Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Ján Májek

Bin packing

Semestrálny projekt z predmetu Evolučné algoritmy

Vedúci projektu: Doc. RNDr. Jiří Pospíchal, DrSc.
Dátum: Jún, 2006

Úvod

Predmetom tohto dokumentu je vypracovať „case study“ s témou Bin packing podľa nasledovného zadania:

Pakovanie je klasickým problémom v optimalizácii. Predpokladajme, že máme nákladný voz schopný uviezť akýkoľvek počet krabíc, pokiaľ ich celková váha neprekročí W. V skladisku máte nelimitované množstvo rozličných typov tovaru. Každý typ položky i má váhu w_i a hodnotu v_i . Váš problém je určiť, ktorou kombináciou objektov dosiahnete najvyššiu hodnotu bez prekročenia maximálnej povolenej celkovej váhy. Napríklad, keď váhy a hodnoty sú

Položka	váha	hodnota
pomaranče	3	5
knihy	8	11

potom nákladďák s maximálnou nosnosťou W=200 môže viezť:

pomaranče	knihy	váha	hodnota
30	13	194	293
40	10	200	310
50	6	198	316
60	2	196	322

Najlepšia stratégia je naložiť toľko krabíc s pomarančmi ako sa len dá a zvyšok zaplniť knihami. Vyriešte problém pre väčší počet položiek s náhodne definovanými hodnotami a váhami pomocou genetického algoritmu, s obmedzeným počtom niektorých položiek, vymyslite si vlastnú reprezentáciu problému a k nej odpovedajúce chromozómy, mutácie a kríženie, porovnajte váš algoritmus s kanonickým genetickým algoritmom, a použite pokutovaciu funkciu. (Adaptované z [1])

Dátová reprezentácia problému

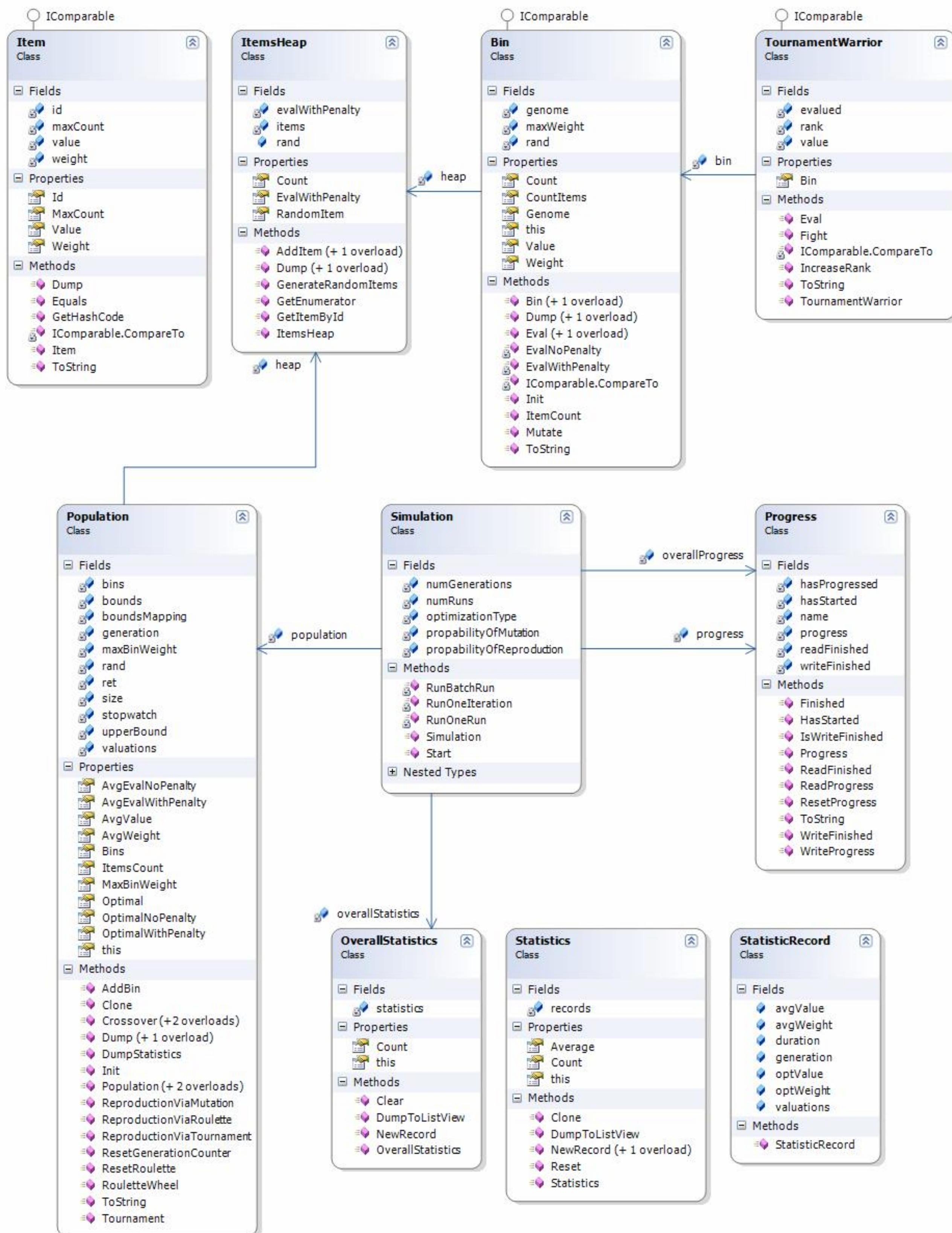
Rozhodol som sa implementovať program v jazyku C# ako dialógovú aplikáciu pre MS Windows. Ako prvé bolo treba zvoliť vhodnú reprezentáciu dátového modelu. Môj návrh vychádzal zo štúdia problému na [1] a [2]. Dátový model zobrazený na obrázku 1 zobrazuje všetky použité triedy a niektoré vzťahy medzi nimi. Niektoré z nich ďalej popíšem.

Trieda Item

Základnom je trieda `Item`, ktorá implementuje jednu položku s definovanou váhou (`weight`), hodnotou (`value`) a maximálnym dostupným počtom tejto položky (`maxCount`).

Trieda ItemsHeap

Všetky použiteľné položky (`items`) sú dostupné pomocou triedy `ItemsHeap`, ktorá predstavuje hromadu všetkých použiteľných položiek pre optimalizačný problém. Na začiatku programu je možné položky na hromade vygenerovať s náhodnými váhami, hodnotami a maximálnymi počtami podľa zvolených intervalov (`GenerateRandomItems()`). Je takisto možné uložiť, resp. nahrať položky na hromade do/zo súboru. Trieda `ItemsHeap` implementuje aj ďalšie obslužné vlastnosti a metódy na získavanie položiek (`RandomItem`, `GetItemById()`), resp. na pridanie položky na hromadu (`AddItem()`).



Obrázok 1 - Dátový model aplikácie Bin Packer

Trieda Bin

Trieda `Bin` predstavuje jednu nádobu, ktorej ideálne naplnenie je predmetom optimalizačného problému. Všetky nádoby majú nastavenú premennú určujúcu ich maximálnu možnú záťaž (`maxWeight`). Nádoby, ktorých súčet váh

položiek prekračuje túto hranicu sú penalizované (viď nižšie) pri ohodnotení fitnes. Položky v nádobe sú reprezentované **neusporiadaným poľom** (`genome`) položiek z hromady položiek, pričom platí, že žiadna položka nemôže byť v nádobe viac krát ako je hodnota jej maximálneho dostupného počtu (`maxCount`). Tým je reprezentovaný genóm jedného jedinca. Trieda implementuje vlastnosti na zistenie počtu položiek (`Count`), zistenie počtu jedinečných položiek rovnakého typu (`ItemCount()`) a na zistenie počtu rozličných druhov položiek v nádobe (`CountItems`). V pôvodnom návrhu som totiž vychádzal z reprezentácie uvedenej v [2], kde sú položky v nádobe určené typom položiek a ich konkrétnym počtom. Keďže ale v [2] vystupuje len operácia mutácie, pri implementácií kríženia som narazil na problém s reprezentáciou genómu a musel som ju zmeniť na všeobecnejší typ uvedený vyššie.

Trieda `Bin` ďalej implementuje metódy na vyhodnotenie položiek v nádobe. Implementoval som dve vyhodnocovacie metódy, ktoré sa líšia v prístupe ohodnotenia nádoby, ktorej súčet váh jednotlivých položiek prekračuje maximálnu povolenú váhu nádoby. V prípade, že maximálna váha nádoby nie je prekročená, obe vyhodnocovacie metódy vrátia rovnakú hodnotu a to súčet hodnôt jednotlivých položiek nachádzajúcich sa v nádobe.

Prvá vyhodnocovacia metóda (`EvalNoPenalty()`) v prípade prekročenia maximálnej možnej váhy vráti jednoducho nulovú hodnotu. Druhá metóda (`EvalWithPenalty()`) používa v prípade prekročenia maximálnej povolenej váhy pokutovaciu funkciu, ktorá zohľadňuje o koľko bola váha prekročená, pričom od skutočnej hodnoty (súčet hodnôt všetkých položiek v nádobe) odpočíta rozdiel váhy, o ktorú bola prekročená maximálna váha nádoby. Týmto spôsobom by mali dostať šancu aj jedinci, ktorí prekračujú maximálnu váhu len o málo a je predpoklad, že by neskôr mohli byť ešte užitoční v optimalizačnom procese.

V triede `Bin` som ešte implementoval aj metódu mutácie, ktorá bude popísaná neskôr.

Trieda Population

Trieda `Population` zastrešuje jednu populáciu nádob v konkrétnom stave. Implementuje jadro optimalizačného a vyhodnocovacieho procesu. Ide hlavne o metódy `ReproduceViaMutation()`, `ReproduceViaTournament()`, `Tournament()`, `ReproduceViaRoulette()`, `ResetRoulette()`, `RouletteWheel()` a vlastnosti na získanie priemerných a optimálnych jedincov (nádob) `AvgEvalNoPenalty`, `AvgEvalWithPenalty`, `AvgValue`, `AvgWeight` a `Optimal`, `OptimalNoPenalty` a `OptimalWithPenalty`.

Trieda `Population` udržiava kolekciu nádob, čím reprezentuje populáciu. Implementuje aj obslužné metódy, vlastnosti a indexery na pridávanie a získavanie nádob (`Bins`, `this[]`, `AddBin()`, `InitBins()`). Na začiatku umožňuje podobne ako trieda `ItemsHeap` vygenerovať náhodných jedincov (nádoby) z položiek hromady, ktorých počet (veľkosť populácie) určuje používateľ. Je rovnako možné uložiť a načítať stav populácie do/z textového súboru (v skutočnosti ide o jeden textový súbor aj pre hromadu aj pre populáciu, keďže tieto na sebe závisia).

Trieda Simulation

Trieda `Simulation` zastrešuje vykonanie samotného optimalizačného procesu podľa zvolených nastavení: počet behov (`numRuns`), počet generácií v jednom behu (`numGenerations`), použitý algoritmus (`optimizationType`), pravdepodobnosť reprodukcie – iba pri rulete (`propabilityOfReproduction`) a pravdepodobnosť mutácie (`propabilityOfMutation`). Na základe toho volá potrebný počet krát príslušné metódy populácie (`population`) na reprodukciu (`RunOneIteration()`, `RunOneRun()` a `RunBatchRun()`), informuje o progrese behu (`progress`), celkovom progrese pri dávkovom spracovaní (`overallProgress`) a ukladá štatistiky (`overallStatistics`). Pri dávkovom behu je na začiatku populácia klonovaná, aby bol zachovaný jej stav a pred každým novým behom je tento stav obnovený.

Trieda Progress

Keďže optimalizácia beží v samostatnom vlákne, trieda `Progress` zabezpečuje synchronizáciu s monitorovacou metódou, ktorá na základe toho mení príslušné vizuálne prvky vo formulári. Pri návrhu tejto triedy som vychádzal zo štandardného problému producenta a konzumenta popísaného v [4]. V simulácií vystupujú dve inštancie tejto triedy, pričom jedna reprezentuje progres jedného behu a druhá progres celej dávky.

Triedy StatisticRecord, Statistics a OverallStatistics

Na zhromažďovanie štatistických údajov o optimalizačnom procese slúžia triedy `StatisticRecord`, `Statistics` a `OverallStatistics`. Trieda `StatisticRecord` ukladá priemernú váhu (`avgWeight`) a hodnotu (`avgValue`)

jedincov v populácií ako aj hodnotu (`optValue`) a váhu (`optWeight`) optimálneho jedinca v populácií. Tieto hodnoty sa ukladajú po každej iterácií. Trieda `Statistics` zhromažďuje štatistické údaje o vývoji v jednom behu. Obsahuje kolekciu záznamov typu `StatisticRecord` pre každú iteráciu behu. Poskytuje vlastnosti a metódy na výpis do súboru a formulárového prvku (`DumpToListView()`) a výpočet priemerných hodnôt (`Average`). Pre každý beh sú ukladané štatistické dáta v triede `OverallStatistics` ako kolekcia štatistík typu `Statistics`. Takisto poskytuje nástroje na výpis do súboru a formulárového prvku (`DumpToListView()`).

Trieda TournamentWarior

Na záver som pre potreby algoritmu turnaja vytvoril triedu `TournamentWarrior`, ktorá bude opísaná neskôr v tomto texte pri opise algoritmu turnaja.

