VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ ŘEŠENÍ PROBLÉMU CESTUJÍCÍHO ZLODĚJE

EVOLUTIONARY APPROACH TO THE TRAVELING THIEF PROBLEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DÁVID FODOR**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

145375

Ústav: Ústav počítačových systémů (UPSY)
Student: **Fodor Dávid**
Program: Informační technologie
Specializace: Informační technologie

Název: **Evoluční řešení problému cestujícího zloděje**
Kategorie: Umělá inteligence
Akademický rok: 2022/23

Zadání:

1. Seznamte se s evolučními algoritmy a s problémem cestujícího zloděje (Traveling Thief Problem (TTP)), metodami jeho řešení a testovacími instancemi.
2. Navrhněte způsob řešení TTP, který bude využívat principů evolučních algoritmů.
3. Implementujte navržené řešení, včetně alespoň dvou variant genetických operátorů.
4. Navržené řešení experimentálně ověřte na zvolených instancích problému TTP, a to pro různá nastavení evolučního algoritmu.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 31.10.2022

Abstrakt

Táto práca predstavuje návrh evolučného algoritmu na riešenie problému cestujúceho zlodeja (TTP), ktorý sa skladá z dvoch vzájomne prepojených podproblémov, problému cestujúceho obchodníka (TSP) a problému batohu (KP). Návrh obsahuje viacero variácií navrhovaného algoritmu. Je založený na genetickom algoritme, evolučnom algoritme (1+1) a ich kombinácii. Algoritmus je implementovaný a testovaný na oficiálnych testovacích inštanciách TTP. Najlepšia navrhnutá varianta evolučného algoritmu bola porovnaná s náhodným vyhľadávaním a najlepšími verejne nájdenými riešeniami pre testované inštancie.

Abstract

This thesis presents design of an evolutionary algorithm for solving the Traveling thief problem (TTP), which is composed of two interconnected subproblems, the traveling salesperson problem (TSP) and the knapsack problem (KP). The proposed algorithm contains multiple variations of evolutionary algorithm. It is based on the genetic algorithm, the evolutionary algorithm (1+1), and their combination. The algorithm is implemented and tested on official TTP benchmark instances. The best variation of the proposed evolutionary algorithm is chosen and compared with random search and the best publicly available solutions for tested problem instances.

Kľúčové slová

Problém cestujúceho zlodeja, evolučné algoritmy, genetický algoritmus, optimalizácia

Keywords

Traveling thief problem, evolutionary algorithms, genetic algorithm, optimization

Citácia

FODOR, Dávid. *Evoluční řešení problému cestujícího zloděje*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Evoluční řešení problému cestujícího zloděje

Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Dávid Fodor
9. mája 2023

Poďakovanie

Ďakujem vedúcemu práce, ktorým je pán prof. Ing. Lukáš Sekanina, Ph.D za jeho odbornú pomoc, konzultácie a prístup.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

S optimalizačnými problémami sa stretávame v rôznych oblastiach ako logistika, financie, dizajn systémov či plánovanie udalostí a procesov. Podstatou optimalizačných problémov je, že sa snažíme nájsť najlepšie vyhovujúce riešenie splnujúce dané podmienky. Mnoho dôležitých skúmaných optimalizačných problémov spadá do kategórie NP-ťažkých[2]. Tieto problémy sa vyznačujú vysokou výpočtovou náročnosťou, nepoznáme pre ne algoritmus, ktorý by ich dokázal vyriešiť v polynomiálnom čase pre všetky možné inštancie problému. Použitie hrubej sily k prehladaniu celej množiny možných riešení problému je možné len pre malé vstupy. Čas prehladávania rastie exponenciálne s veľkosťou vstupu a výpočtová náročnosť tak rýchlo presiahne naše možnosti. Pomáhame si používaním rôznych heuristík k hľadaniu dobrých riešení, ktoré nemusia byť optimálne.
Problémom, ktorým sa budem zaoberať v tejto práci, je problém cestujúceho zlodeja (angl. Traveling Thief Problem). Jedná sa o pomerne nový problém, vytvorený s cieľom priblížiť sa viac problémom reálneho sveta[3]. A to tak, že spojením dvoch prepojených podproblémov bude napodobňovať komplexnosť reálnych problémov. Pre tento účel si vybrali autori dva dobre známe a skúmané NP-ťažké problémy. Problém cestujúceho zlodeja je spojenie problému obchodného cestujúceho a problému batohu. Hlavnou postavou je zlodej hľadajúci trasu, po ktorej prejde všetky mestá, a zoznam predmetov, ktoré naberie po ceste do svojho batohu. Prepojenie problémov je realizované pomocou ceny prenájmu batohu na jednotku času, ktorú musí zlodej zaplatiť. A meniacej sa rýchlosti zlodeja v závislosti od váhy v batohu[1]. Hľadá sa riešenie, ktoré prinesie najväčší zisk pre zlodeja, ktorý dostaneme keď z hodnoty získaných predmetov odčítame výsledný prenájom za batoh. Tento problém si rýchlo získal popularitu a vzniklo niekoľko štúdií testujúcich svoje algoritmy na probléme cestujúceho zlodeja[12]. Vznikla aj databáza benchmarkových príkladov, na ktorých si rôzne riešenia môžme otestovať.
Mojim cieľom je použiť evolučné algoritmy k hľadaniu suboptimálneho riešenia tohto problému. Jedná sa o výpočtové metódy inšpirované procesmi v prírode. Pre ich schopnosť vyhľadávať dobré riešenia nad veľkými množinami možných riešení, sú často voľbou pri NP-ťažkých problémoch. Moje riešenie je založené na genetickom algoritme a evolučnom algoritme (1+1). Genetický algoritmus vyplýva z Darwinovej evolučnej teórie, využíva niekoľko genetických operátorov. Riešenia sa reprezentujú ako jedinci v populácií, uplatňuje sa na nich selekcia, kríženie a mutácia. Podľa typu problému a jeho zakódovania existuje viacero variant ako implementovať tieto operátory[5]. Evolučný algoritmus (1+1) je výrazne jednoduchší a vykonáva evolúciu nad jediným jedincom. Výsledný algoritmus bude môcť bežať vo viacerých nastaveniach a kombináciach. V tejto práci budem testovať a porovnávať medzi sebou rôzne nastavenia môjho algoritmu, za účelom nájsť jeho najlepšie pracujúcu

