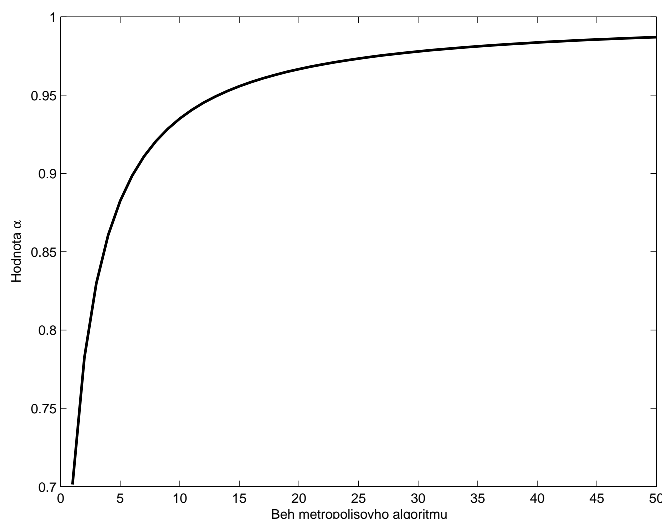
Stav systému je reprezentovaný stavovým vektorom. Ten pozostáva z určitého počtu prvkov, pričom každý reprezentuje určitú postupnosť krokov pohybu koňa po šachovnici.

Na začiatku procesu žíhania, je vygenerovný stav systému tak, že stavový vektor je naplnený generovanými postupnosťami. Generovanie spočíva v postupnom náhodnom generovámí krokov. Každý pridaný krok je pritom validovaný. Validácia spočíva v testovaní, či sme sa daným krokom nedostali mimo hracieho poľa, alebo na už prejdené políčko. V prípade zlyhania validácie, sa generovanie ukončí.

Perturbácia postupnosti spočíva v skopírovaní postupnosti krokov do určitej časti a následnom dogenerovaní postupnosti. Jednou modifikáciou, ktorú sme skúšali, bola úprava pozície po ktorú sa kopíruje. Zmena spočívala v tom, že sme určili hranicu po ktorú sa postupnosť skopíruje. Miesto od ktorého sa náhodne dogenerovávala postupnosť bola vyberaná len z pozícií nad touto hranicou. Skušali sme túto hranicu voliť ako 30%, 50% a 70%. Čim vyžšie bola hranica definovaná, tým rýchlejšie sme získávali lepšie riešenia. Samozrejme bola vyžšia pravdepodobnosť, že získané riešenie predstavuje lokálne optimum a znižovala sa šanca dosiahnuť požadované globálne optimum.

Pri znižovaní teploty sme použili multiplikatívny prístup. Teplota sa pri každom spustení metropolisovho algoritmu upravuje pravidlom $T = \alpha \cdot T$, kde $0 \ll \alpha < 1$. Priebeh teploty pri žíhaní je charakterizovaný rýchlejším ochladzovaním na začiatku procesu, ktoré sa následne spomaluje.

Pre zabezpečenie tohoto priehebu sme parameter α určovali dvoma spôsobmi. V prvom prípade bol parameter α definovaný ako konštanta. Pri použití konštantnej hodnoty parametra α tento priebeh nemusí byť dostatočne efektívny. Tieto nedostatky sme sa pokúsili riešiť dynamickým nastavovaním parametra α pred každým spustením metropolisovho algoritmu. V [4] sa hovorí o spôsobe, kedy sa proces začína pri veľmi vysokej teplote, následne je prudko ochladený a potom prebieha proces ochladzovania pozvolne. Tento priebeh sme sa snažili vytvoriť spomínaným dynamickým nastavovaním hodnoty α. Jeho hodnoty boli určované podľa funkcie zobrazenej na obrázku 3.



Obrázok 3: Priebeh parametra α

Použitie opísaného prístupu dynamického nastavovania hotnoty α sa nám ukázal byť prínosný. Výhodou bolo zníženie rizika zlého nastavenia konštantnej hodnoty tohto parametra.