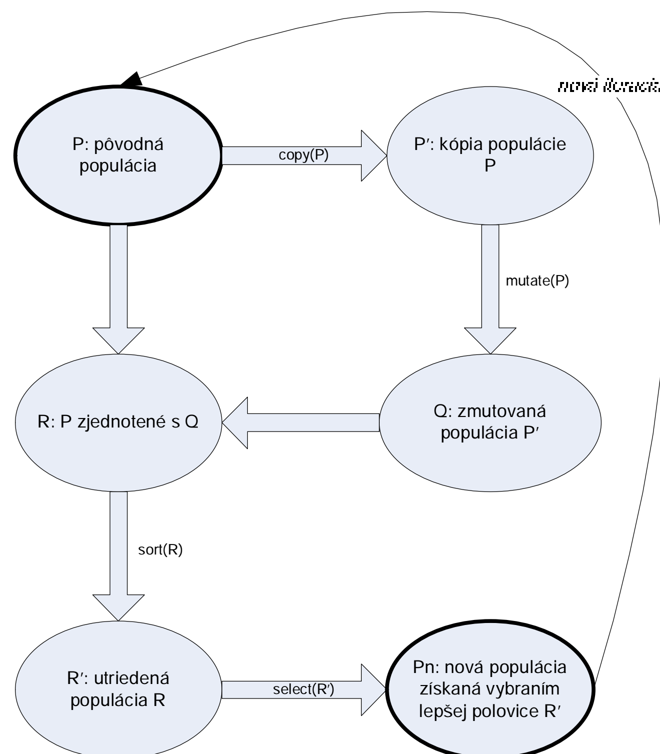
Použité algoritmy

Rozhodol som implementovať tri algoritmy optimalizácie:

- optimalizácia mutáciou
- optimalizácia genetickým algoritmom s použitím turnaja,
- optimalizácia genetickým algoritmom s výberom prvkov pomocou rulety

Optimalizácia mutáciou

Tento najjednoduchší koncept vychádza z [2] a využíva metódu mutovania jednotlivých jedincov v populácií. Pri každej iterácií sú prejdení všetci jedinci v populácií a sú vygenerovaní noví jedinci, na ktorých je prevedený proces mutácie s určitou pravdepodobnosťou. Nová „zmutovaná" populácia je pridaná k pôvodnej populácií. Následne sú všetci jedinci zoradení podľa ohodnotenia fitnes príslušnou metódou (s alebo bez penalizácie, podľa nastavenia) a do ďalšieho procesu postupujú tí jedinci, ktorí sa nachádzajú v lepšej polovici populácie. Priebeh jednej iterácie je znázornený na obrázku 2.



Obrázok 2 - Priebeh jednej iterácie pri použití algoritmu mutácie

Samotná mutácia jedného jedinca (jednej nádoby) je implementovaná metódou `Mutate()` triedy `Bin`. Prebieha tak, že jednotlivé položky z tejto nádoby sú s určenou pravdepodobnosťou nahradené inými náhodnými položkami z hromady, ak je to možné (nie je už v danej nádobe počet položiek takéhoto typu maximálny; inými slovami, nie sú tieto položky pre túto nádobu už vyčerpané).

Po prvej implementácií takejto mutácie sa ukázalo, že počty prvkov v jednotlivých nádobách populácie sa postupne približujú k jednému číslu (a to pre všetky použité algoritmy). Preto som proces mutácie rozšíril tak, že s určenou pravdepodobnosťou bude do nádoby pridaná náhodná položka a s rovnakou pravdepodobnosťou bude niektorá položka z nádoby odstránená.

Výpočtová zložitosť je pri tomto algoritme veľmi veľká, keďže sa prechádzajú všetci jedinci v každej iterácií a všetci podstupujú mutáciu.

Optimalizácia genetickým algoritmom s použitím turnaja

Ako ďalší algoritmus som sa rozhodol použiť klasický genetický algoritmus pomocou turnaja. Vychádzal som zo štandardnej implementácie uvedenej napr. v [3]:

1. Náhodne sa vyberie istý počet jedincov z populácie, ktorí predstavujú rodičovskú populáciu. Ich počet je náhodný z intervalu od jedna do počtu jedincov v populácií. Zvyšok jedincov sa nezúčastňuje reprodukčného procesu.
2. Táto rodičovská populácia sa skopíruje a jej kópia prejde procesom mutácie s určenou pravdepodobnosťou (ten je rovnaký ako bol popísaný vyššie v kapitole o optimalizácií mutáciou).
3. Rodičovská a novo zmutovaná populácia vstupuje do turnaja.
4. Turnajom sa vyberú jedinci, pričom ich počet je rovnaký ako bol pôvodný počet jedincov v rodičovskej populácií.
5. Títo vybraní jedinci sú zjednotení so zvyškom pôvodnej populácie, ktorá sa nezúčastnila reprodukčného procesu a vytvárajú novú populáciu.
6. Nová populácia vstupuje do ďalšej iterácie.

Turnaj som implementoval v triede `Population` metódou `Tournament()`. Do nej vstupujú vybrané nádoby (jedinci). Tieto sú zaobalené do triedy `TournamentWarrior`, ktorá poskytuje premennú určujúcu počet vyhratých turnajov (`rank`). V metóde `Population.Tournament()` prebehne náhodný počet bojov (turnajov), ktorý je rovný minimálne počtu vstupujúcich bojovníkov a najviac trojnásobku ich počtu.

Na začiatku majú všetci jedinci `rank=0`. Samotný turnaj je implementovaný tak, že sa vyberú dvaja náhodní rôzni jedinci, ktorí zvedú súboj. Ten je implementovaný v triede `TournamentWarrior` metódou `Fight()` a vyhráva ten jedinec, ktorý má vyššie ohodnotenie fitnes získané príslušnou hodnotiacou funkciou. Víťazovi sa zvýši hodnota `rank` o jedna. V prípade, že obaja jedinci majú rovnaké ohodnotenie fitnes, nezvýši sa `rank` ani jednému z nich. Na záver, po uskutočnení všetkých turnajov sú jedinci zoradení podľa vyhratých súbojov (`rank`) a lepšia polovica z nich je výstupom tohto reprodukčného procesu.

Pokiaľ ide o zložitosť tohto algoritmu je tiež pomerne veľká, v najhoršom možnom prípade väčšia alebo rovná ako v prípade použitia samotnej mutácie. V praxi odhadujem zložitosť asi na polovicu zložitosti algoritmu mutácie.

Optimalizácia genetickým algoritmom s výberom prvkov pomocou rulety

Pri tomto algoritme som takisto vychádzal z klasického genetického algoritmu s použitím rulety ako je opísaný v [3]:

1. Vytvorí sa prázdna nová populácia.
2. Pokiaľ nie je jej početnosť rovná početnosti aktuálnej populácie, vykonáva sa nasledovný proces:
   a. Kvázi náhodne pomocou algoritmu rulety sú vybraní dvaja rôzni jedinci.
   b. Títo s určenou pravdepodobnosťou podstúpia proces kríženia a proces mutácie.
   c. Jedinci sú pridaní do novej populácie.
3. Nová populácia vstupuje do ďalšej iterácie.

Proces kvázi náhodného výberu pomocou rulety som neriešil klasickým sekvenčným hľadaním intervalov ako je uvedené v [3], ale použil som myšlienku binárneho hľadania. Pred každou iteráciou sa raz prejde celá populácia a namapujú sa do poľa hranice fitnes pre tých jedincov, ktorí ju majú vyššiu ako nula (`bounds` a `boundsMapping`). To robí metóda `ResetRoulette()` triedy `Population`. Kvázi náhodný výber pomocou rulety som potom už jednoducho implementoval v metóde `RouletteWheel()` tej istej triedy tak, že som zvolil náhodné číslo z intervalu nula až súčet všetkých kladných fitnes jedincov v populácií (`upperBound`) a na základe neho som vykonal binárne hľadanie v mapovacom poli (`bounds`), pomocou ktorého sa určil prislúchajúci jedinec (`boundsMapping`).

Proces kríženia jedincov je implementovaný v metódou `Crossover()` triedy `Population`. Ide o jednobodové kríženie a jeho princíp je klasický. Náhodne sa určí bod kríženia z intervalu jedna až dĺžka kratšieho z dvoch krížených jedincov. V tomto bode sú genómy týchto jedincov vymenené. Napríklad pre dvoch jedincov s genómom *a* s dĺžkou *i* a *b* s dĺžkou *j* a bod kríženia *k*, pričom platí, že $1 \leq k < min(i, j)$ by proces kríženia vyzeral nasledovne:

$a_1, a_2, ..., a_{k-1}, a_k, a_{k+1}, ..., a_i$ è $a_1, a_2, ..., a_{k-1}, b_k, b_{k+1}, ..., b_j$
$b_1, b_2, ..., b_{k-1}, b_k, b_{k+1}, ..., b_j$ è $b_1, b_2, ..., b_{k-1}, a_k, a_{k+1}, ..., a_i$

Proces mutácie je rovnaký ako bol opísaný pri optimalizácií mutáciou.

Pokiaľ ide o zložitosť algoritmu, tá je o niečo lepšia ako pri predchádzajúcich dvoch algoritmoch. Závisí hlavne od zvolenej pravdepodobnosti kríženia, ktorá sa v predchádzajúcich dvoch nevyskytuje.

Experiment

Aplikáciu som sa snažil napísať čo najvšeobecnejšie, preto obsahuje pomerne veľké množstvo možných nastavení:

- veľkosť hromady,
- horná hranica váhy položiek použitá pri náhodnej inicializácií hromady,
- horná hranica hodnoty položiek použitá pri náhodnej inicializácií hromady,
- horná hranica maximálneho počtu jednotlivých položiek na hromade (pre použitie v jednej nádobe) požitá pri náhodnej inicializácií hromady
- horný interval počtu položiek v nádobách použitá pri náhodnej inicializácií populácie
- maximálna povolená váha položiek v jednej nádobe
- počet nádob v jednej populácií
- algoritmus, ktorý má byť použitý (mutácia, turnaj alebo ruleta),
- či použiť pokutovaciu funkciu pri ohodnocovaní nádob alebo nie,
- pravdepodobnosť kríženia pri použití genetického algoritmu s ruletou,
- pravdepodobnosť mutácie,
- počet generácií v jednom behu
- počet behov v dávke

Nebolo v mojich silách vyskúšať všetky kombinácie a nájsť optimálne parametre, dovolím si tvrdiť, že takáto úloha by bez problémov spĺňala nároky na samostatnú záverečnú prácu bakalárskeho štúdia na našej škole. Preto som sa rozhodol nastaviť niektoré parametre fixne nasledovným spôsobom:

- veľkosť hromady = **100**,
- horná hranica váhy položiek použitá pri náhodnej inicializácií hromady = **10000**,
- horná hranica hodnoty položiek použitá pri náhodnej inicializácií hromady = **1000**,
- horná hranica maximálneho počtu jednotlivých položiek na hromade (pre použitie v jednej nádobe) požitá pri náhodnej inicializácií hromady = **100,**
- horný interval počtu položiek v nádobách použitá pri náhodnej inicializácií populácie = **30,**
- maximálna povolená váha položiek v jednej nádobe = **100000,**
- počet nádob v jednej populácií = **100,**
- pravdepodobnosť kríženia pri použití genetického algoritmu s ruletou = **30%**,
- pravdepodobnosť mutácie = **20%**,

Počet behov som stanovil na **100** a počet generácií v jednom behu na **10000**, čo som musel neskôr pre časovú náročnosť znížiť pre algoritmus mutácie na **1000** a pre genetický algoritmus s použitím turnaja na **5000**.

Všetky experimenty vychádzali z jednej rovnakej náhodne vygenerovanej populácie, ktorú som uložil do súboru a nahral vždy pred spustením experimentu. Tento súbor je možné nájsť v priloženom programe (`initial-state.txt`).

Vykonal som celkovo 6 experimentov pre každý algoritmus s a bez použitia penalizačnej funkcie. Za cieľ experimentu som si stanovil zistiť rozdiely medzi jednotlivými algoritmami a vplyv použitia penalizačnej funkcie na výsledky dosiahnuté jednotlivými algoritmami.

Výsledky

Niektoré zaujímavé výsledky sú zobrazené v tabuľke 1 a ako porovnanie jednotlivých hodnôt, resp. vývoja týchto hodnôt v grafoch 1 až 14. Kompletné štatistické súbory sú k dispozícií v priloženom programe (adresár `statistics`).

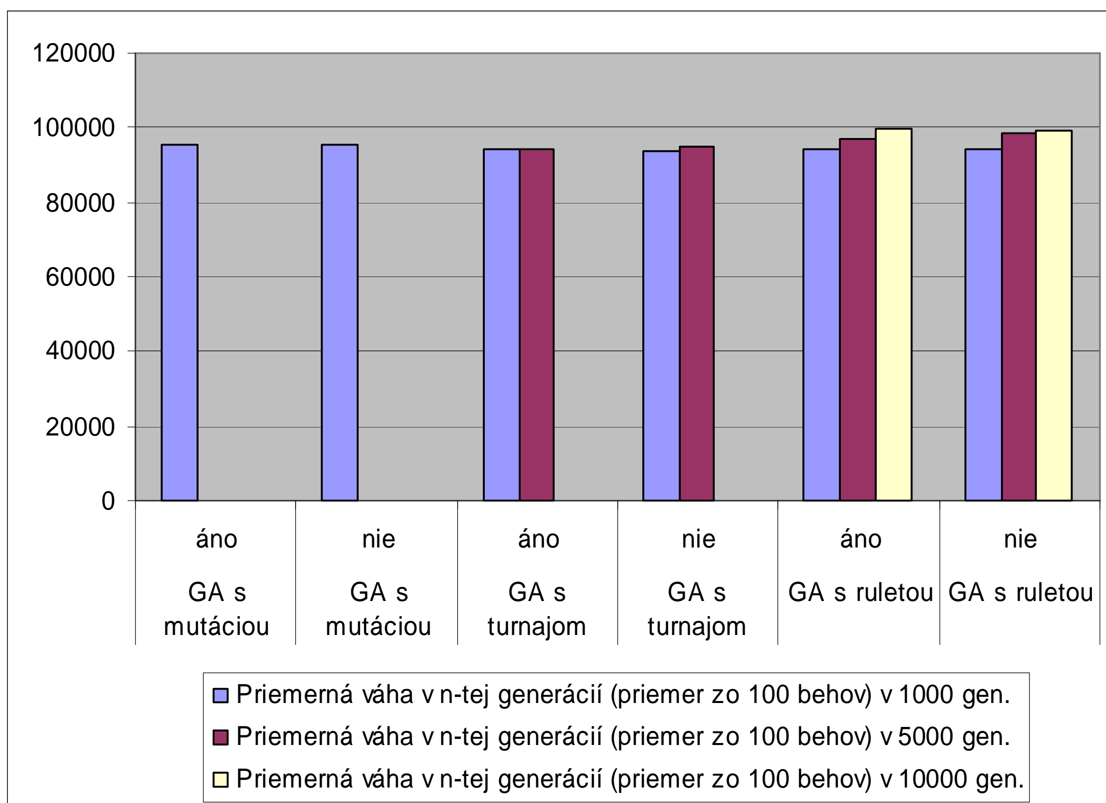
Algoritmus	**Pokutovacia funkcia**	**Priemerná váha v n-tej generácií (priemer zo 100 behov)**		
		v 1000 gen.	**v 5000 gen.**	**v 10000 gen.**
GA s mutáciou	áno	95332.82	0	0
GA s mutáciou	nie	95412.81	0	0
GA s turnajom	áno	94061.62	94319.62	0
GA s turnajom	nie	93845.19	94562.08	0
GA s ruletou	áno	94058.87	96865.48	99402.86
GA s ruletou	nie	94529.65	98528.89	99160.37