variantu.
V kapitole súčasný stav poznania, opisujem problém cestujúceho zlodeja, uvádzam jeho definíciu, a teoretický úvod do evolučných algoritmov. Nasleduje kapitola s návrhom implementácií algoritmu k riešeniu tohto problému, a kapitola implementácia obsahujúca stručný opis výsledného programu. Ďalej uvádzam testovanie nastavení algoritmu a nasledovné zhodnotenie výsledkov.

Kapitola 2

Súčasný stav poznania

Táto kapitola sa zaoberá problémom cestujúceho zlodeja, jeho predstavením, popisom jeho podproblémov a definíciou. Ďalej budem pokračovať úvodom do témy evolučných algoritmov, a detailnejším popisom pre genetický algoritmus a jeho operátory.

2.1 Problém cestujúceho zlodeja

Tento problém bol predstavený v štúdií[3], publikovanej v roku 2013, (angl. Travelling Thief Problem) ďalej v texte ho budem označovať skratkou TTP. Autori uvádzajú ako motiváciu pre vytvorenie nového problému priepasť, ktorá vzniká medzi problémami reálneho sveta a klasickými benchmarkovými problémami. Komplexnosť problémov v reálnom svete rastie rýchlo, kým benchmarkové problémy, na ktorých sa testujú algoritmy, ostávajú rovnaké už dlhšiu dobu. Autori navrhujú ako jeden z hlavných rozdielov v komplexnosti medzi reálnym svetom a benchmarkovými problémami kombináciu viacerých prepojených podproblémov v reálnom svete. Riešenie jedného podproblému je tak závislé na ostatných a nedokážeme ho jednoducho izolovať a riešiť samostatne. Autori sa tak rozhodli vytvoriť nový problém simulujúci takéto vzájomné prepojenie podproblémov. Pre tento účel si zvolili dva veľmi dobre známe optimalizačné problémy, jedná sa o problém obchodného cestujúceho a problém batohu.

2.1.1 Problém obchodného cestujúceho

Problém obchodného cestujúceho je jeden z najznámejších problémov v optimalizácií (angl. Traveling Salesman Problem), ďalej budem označovať skratkou TSP. Patrí do kategórie NP-úplných problémov. Formulovaný bol už v roku 1930, odvtedy bolo na ňom otestovaných viacero optimalizačných metód. Problém kladie otázku: Ak je daný zoznam miest a vzdialenosť medzi každým párom miest, aká je najkratšia možná cesta cez všetky možné mestá, tak aby sme každé navštívili iba raz a na konci sa vrátili do počiatočného mesta?

Riešenia tohto problému majú viaceré aplikácie v reálnom svete, jeden z najznámejších príkladov je rozvoz zásielok zo skladu ku zákazníkom. Okrem logistiky sa dajú využiť pri rôznych plánovacích úlohách a tiež dizajne, ako pri návrhu prepojení v elektronických obvodoch. Aby sme sa s istotou dostali k presnému riešeniu TSP, bolo by potrebné použiť hrubú silu a prehľadať všetky možnosti, pre n miest existuje však $(n-1)!/2$ možností. Preto sa využívajú rôzne aproximačné algoritmy. Známe spôsoby, ako riešiť TSP pomocou algoritmov inšpirovaných prírodou je optimalizácia mravenčou kolóniou, genetické algoritmy a simulované žíhanie[4].