Funkciu, ktorou sme ohodnocovali stavový vektor (fitnes), sme konštruovali dvoma spôsobmi. Prvý spôsob bol zameraný na najlepšiu postupnosť krokov spomedzi všetkých postupností v stavovom vektore. V druhom prípade boli zahrnuté všetky postupnosti a výsledná fitnes bola založená na priemere zo všetkých postupností. Samostné ohodnotenie postupností bolo založené na pomere počtu zatial dosiahnutých krokov voči počtu krokov potrebných na prejdenie celého poľa. Samozrejme tento prístup je vhodný len na riešenie prvej časti zadania, teda pre prípad, keď sa kôň nemusí vrátiť na východiskovú pozíciu. Pre druhú časť zadania, kde sa mal kôň vrátiť na pôvodnú pozíciu, bola fitnes upravená tak, aby zahŕňala aj vzdialenosť počiatočnej pozície od pozície dosiahnutej po vykonaní nájdenej postupnosti krokov. Pri použití fitnes, ktorá pracovala s priemerom sme nedosiahli dobré výsledky. Všetky uvedené výsledky sa preto budú vzťahovať na prípad, kedy fitnes závisela len od najlepšej postupnosti v stavovom vektore.

Fitnes pre prvú časť zadania bola teda definovaná nasledovne:

$$f(x) = 1 - \max_{s \in U(x)} \left(\frac{n_{reached}}{n_{total}} \right),$$

kde x je stavový vektor, $U(x)$ je množina všetkých postupností v stavovom vektore, $n_{reached}$ je počet dosiahnutých krokov v danej postupnosti z $U(x)$, n_{total} je počet krokov potrebných na prejdenie celého hracieho poľa.

Pre druhú časť zadania bola upravená nasledovne:

$$f(x) = 1 - \max_{s \in U(x)} \left(\frac{n_{reached} + diff_{reached}}{n_{total} + diff_{max}} \right),$$

kde $diff_{reached}$ je vzdialenosť medzi počiatočnou a koncovou pozíciou koňa. Vzdialenosť sa počíta ako súčet absolútnych hodnôt rozdielov horizontálnych a vertikálnych súradníc počiatočnej a koncovej pozície koňa. $diff_{max}$ predstavuje maximálnu možnú vzdialenosť medzi počiatočnou a koncovou pozíciou koňa, počítanú ako dvojnásobok veľkosti hracieho poľa bez jednej.

Proces žíhania je navrhnutý tak, aby minimalizoval ohodnocovaciu funkciu. Z toho dôvodu sa vo funkcii nachádza odčítavanie od jednej. Vyplýva z toho, že čím nižšie ohodnotenie, tým lepšia postupnosť.

Kritickou časťou riešenia problému pomocou simulovaného žíhania je spomínané nastavenie parametrov algoritmu. Ide o nastavenie začiatočnej teploty T_{max}, minimálnej teploty T_{min} a počtu aplikácií metropolisovho algoritmu k_{max}.

Spočiatku sme tieto parametre nastavovali viacmenej náhodne. Po získaní poznatkov o chovaní sa procesu pri rôznych hodnotách týchto parametrov sme sa ich snažili nastaviť na optimálne hodnoty, ktoré prinášali najlepšie výsledky. Hodnotu T_{max} sme sa snažili nastaviť tak, aby približne polovica porušených stavov bola akceptovaná metropolisovým algoritmom. Pri dynamickom nastavovaní parametra α sme volili hodnotu T_{max} vyžšiu. Rýchle ochladzovanie však spôsobilo, že teplota rýchlo dosiahla hodnotu pri ktorej bolo rozhodovanie o akceptovaní stavov blízko optimálne. Keďže našim cieľom bolo získať presne definovanú postupnosť krokov, ktorú vieme overiť, tak sme hodnotu parametra T_{min} zvolili nulovú. Ukončenie procesu bolo definované nájdením požadovanej postupnosti. Počet aplikácií metropolisovho algoritmu k_{max} sme nastavovali na základe výsledkov pri jednotlivých hodnotách.