Algoritmus	**Pokutovacia funkcia**	**Váha optima v n-tej generácií (priemer zo 100 behov)**		
		v 1000 gen.	**v 5000 gen.**	**v 10000 gen.**
GA s mutáciou	áno	96700.88	0	0
GA s mutáciou	nie	95781.79	0	0
GA s turnajom	áno	95148.66	95247.05	0
GA s turnajom	nie	94384.72	95922.81	0
GA s ruletou	áno	91988.82	93296.75	94675.38
GA s ruletou	nie	92946.78	95037.46	94463.67

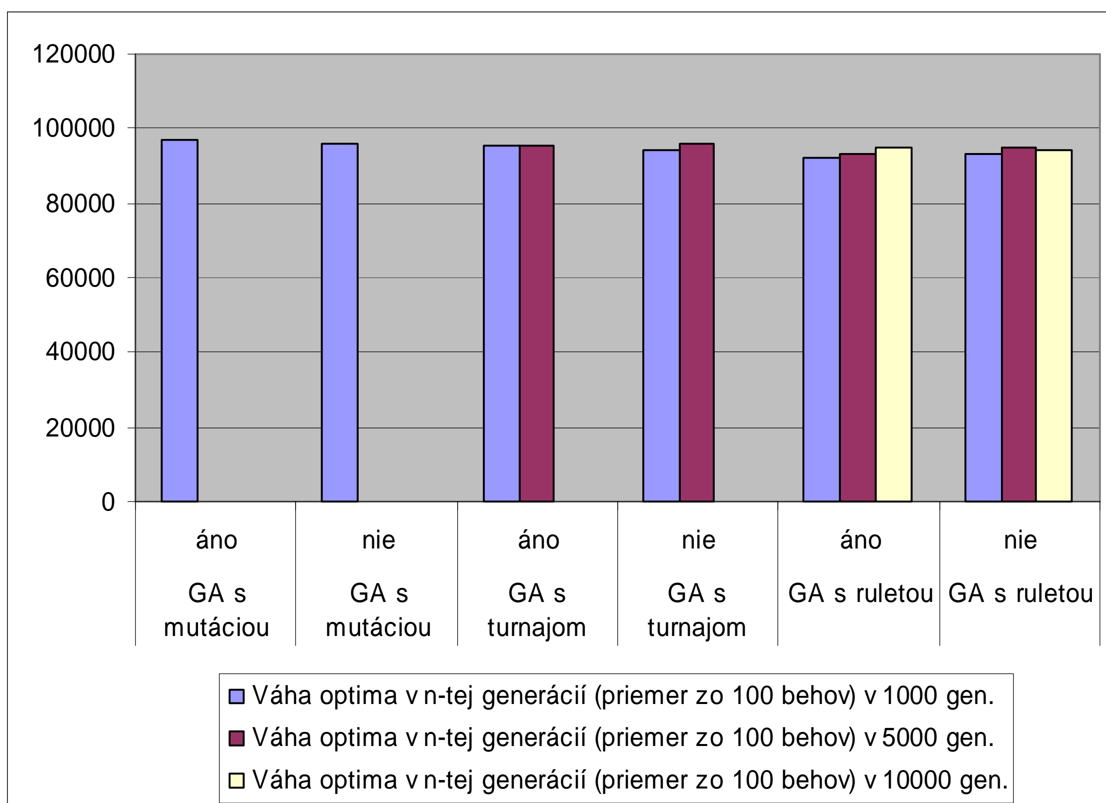
Algoritmus	**Pokutovacia funkcia**	**Priemerná hodnota v n-tej generácií (priemer zo 100 behov)**		
		v 1000 gen.	**v 5000 gen.**	**v 10000 gen.**
GA s mutáciou	áno	30982.29	0	0
GA s mutáciou	nie	30938.91	0	0
GA s turnajom	áno	21704.1	23059.41	0
GA s turnajom	nie	21776.1	23594.62	0
GA s ruletou	áno	17705.03	20716.26	23340.59
GA s ruletou	nie	17963.65	22193.55	24192.63

Algoritmus	**Pokutovacia funkcia**	**Hodnota optima v n-tej generácií (priemer zo 100 behov)**		
		v 1000 gen.	**v 5000 gen.**	**v 10000 gen.**
GA s mutáciou	áno	32281.45	0	0
GA s mutáciou	nie	32250.29	0	0
GA s turnajom	áno	24668.95	26472.33	0
GA s turnajom	nie	24722.16	26807.02	0
GA s ruletou	áno	20622.07	23130.03	25427.78
GA s ruletou	nie	20732.39	24426.91	26317.24

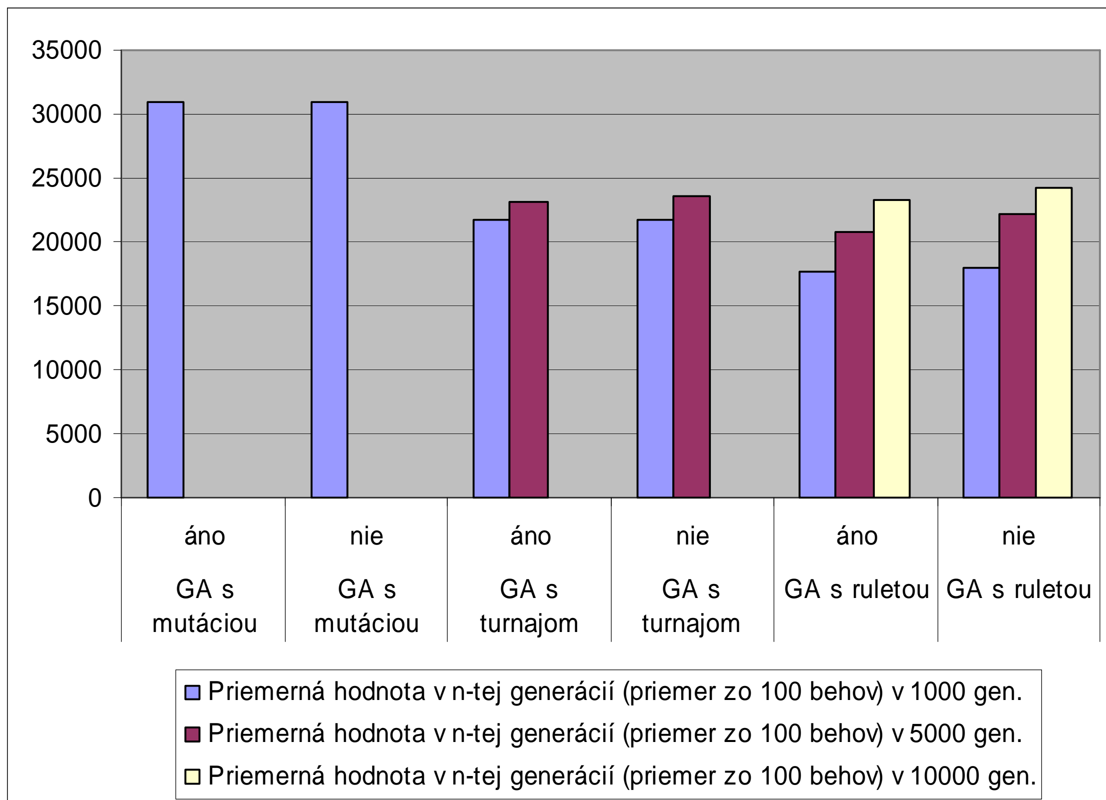
Tabuľka 1 - Priemerné hodnoty a váhy a optimálne hodnoty a váhy pre jednotlivé kombinácie algoritmov s a bez použitia pokutovacej funkcie namerané v 1000., 5000. a 10000. generácií (priemer zo 100 behov)



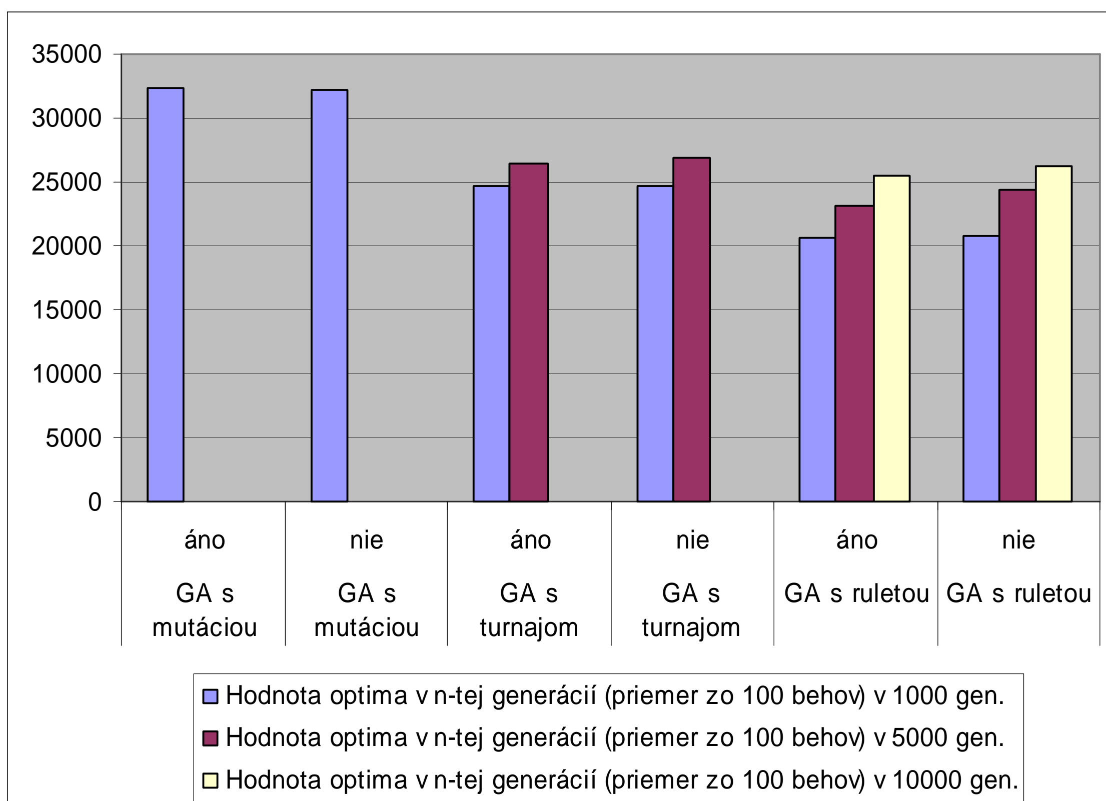
Graf 1 - Porovnanie priemernej váhy v 1000., 5000. a 10000. generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



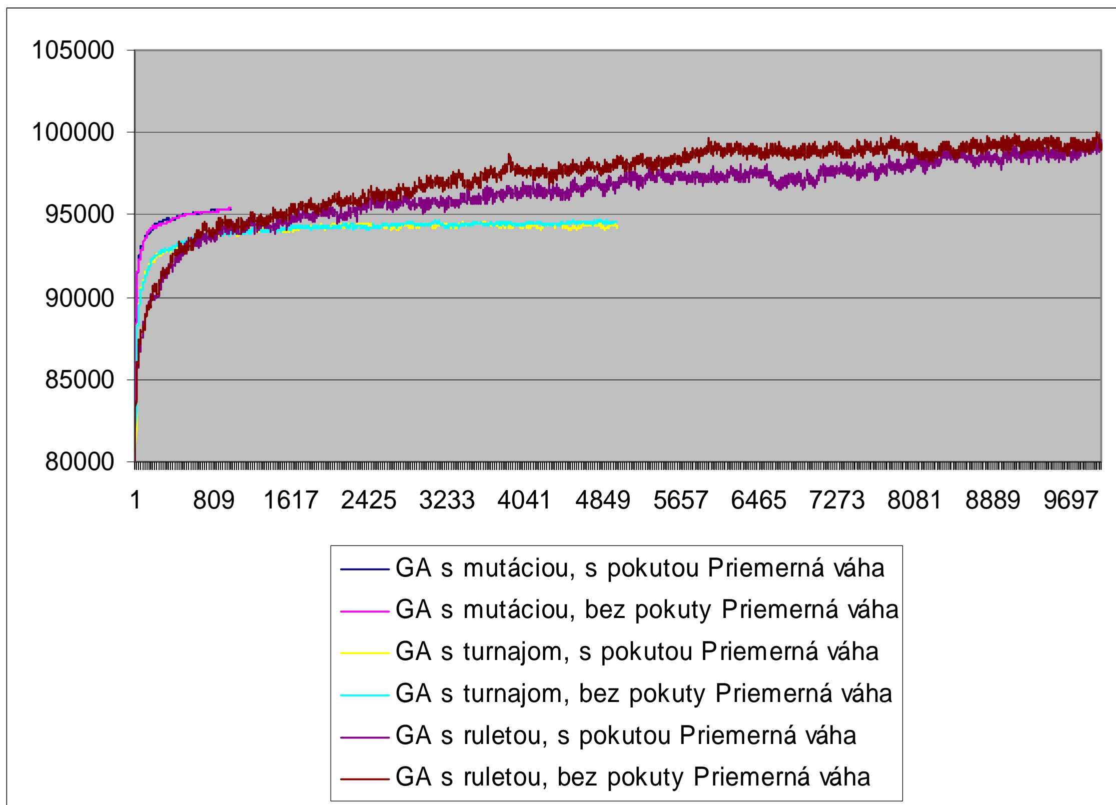
Graf 2 - Porovnanie váhy optimálneho jedinca v 1000., 5000. a 10000. generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



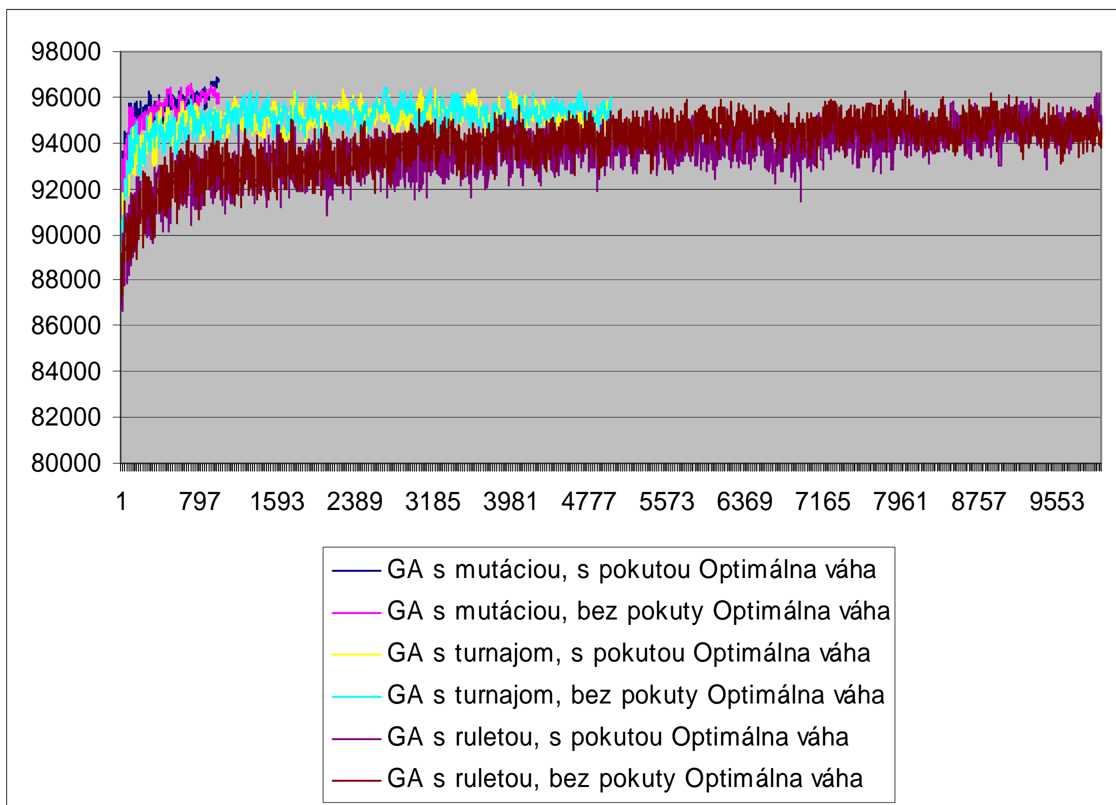
Graf 3 - Porovnanie priemernej hodnoty v 1000., 5000. a 10000. generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



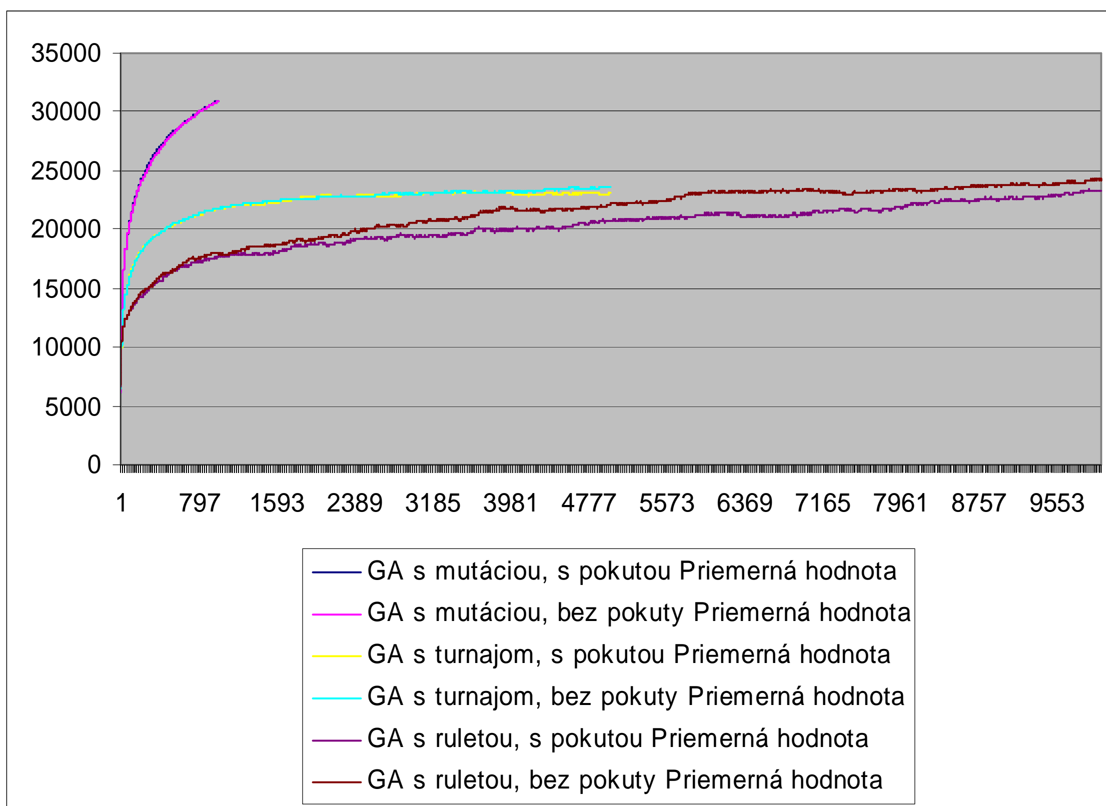
Graf 4 - Porovnanie hodnoty optimálneho jedinca v 1000., 5000. a 10000. generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



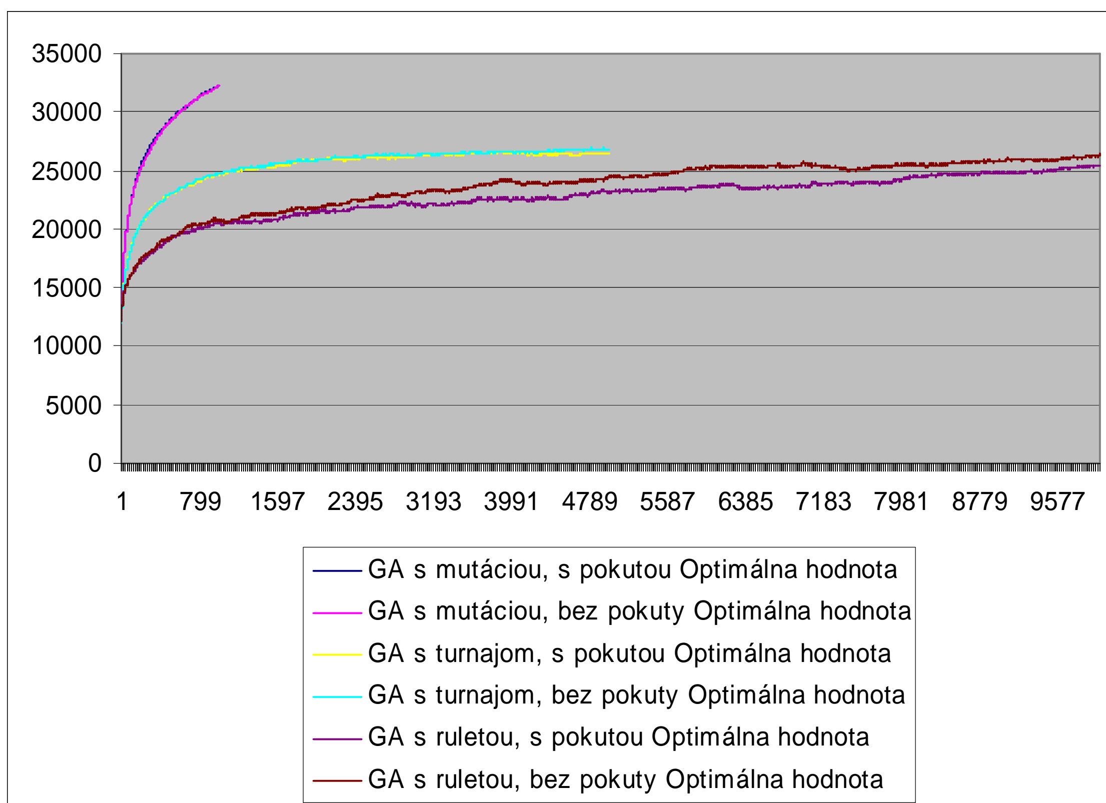
Graf 5 - Porovnanie vývoja priemernej váhy pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



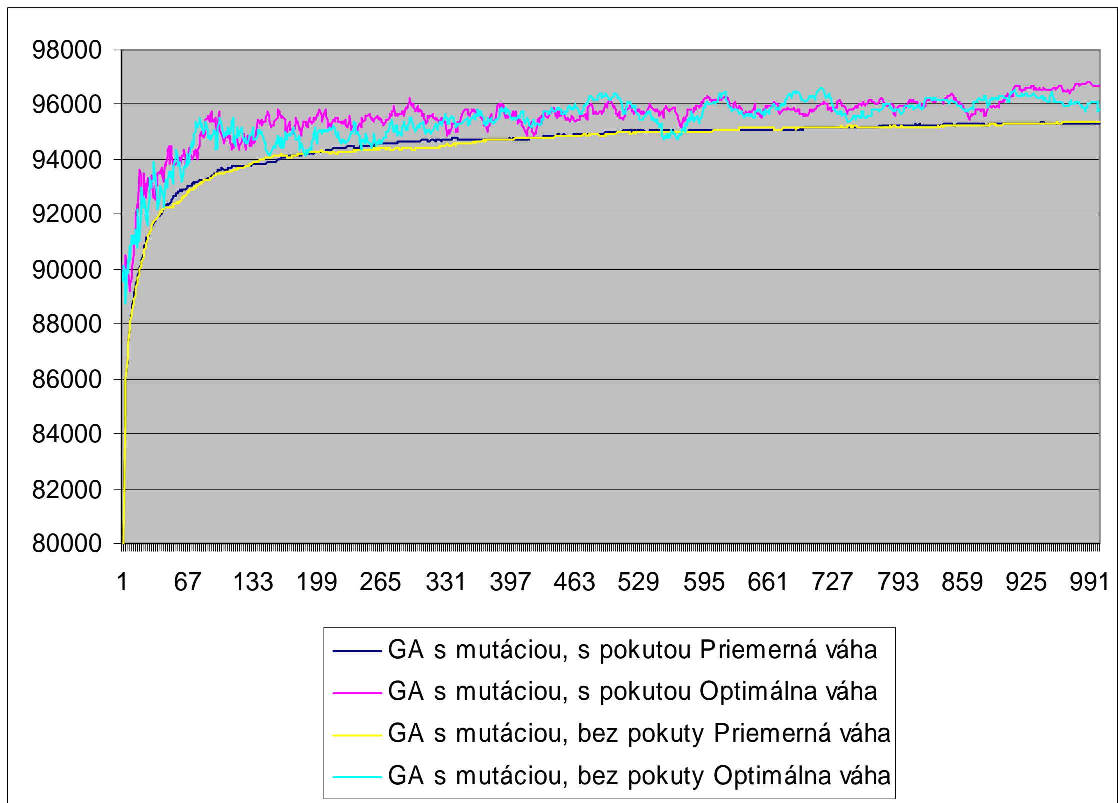
Graf 6 - Porovnanie vývoja váhy optimálneho jedinca pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



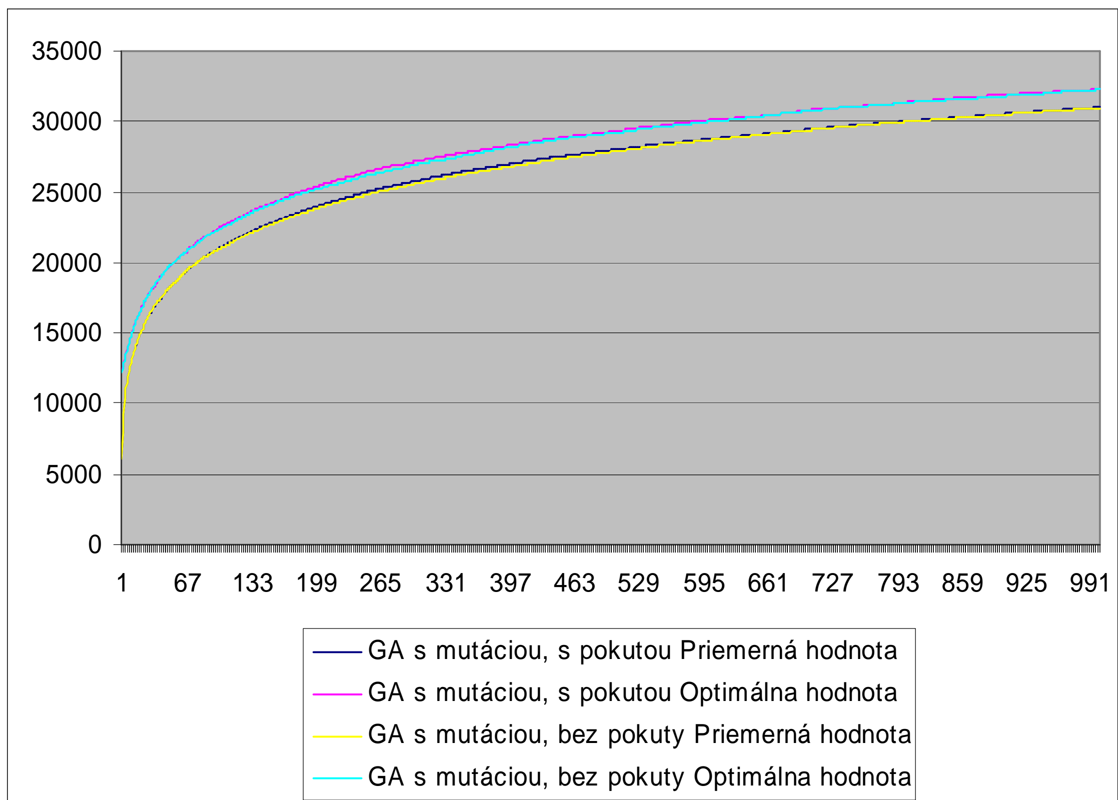
Graf 7 - Porovnanie vývoja priemernej hodnoty pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