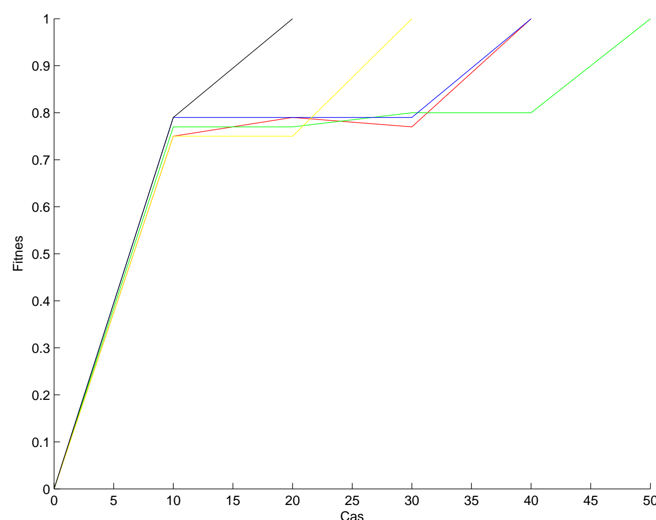
4.2 Výsledky

V tejto kapitole sa budeme venovať vyhodnoteniu výsledkov, ktoré sme dosiahli pri riešení problému. Výsledky, ktoré budú prezentované sme dosiahli pri nastavení počiatočnej pozície koňa do ľavého dolného okraja hracej plochy.

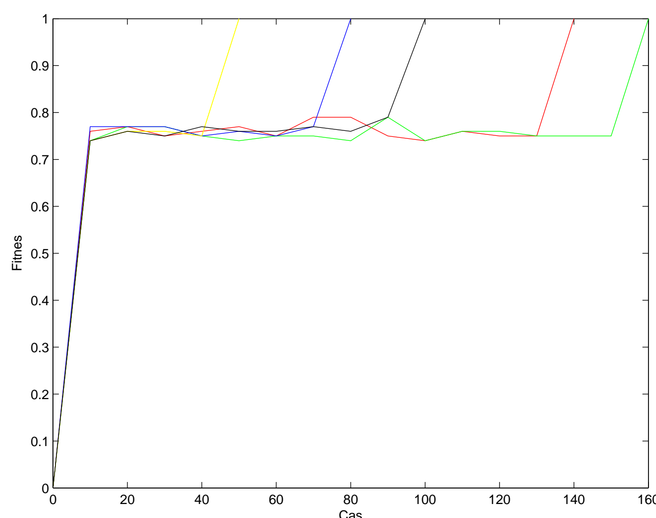
Počas testovania procesu žíhania sme odksúšali rôzne kombinácie parametrov algoritmu. Údaje, ktoré budeme ďalej uvádzať boli získané pri nastaveniach, ktoré sme považovali za najvhodnejšie.

V prvej časti zadania bolo treba riešiť pohyb koňa po šachovnici tak, že po prejdení všetkých polí sa kôň nemusí dostať na východiskovú pozíciu. Tento prípad predstavoval omnoho jednoduchší problém ako situácia, keď sa mal kôň dostať na východiskovú pozíciu. Riešenie sme získali po pomerne krátkom čase behu programu. Na obrázku 4 vidíte časový priebeh fitnes pre tento prípad počas niekoľkých behov procesu (v grafoch je fitnes pre lepšiu prehľadnosť znázornená ako pomer dosiahnutého výsledku voči finálnemu, žiadanému výsledku). Na horizontálnej osi je zobrazený čas behu programu uvedený v sekundách. Samozrejme táto časová informácia nie je smerodajná v absolútnej miere, keďže doba vykonávania programu je silne závislá od hardvérovej konfigurácie počítača na ktorom program beží a celkových podmienkach prostredia operačného systému. Zaujímavé je porovnávanie s ďalšími grafmi, ktoré uvedieme. Pre behy zobrazené na obrázku 4 obsahoval stavový vektor 10 postupností. Z grafu je vidieť, že riešenie bolo nájdené veľmi rýchlo. Zaujímavé je porovnanie s údajmi z grafu z obrázka 5. Pre tieto behy obsahoval