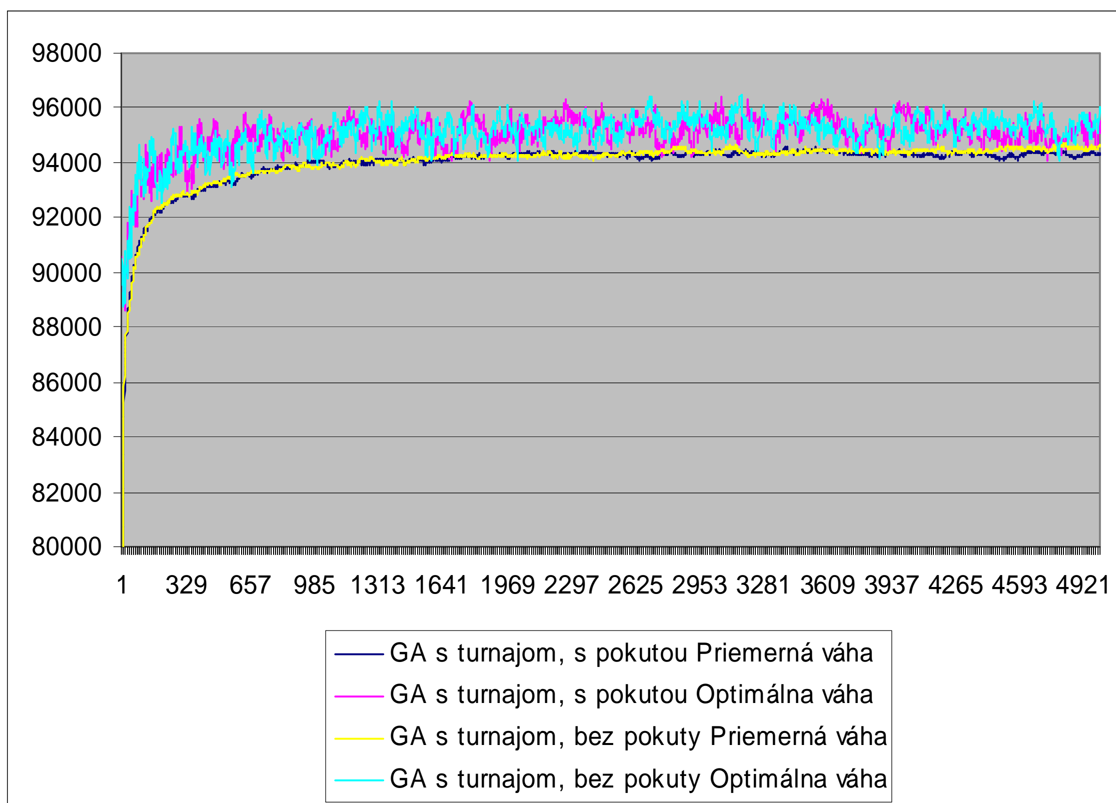
Graf 8 - Porovnanie vývoja hodnoty optimálneho jedinca pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



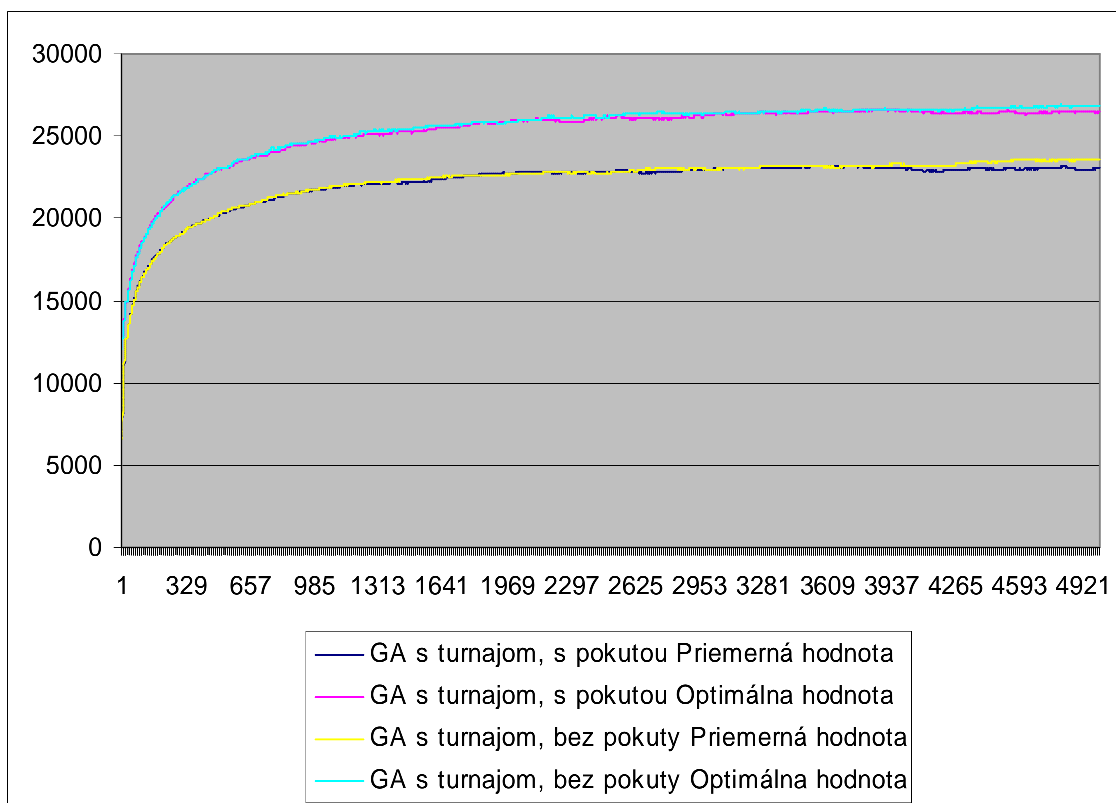
Graf 9 - Porovnanie vývoja priemernej váhy a váhy optimálneho jedinca pre GA s mutáciou s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



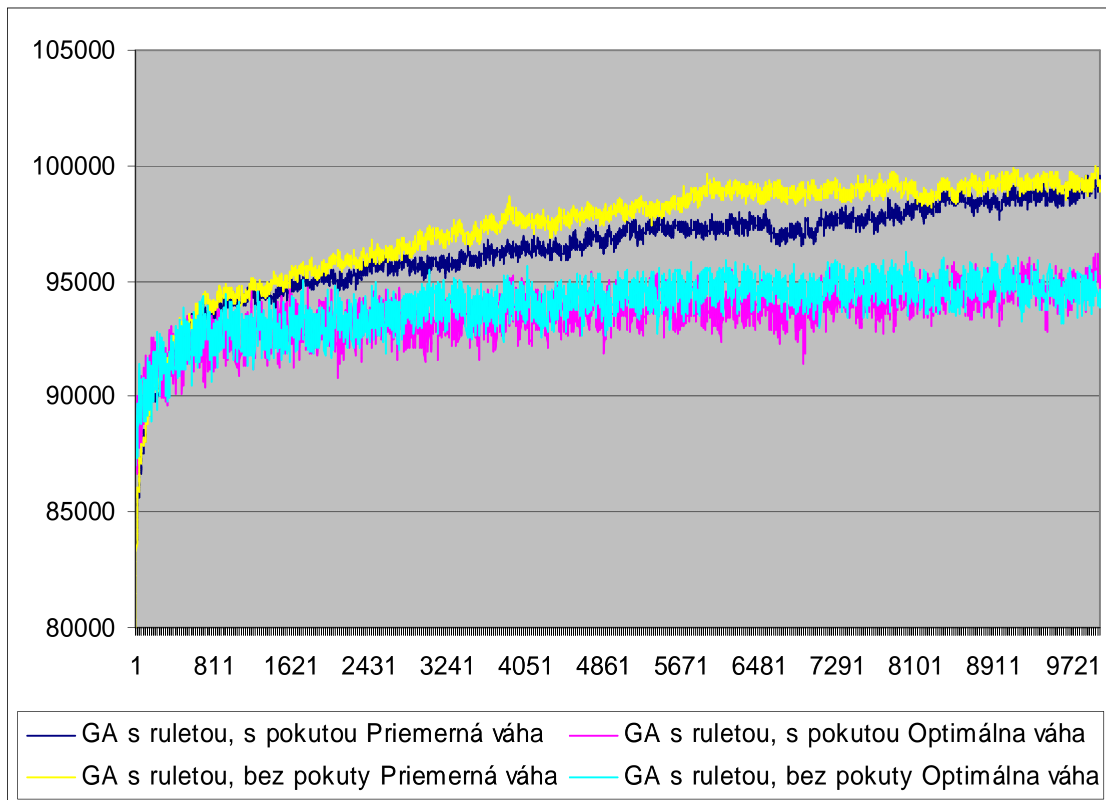
Graf 10 - Porovnanie vývoja priemernej hodnoty a hodnoty optimálneho jedinca pre GA s mutáciou s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



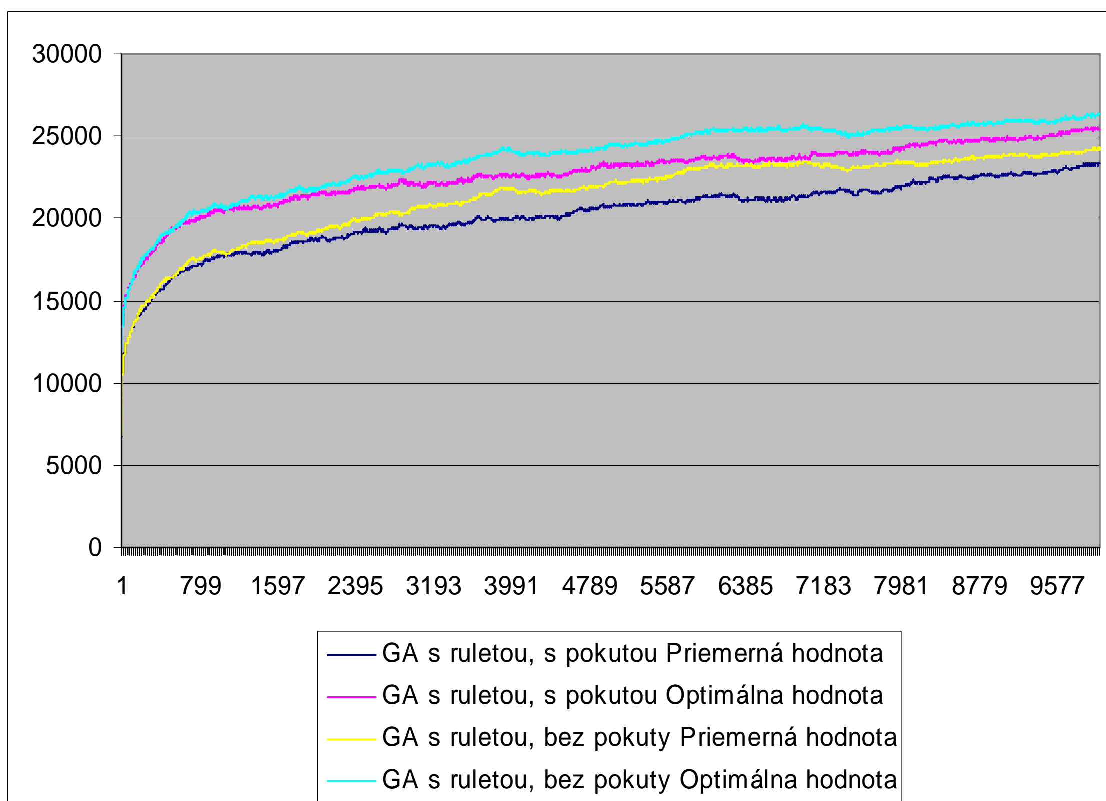
Graf 11 - Porovnanie vývoja priemernej váhy a váhy optimálneho jedinca pre GA s použitím turnaja s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



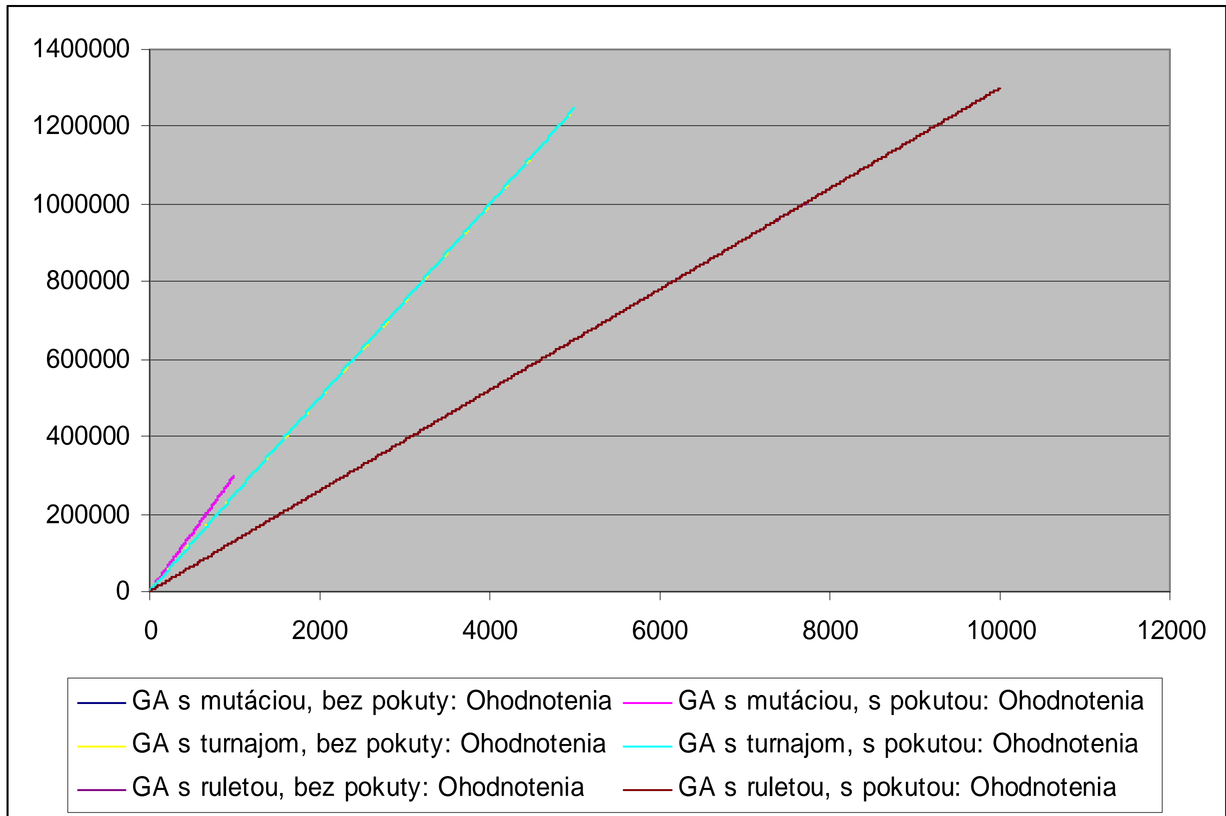
Graf 12 - Porovnanie vývoja priemernej hodnoty a hodnoty optimálneho jedinca pre GA s použitím turnaja s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



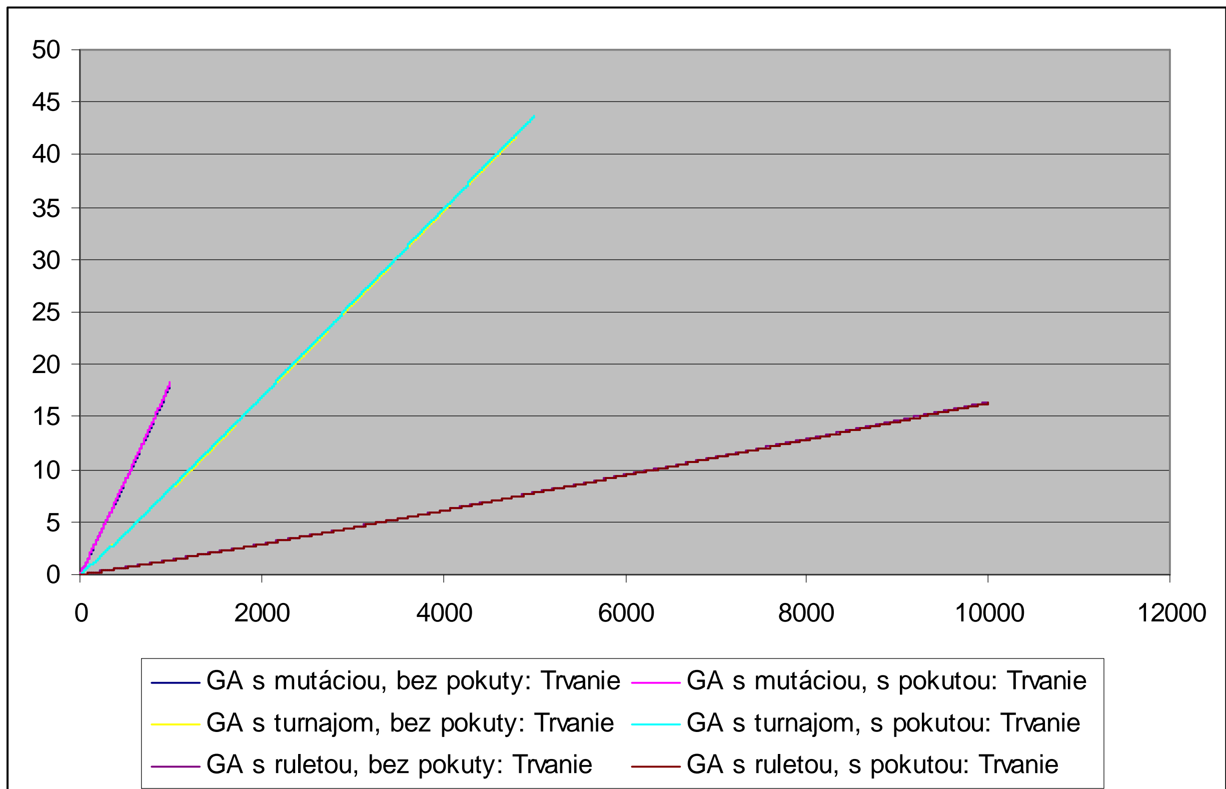
Graf 13 - Porovnanie vývoja priemernej váhy a váhy optimálneho jedinca pre GA s výberom pomocou rulety s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



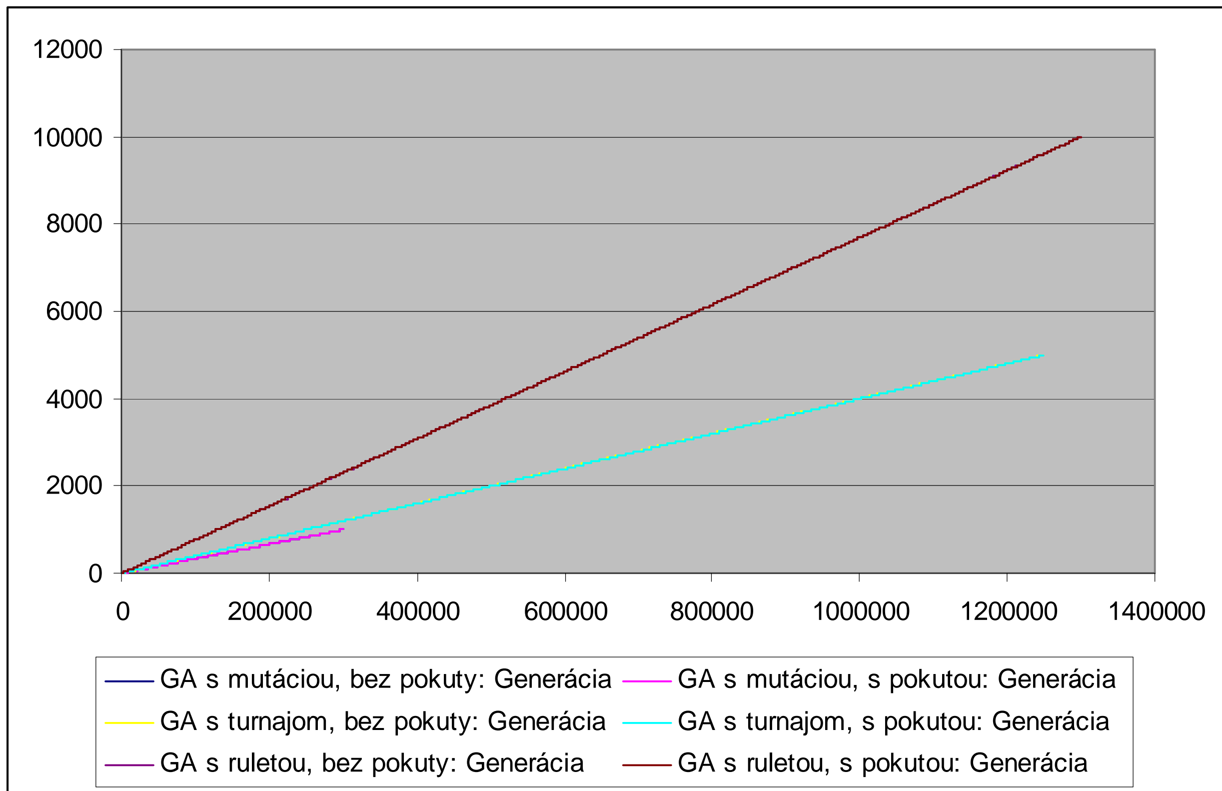
Graf 14 - Porovnanie vývoja priemernej hodnoty a hodnoty optimálneho jedinca pre GA s výberom pomocou rulety s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



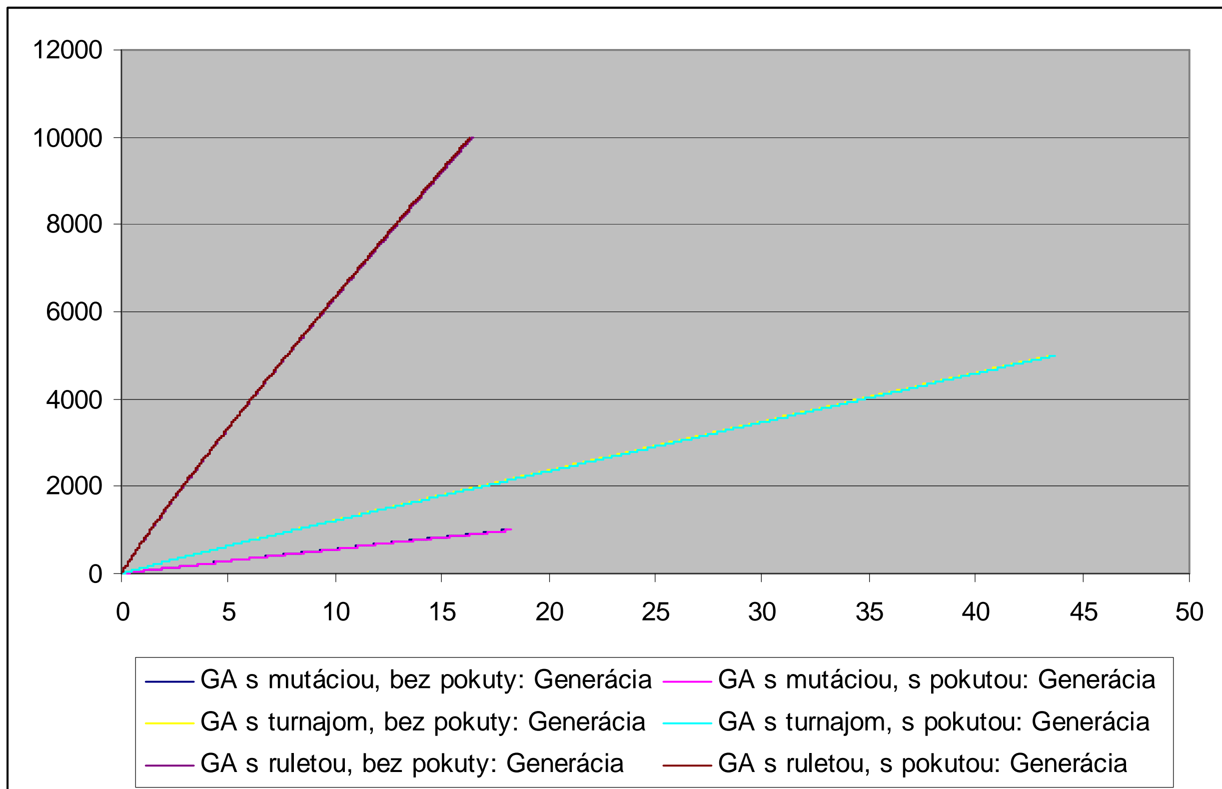
Graf 15 - Porovnanie počtu ohodnotení podľa počtu generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



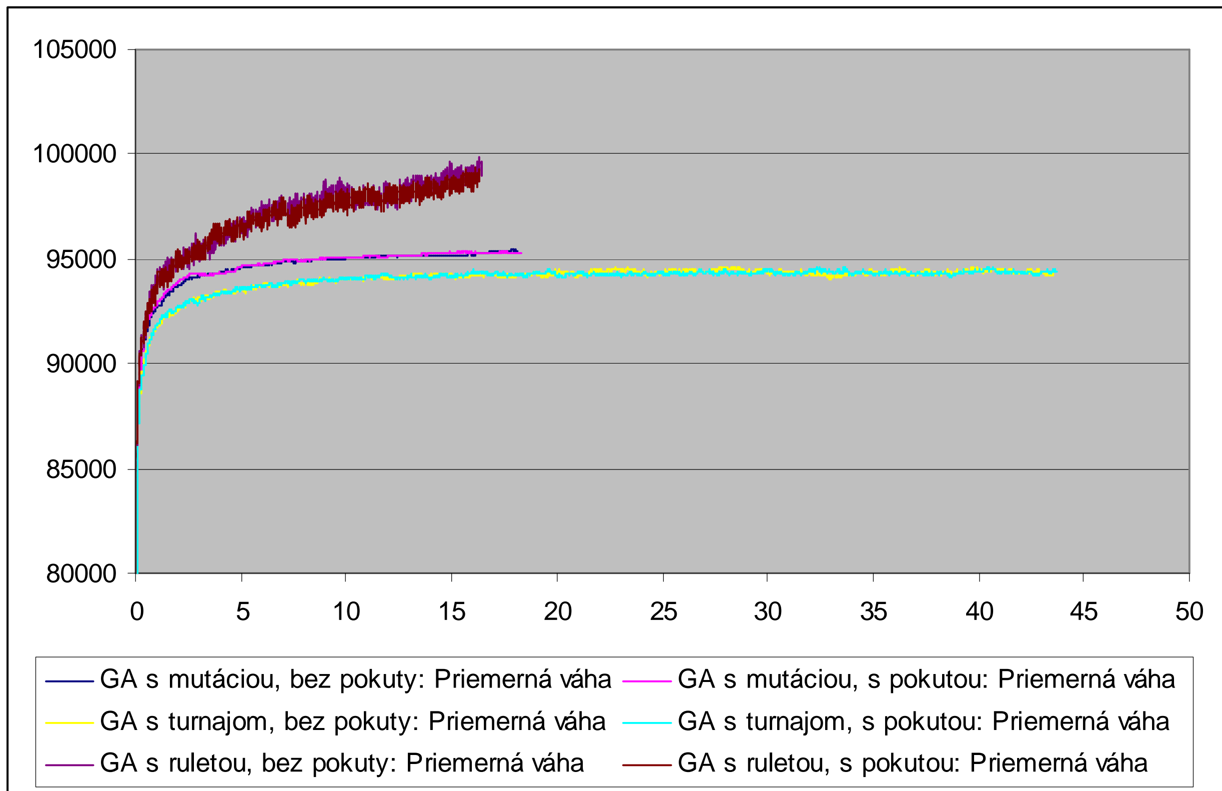
Graf 16 - Porovnanie času trvania podľa počtu generácií pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