stavový vektor 500 postupností. Z porovnania grafov je vidieť, že v druhom prípade trvalo nájdenie riešenia dlhšie, pritom stavový vektor obsahoval 50-krát viac postupností, teda omnoho viac potenciálnych kandidátov na riešenie. Problém však spôsobovalo to, že pre dlhší stavový vektor trvá aj výpočet úmerne dlhšie. Proces simulovaného žíhania je preto tiež pomalší, čím sa odďaluje jeho efektívny účinok.



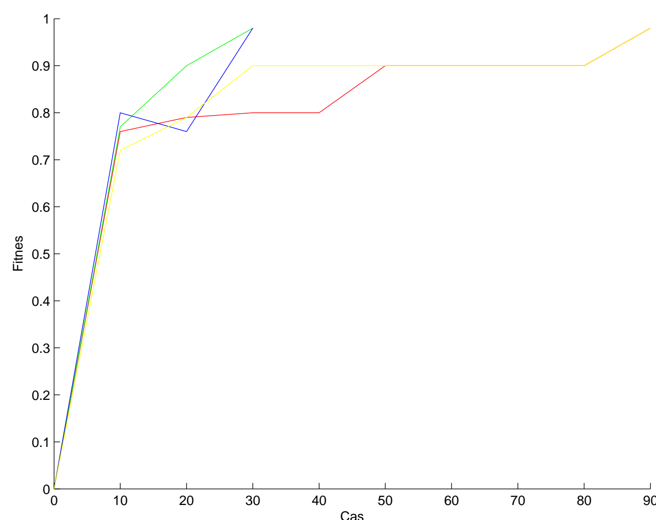
Obrázok 4: Priebeh fitnes



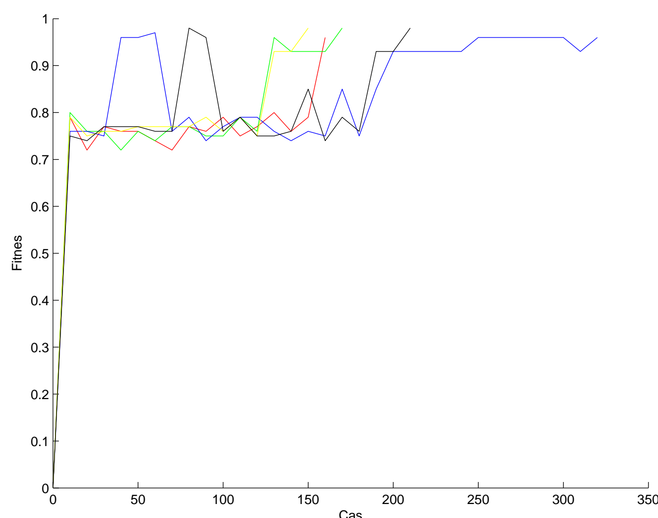
Obrázok 5: Priebeh fitnes

V ďalšej časti sa budeme venovať druhej časti zadania, t.j. situácii, keď sa má kôň dostať po prechode všetkých pozícií do východiskovej. Treba si uvedomiť, že vyhodnocovacia funkcia je pre tento prípad iné ako v predošlom prípade, čomu treba prispôsobiť aj interpretáciu výsledných údajov. Na obrázku 6 vidíme podobne ako v predchádzajúcej časti znázornený priebeh vyhodnocovacej funkcie v čase. Počet postupností v stavovom vektore je opäť 10. Z grafu je vidieť, že vývoj fitnes je v tomto prípade o niečo pomalší ako v predchádzajúcom prípade. Vývoj riešenia sa však pozastavuje po dosiahnutí určitého bodu. Po krátkej dobe sa síce nájde nejaká postupnosť,

avšak hľadanie riešenia, pri ktorom sa kôň po prejdení všetkých políčok ocitne čo najbližšie k východiskovej pozícii je už pozvolné. Dosiahnuť maximálnu fitnes sa nám v reálnom čase nedarilo. Rovnako ako v predchádzajúcej časti, aj v tomto prípade sme spravili experimenty pri výrazne vyžších počtoch postupností v stavovom vektore. Dosiahnuté výsledky znova potvrdzujú, že voľba príliš vysokého počtu postupností nie je vhodná, pozri obrázok 7.



Obrázok 6: Priebeh fitnes



Obrázok 7: Priebeh fitnes

Štatistické výsledky dosiahnuté v jednotlivých prípadoch behov simulovaného žíhania sú sumarizované v tabuľke číslo 1. V tabuľke sú uvedené minimálne, maximálne a priemerné hodnoty časov, po ktorých bolo dosiahnuté riešenie. Z uvedených údajov je zrejmé, že situácia, keď sa kôň nemusí dostať na východiskovú pozíciu predstavuje omnoho jednoduchší problém. V druhom prípade, keď sa kôň má dostať na východiskovú pozíciu je omnoho náročnejší. Priemerné časové hodnoty, po ktorých sa dosiahol výsledok bol takmer dvojnásobný. Z výsledkov je rovnako vidieť, že odchýlky v jednotlivých behoch algoritmu môžu byť dosť veľké. Tento fakt interpretujeme

tak, že v algoritme hrá nezanedbateľnú úlohu náhoda, ktorá spôsobuje, že v jednotlivých behoch môže byť priebeh rozdielny.

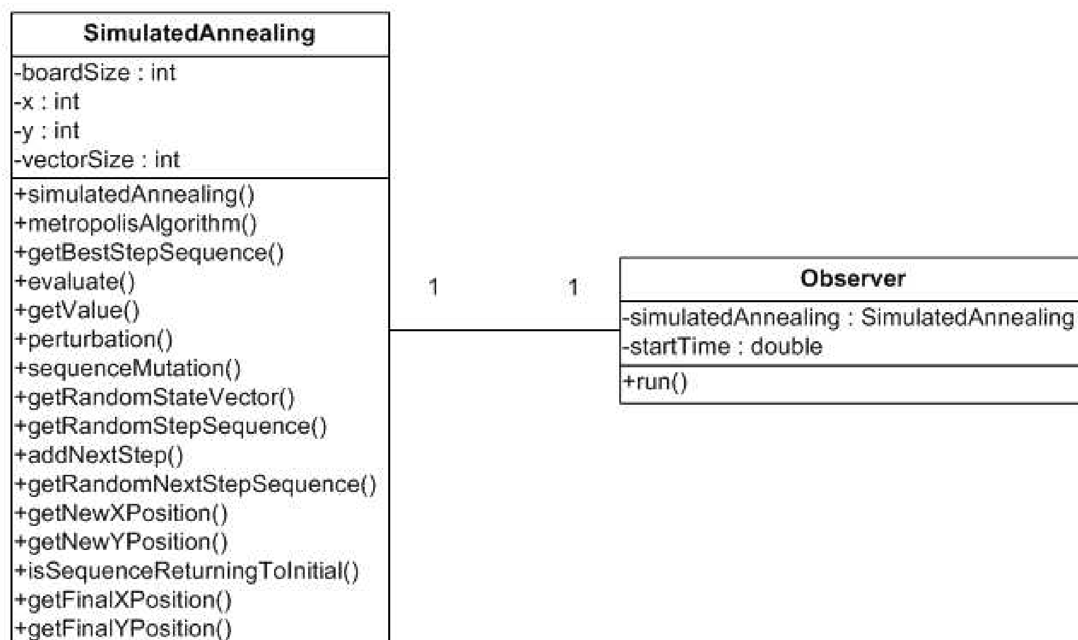
Tabuľka 1: Štatistiky: časy dosiahnutia riešenia v jednotlivých prípadoch