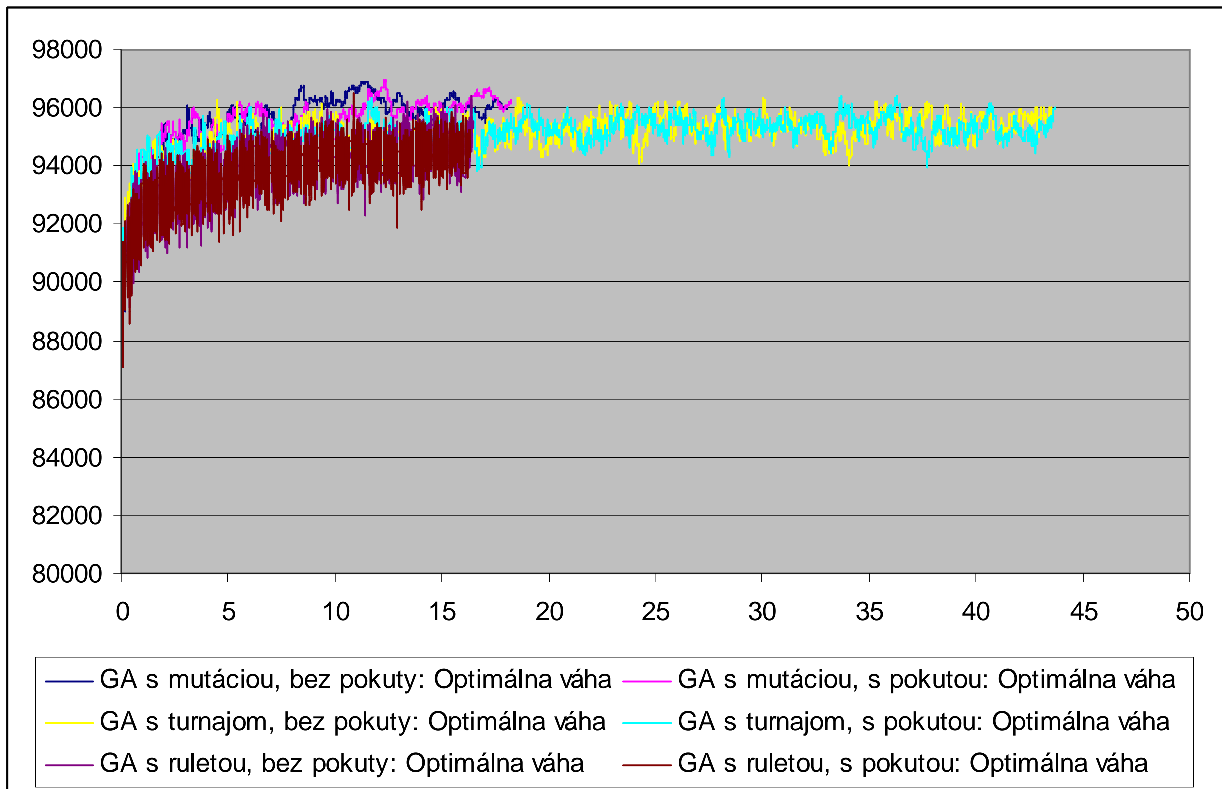
Graf 17 - Porovnanie počtu generácií v závislosti od počtu ohodnotení pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



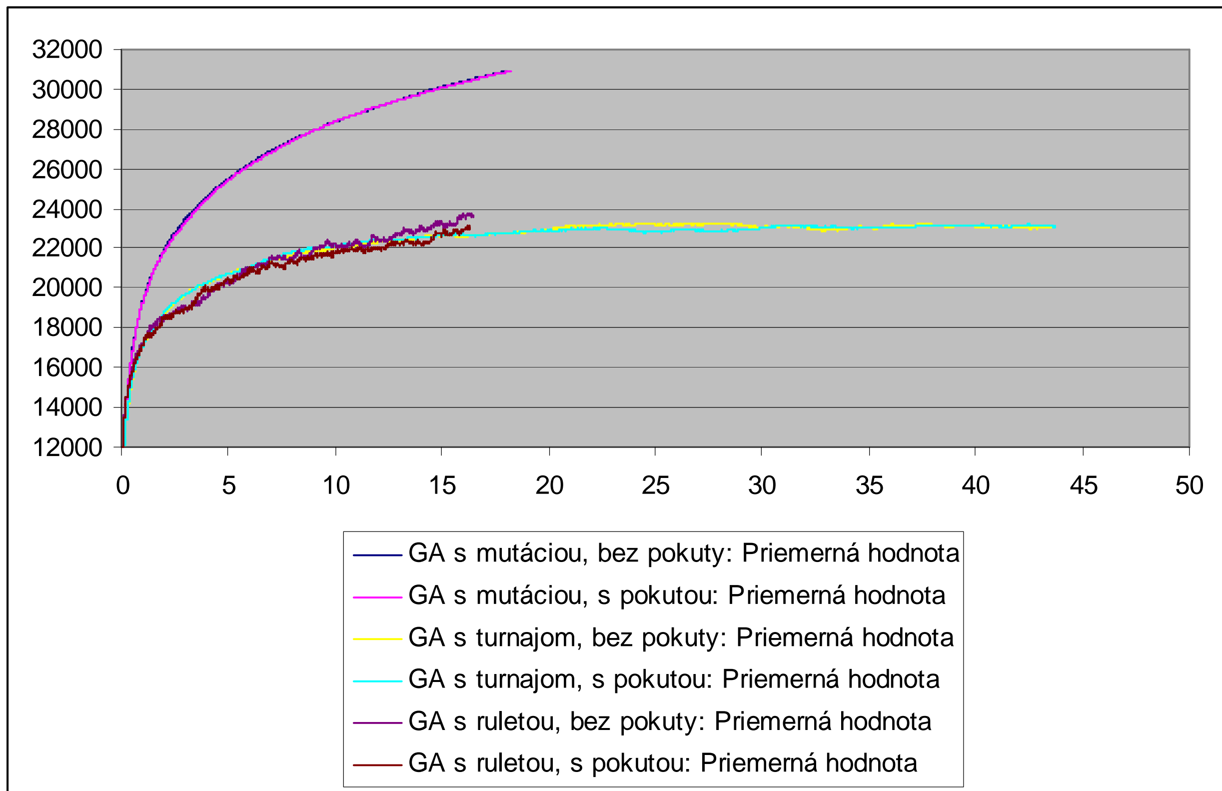
Graf 18 - Porovnanie počtu generácií v závislosti od uplynutého času pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



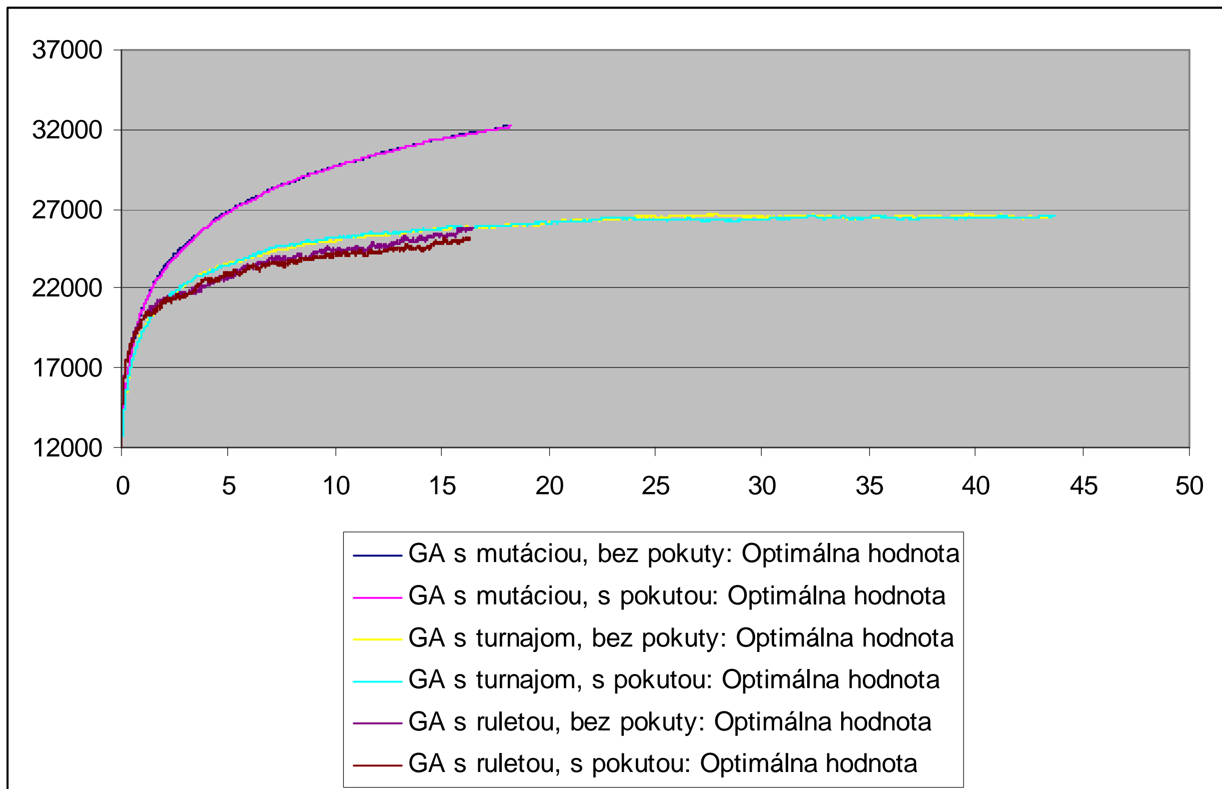
Graf 19 - Porovnanie vývoja priemernej váhy v čase pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



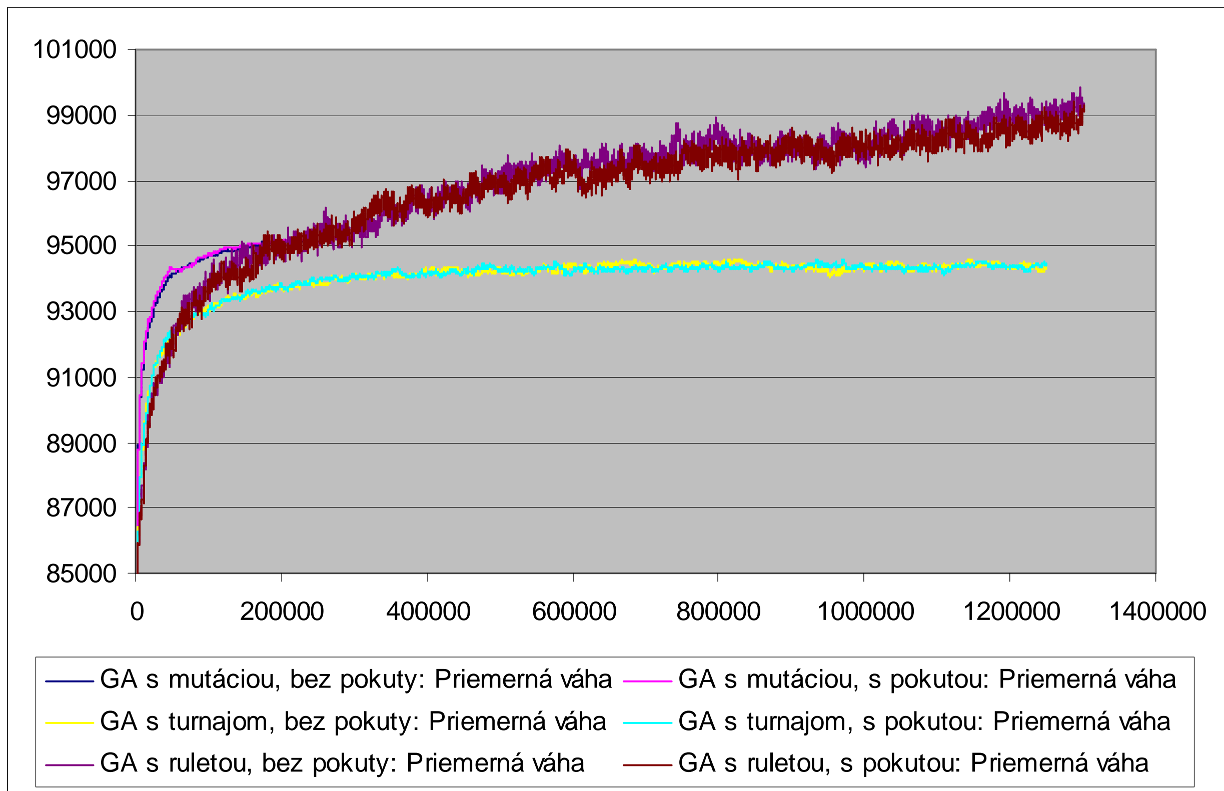
Graf 20 - Porovnanie vývoja váhy optimálneho jedinca v čase pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



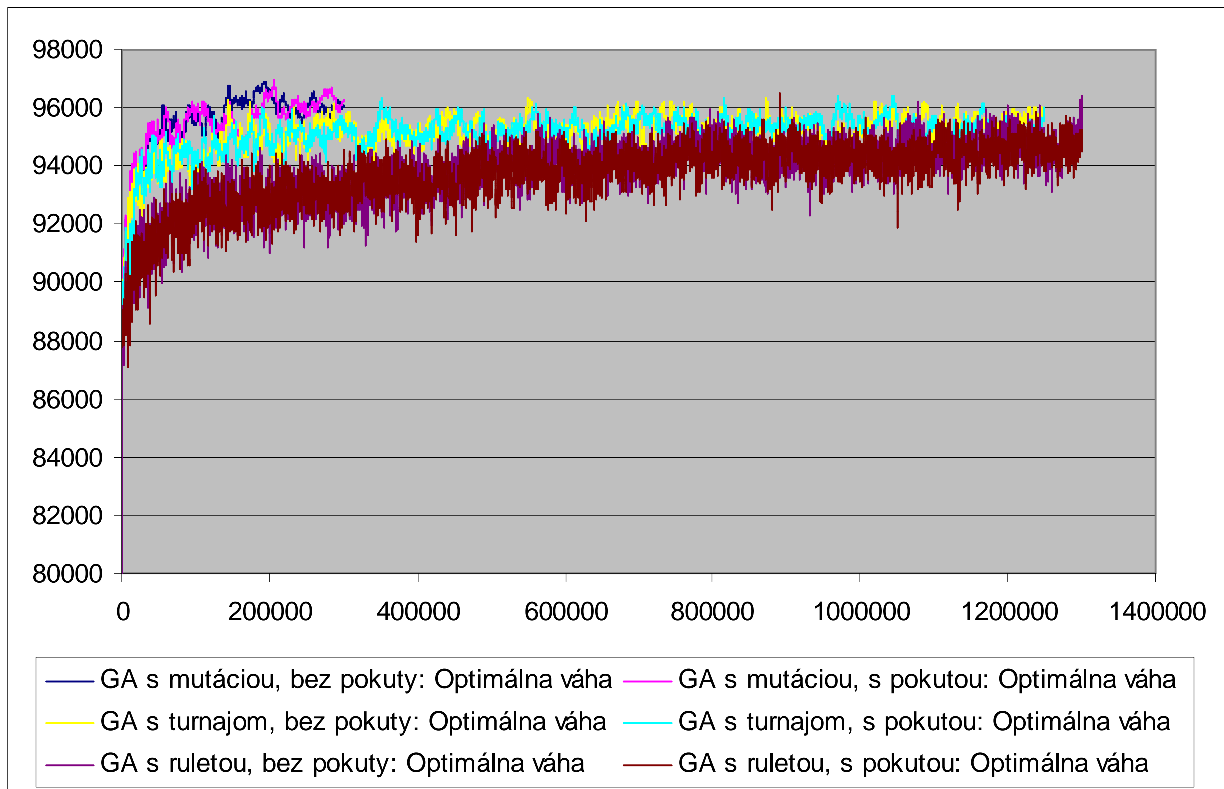
Graf 21 - Porovnanie vývoja priemernej hodnoty v čase pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



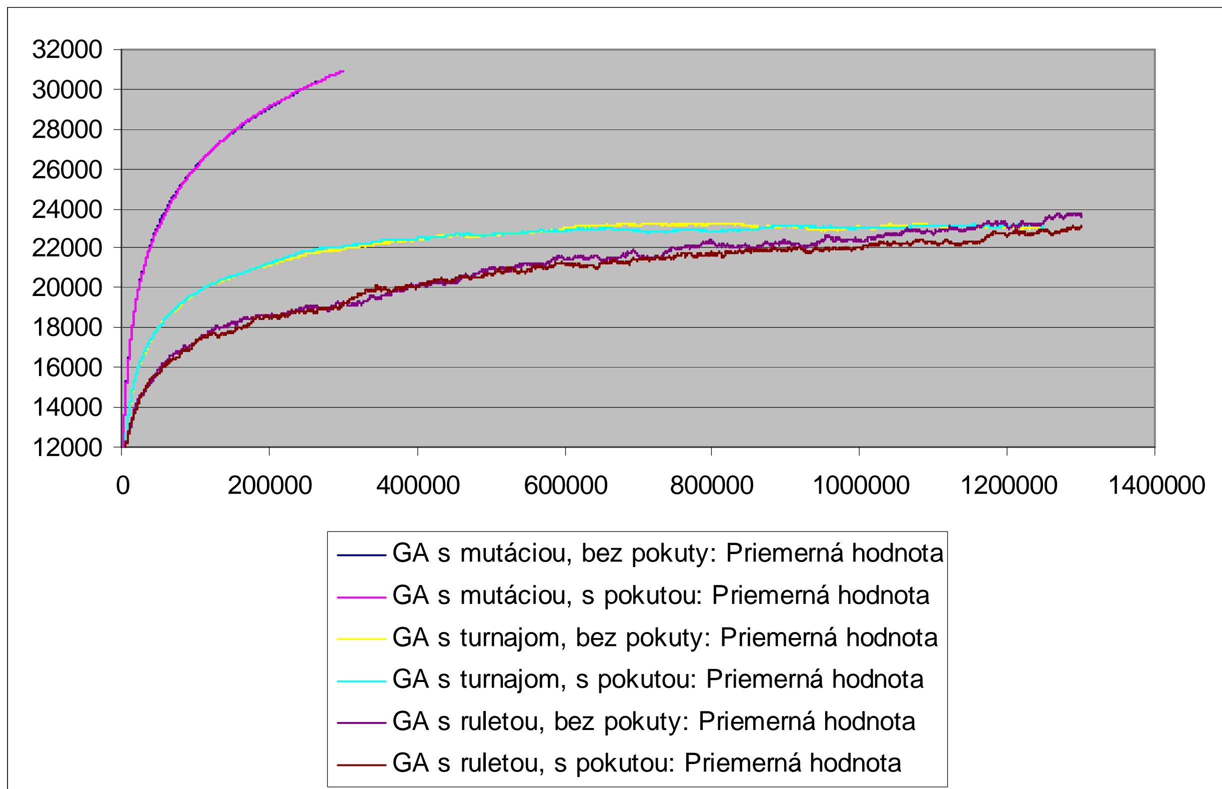
Graf 22 - Porovnanie vývoja hodnoty optimálneho jedinca v čase pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



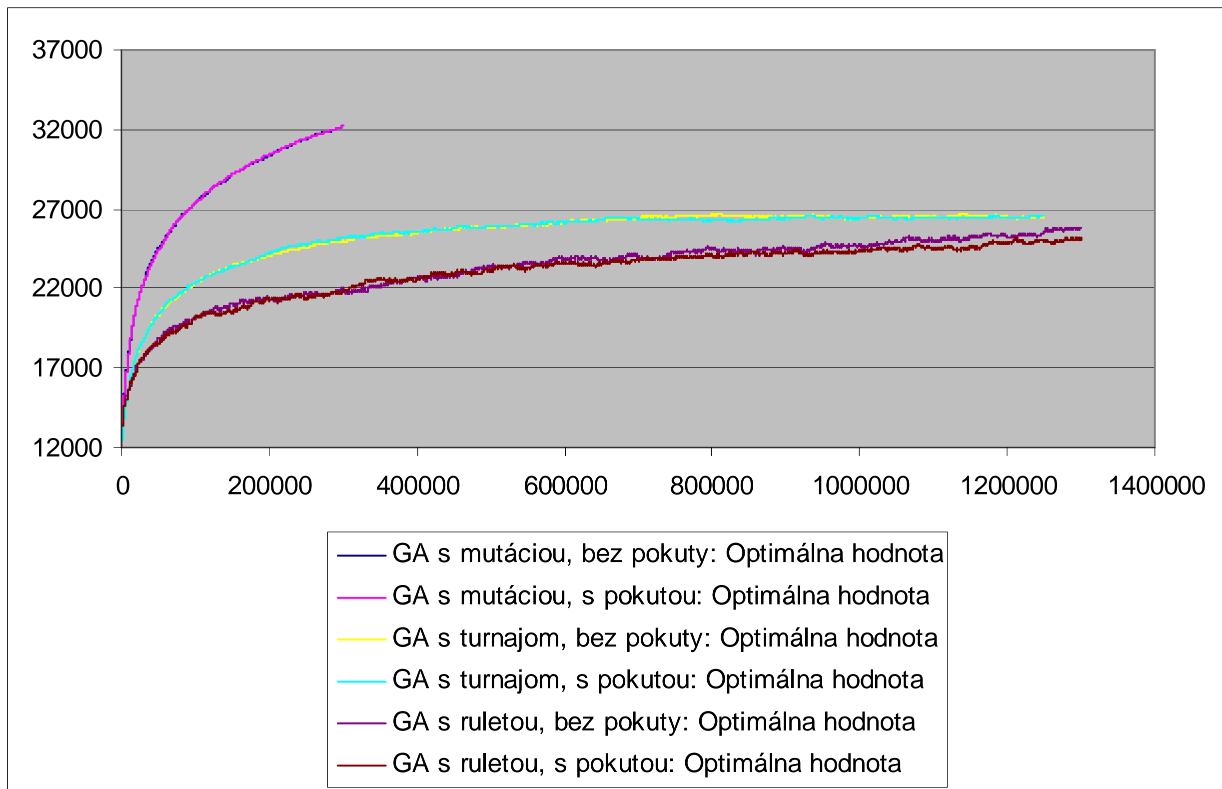
Graf 23 - Porovnanie vývoja priemernej váhy v závislosti od počtu ohodnotení pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



Graf 24 - Porovnanie vývoja váhy optimálneho jedinca v závislosti od počtu ohodnotení pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



Graf 25 - Porovnanie vývoja priemernej hodnoty v závislosti od počtu ohodnotení pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)



Graf 26 - Porovnanie vývoja hodnoty optimálneho jedinca v závislosti od počtu ohodnotení pre všetky kombinácie algoritmov s a bez použitia pokutovacej funkcie (priemer zo 100 behov)

Zhodnotenie výsledkov

V nameraných a zobrazených dátach môžeme sledovať niekoľko zaujímavých trendov.

Predovšetkým, pokiaľ ide o porovnanie jednotlivých algoritmov navzájom (tabuľka 1, grafy 1 až 4), ukazuje sa, že najlepšie výsledky dosahuje algoritmus s použitím mutácie, ktorý už v 1000. generácií prekročil hodnotu 30000, pričom algoritmus s turnajom dosiahol druhé najlepšie hodnoty a algoritmus s použitím výberu pomocou rulety sa javí ako keby dosahoval najhoršie výsledky. Chcel by som však podotknúť, že vzhľadom na rozdielnu výpočtovú zložitosť jednotlivých algoritmov je takáto interpretácia mierne zavádzajúca.

Jeden beh pre 1000 generácií algoritmu s mutáciou trvá priemerne 18,58 sekundy, jeden beh algoritmu s použitím turnaja trvá v priemere iba 8,46 sekundy a jeden beh algoritmu s použitím výberu pomocou rulety trvá priemerne dokonca iba 1,38 sekundy (tieto údaje boli zistené meraním priemeru zo 100 behov jednotlivých algoritmov na počítači s procesorom typu Intel Pentium M 1,7GHz s 512MB RAM). Z toho vyplýva, že na dávku 100 behov po 1000 generáciách bolo treba zhruba 30 minút, zatiaľ čo algoritmus s turnajom dokázal za 70 minút vykonať 100 behov po 5000 generáciách a algoritmus s použitím rulety za niečo viac ako 20 minút zvládol 100 behov po 10000 generáciách.

Na grafe 15 vidíme vývoj počtu ohodnotení v závislosti od generácie, na grafe 17 naopak porovnanie počtu generácií od počtu ohodnotení. Podobne, na grafe 16 je porovnaný uplynutý čas podľa počtu generácií a na grafe 18 porovnanie počtu generácií v čase. Môžem skonštatovať, že najmenší počet ohodnotení dosahuje algoritmus s použitím rulety, za ním s pomerne veľkým odstupom algoritmus s turnajom a o málo horší bol algoritmus s použitím mutácie, ktorého počet ohodnotení rastie konštantne. Algoritmus s ruletou bol aj najrýchlejší. V čase keď algoritmus s použitím mutácie dosiahol 1000. generáciu, algoritmus s turnajom bol práve v 2000. generácií, kdežto algoritmus s použitím rulety dosiahol v tom istom čase už 10000. generáciu. Rýchlosť algoritmu s ruletou je určite ovplyvnená aj spomínanou implementáciou binárneho vyhľadávania pri kvázi náhodnom výbere jedincov namiesto klasického sekvenčného hľadania.

Rýchlosť, ale dovolím si tvrdiť, že aj výsledky algoritmu s výberom pomocou rulety sú zároveň do istej miery ovplyvnené aj tým, že pravdepodobnosť kríženia bola iba 30%, pričom tento parameter sa aplikoval iba pri tomto algoritme, pri ostatných bola reprodukcia vykonávaná náhodne na všetkých jedincoch (pri turnaji bol výber počtu jedincov vstupujúci do reprodukcie náhodný od 1 do počtu jedincov v celej generácií, pri algoritme s mutáciou vstupovali do reprodukčného procesu vždy všetci jedinci).

Okrem toho, hodnota pri použití algoritmu s mutáciou pomerne prudko stúpa oproti ostatným algoritmom (porovnaj grafy 7 a 8), aj keď váha je v 1000. generácií pre algoritmus mutácie o niečo nižšia ako pre ostatné algoritmy (porovnaj graf 5, resp. 6), takže je istý predpoklad, že hodnota môže ďalej rásť, čo však platí aj u ostatných dvoch algoritmov, ktorých hodnoty majú stály rastúci trend.

Uvedené môžeme pozorovať aj na porovnaní vývoja priemerných váh a hodnôt a váh a hodnôt optimálneho jedinca podľa času (grafy 19 až 22) ako aj v závislosti od počtu ohodnotení (grafy 23 až 26), pričom vidíme, že počet ohodnotení nemá až taký rozhodujúci vplyv na rýchlosť (algoritmus s ruletou vykazuje pomerne vysoký počet ohodnotení rovnako aj algoritmus s turnajom, ale algoritmus s ruletou je výrazne rýchlejší). Tu zrejme zohráva svoju rolu implementácia spomínaného binárneho vyhľadávania pri rulete, resp. možno menšia optimalizácia pri turnaji.

Pri porovnaní jednotlivých parametrov sa ukazuje, že použitie pokutovacej funkcie nemá očakávaný význam. Priemerné váhy aj priemerné hodnoty ako aj váhy a hodnoty optimálnych jedincov majú pri porovnaní tohto parametra zhruba rovnaký trend (pozri grafy 9 až 14).

Záver

V tomto semestrálnom projekte som navrhol a implementoval program na porovnanie rôznych genetických algoritmov aplikovaných na optimalizáciu pakovacieho problému. Tento dokument zahŕňa aj prípadovú štúdiu, v ktorej som experimentálne porovnal jednotlivé algoritmy a použitie, resp. nepoužitie pokutovacej funkcie.

Aj napriek pomernej rozsiahlosti aplikácie (vyše 2200 riadkov zdrojových kódov bez komentárov – na tie naozaj nezvýšil čas) verím, že by sa na nej ešte dalo mnohé vylepšiť. Ako príklad uvediem vyššiu optimalizáciu niektorých algoritmov alebo rozsiahlejší a pestrejší zber štatistík, prípadne inú implementáciu pokutovacej funkcie. Navyše, ako som už spomenul vyššie, ďalšie množstvo práce by mohlo byť odvedené pri „hraní sa“ s touto aplikáciou a zisťovaní, ako a ktoré parametre vplývajú na jednotlivé optimalizačné algoritmy a aké sú ich vhodné kombinácie nastavenia.

Prácou na projekte som strávil značné množstvo času (vyše 1 mesiac a zhruba 80-95 hodín čistého času), ale poskytol mi možnosť získania hlbších poznatkov jednak o princípoch genetických algoritmov praktickou cestou ako aj získania nových poznatkov v oblasti programovacích techník (použitie vlákien a niektorých prvkov prostredia .NET). Dovolím si tvrdiť, že tento projekt mal pre mňa jednoznačný prínos a zároveň si dovolím dúfať, že sám bude prínosom pre ďalších.

Použitá literatúra a odkazy na zdroje

1. EPIC GA Course: „Excercise 1: Bin Packing", http://www.epcc.ed.ac.uk/epic/ga/intro.html (15.5.2006)
2. EPIC GA Course: „Excercise 4: Bin Packing revisited", http://www.epcc.ed.ac.uk/epic/ga/ optimisation.html (15.5.2006)
3. Kvasnička, V., Pospíchal, J., Tiňo, P.: Evolučné algoritmy, STU Bratislava, 2000
4. Microsoft Development Network Library: Threading Tutorial, Microsoft Corporation, 2006, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkThreadingTutorial.asp (12.6.2006)