	Bez návratu		**S návratom**	
Dĺžka vektora	**10**	**500**	**10**	**500**
min	20	50	30	160
max	50	160	90	320
priemer	36	104	58	202

5 Implementácia

Nasledujuca kapitola je venovaná implementácii riešenia zadania.

Ako implementačný jazyk bol použitý jazyk Java verzie 1.5. Na obrázku 8 je znázornený diagram tried, ktoré boli vytvorené pre riešenie zadania. Hlavná funkcionalita je implementovaná v triede *SimmulatedAnnealing*. Trieda *Observer* slúži len na priebežné monitorovanie stavu procesu žíhania.



Obrázok 8: Diagram tried

Atribútmi triedy *SimmulatedAnnealing* sú:

- *boardSize* - rozmer hracieho poľa
- x - horizontálna súradnica koňa
- y - vertikálna súradnica koňa

- *vectorSize* - dĺžka vektora reprezentujúci stavovú premennú (počet postupností)

V triede sa nachádzajú operácie zabezpečujúce samotný proces žíhania a rôzne pomocné operácie.

Základ vykonávania procesu žíhania je tvorený operáciami *simulatedAnnealing* a *metropolisAlgorithm*. Význam ostatných operácií je nasledovný:

- *getBestStepSequence* - vráti najlepšiu postupnosť zo stavového vektora (najdlhšia korektná postupnosť)
- *evaluate* - ohodnotí stavový vektor (fitnes)
- *getValue* - ohodnotí jednu postupnosť
- *perturbation* - perturbácia stavového vektora
- *sequenceMutation* - perturbácia jednej postupnosti
- *getRandomStateVector* - vytvorí náhodný stavový vektor
- *getRandomStepSequence* - vytvorí náhodnú korektnú postupnosť krokov
- *addNextStep* - pridá jeden korektný krok ku existujúcej postupnosti
- *getRandomNextStepSequence* - vráti náhodnú postupnosť ôsmych rôznych možných krokov koňa (postupnosť dĺžky osem)
- *getNewXPosition* - vypočíta novú horizontálnu pozíciu koňa pre daný krok z danej pozície
- *getNewYPosition* - vypočíta novú vertikálnu pozíciu koňa pre daný krok z danej pozície
- *isSequenceReturningToInitial* - zistí, či nájdená postupnosť, je taká, že jej koncová pozícia je zhodná s východiskovou
- *getFinalXPosition* - zistí finálnu horizontálnu pozíciu pre danú postupnosť
- *getFinalYPosition* - zistí finálnu vertikálnu pozíciu pre danú postupnosť

Trieda *Observer* je implementovaná ako zvlášť vlákno, ktoré sleduje štatistické premenné a v definovaných intervaloch ich vypíše na konzolu.

Pre spustenie procesu žíhania použite metódu *main* triedy *SimulatedAnnealing*.

Použitá literatúra

[1] Kvasnička, V., Pospíchal J., Tiňo P.: Evolučné algoritmy. Slovenská technická univerzita v Bratislove, vydavateľstvo STU, 2000.

[2] Laarhoven, P.J.M., Aarts E.H.L.: Simulated annealing. Theory and applications. Reidel, Dordrecht, 1987.

[3] Otten, R.H.J.M., Ginneken, L.P.P.P.: Annealing algorithm. Kluwer, Boston, 1989.

[4] Simulated Annealing
`http://www.cs.nott.ac.uk/ gxk/aim/notes/simulatedannealing.doc`