2.1.2 Problém batohu

Druhý podproblém je takisto veľmi známi a z kategórie NP-úplných problémov, problém batohu (angl. Knapsack Problem), ktorý budem označovať skratkou KP. Otázkou, ktorú riešime v KP, je: Máme daný batoh s maximálnym limitom nosnosti a zoznamom predmetov. Každý má dané dve hodnoty, jeho cenu a váhu. Aké predmety máme zabaliť do batohu tak, aby sme dostali súčtom ich hodnôt maximálnu možnú celkovú cenu a zároveň neprekročili jeho maximálnu nosnosť? Riešenia tohto problému majú dobré uplatnenie pri rozdeľovaní rôznych zdrojov projektom na základe ich priority alebo hodnoty. Taktiež pri investíciách, kde sa berie do úvahy očakávaný návrat a limituje nás rozpočet, prípadne riziká investíc. K riešeniu tohoto problému sa často využíva dynamické programovanie alebo rôzne metaheuristiky ako geneticky algoritmus, simulované žíhanie, tabu prehľadávanie[11].

2.1.3 TTP definícia

Problém TTP bol vytvorený prepojeným spomínaných problémov do jedného. Existujú rôzne definície problému, ktoré sa líšia v určitých detailoch. Ja budem pracovať s najčastejšie sa vyskytujúcou pôvodnou definíciou označenú v práci *The travelling thief problem: The first step in the transition from theoretical problems to realistic problems* ako TTP1[3].

Definícia problému cestujúceho zlodeja

Zlodej sa vydáva na cestu, má daný zoznam miest, v ktorých sa musí zastaviť, tiež pozná všetky vzdialenosti medzi nimi. Má daný zoznam predmetov, tie môže počas svojej cesty ukradnúť. Každý predmet má pridelenú hodnotu, váhu a mesto výskytu. Cestujúci zlodej musí navštíviť každé mesto presne raz a vrátiť sa do počiatočného mesta. Postupne počas svojej cesty pridáva vybrané predmety do svojho batohu. Batoh má ale určenú maximálnu nosnosť, tú zlodej nesmie prekročiť, batoh je tiež prenajímaný, má stanovenú konštantnú cenu za jednotku času. S pridávaním predmetov do batohu a zvyšovaním váhy nákladu, klesá rýchlosť zlodeja, a teda rastie čas cesty.

Cieľom je nájsť vhodnú trasu a plán vyzvedávania predmetov pre zlodeja. Výsledný zisk zlodeja musí byť, čo najväčší. Dostaneme ho súčtom hodnôt vybraných predmetov mínus stanovený prenájom za batoh na celý čas trvania cesty.

Množina miest $X = \{1, 2, ..., n\}$

Vzdialenosť d_{ij} vzdialenosť medzi mestami i a j

Predmety $I = \{I_1, I_2, ..., I_m\}$

Hodnota predmetu p_k hodnota predmetu k

Hmotnosť predmetu w_k hmotnosť predmetu k

Nosnosť batohu C

Prenájom R cena prenájmu batohu na jednotku času

Rýchlosť v_{min}, v_{max} minimálna, maximálna rýchlosť zlodeja

Plán cesty $\bar{x} = (x_1, ...x_n)$ vektor s idexami miest v poradí plánu cesty

Plán predmetov $\bar{y} = (y_1, ...y_m)$ binárny vektor určujúci, ktoré predmety boli vybrané, ak $y_i = 1$ predmet s indexom i bol vybraný

Obmedzujúca podmienka hmotnosť vybraných predmetov nesmie prekročiť nosnosť batohu

$$\sum_{i=1}^{m} w_i * y_i \leq C$$

Zmena rýchlosti po vložení predmetu k do batohu sa rýchlosť zlodeja zmení nasledovne

$$\Delta v = \frac{w_k}{C} * (v_{max} - v_{min})$$

$$v_{po} = v_{pred} - \Delta v$$

Čas cesty trvanie cesty medzi mestami i a j

$$t_{ij} = \frac{d_{ij}}{v}$$

Účelová funkcia/fitnes funkcia Zisk z vybraných predmetov, mínus prenájom batohu vynásobený celkovým časom cesty

$$f = \sum_{i=1}^{m} p_i * y_i - R * (t_{x_n,x_1} + \sum_{i=1}^{n-1} t_{x_i,x_{i+1}}) \tag{2.1}$$

2.2 Evolučné algoritmy

Evolučné výpočty čerpajú svoje základy z biológie. Inšpirujú sa prácou Charlesa Darwina popisujúcou evolúciu, prirodzenou selekciou v prírode, správaním a fungovaním rôznych organizmov či fyzikálnymi zákonmi. Evolučné výpočty môžme rozdeliť ešte na viaceré podkategórie s rôznymi prístupmi k hľadaniu riešenia problému[7].

Genetické algoritmy Najznámejší základný evolučný algoritmus inšpirovaný prirodzenou selekciou v prírode. Možné riešenia problému, jedinci, sú reprezentovaný napríklad ako celočíselné vektory. Ako moc kvalitné dané riešenie je reprezentujeme hodnotou zvanou fitness. Na začiatku sa vytvára počiatočná populácia skladajúca sa z niekoľkých jedincov, pomocou genetických operátorov ako selekcia, kríženie a mutácia simulujeme prirodzenú selekciu a evolúciu. Algoritmus cyklicky vytvára nových jedincov, kandidátne riešenia problému, ktoré ohodnocuje a snaží sa maximalizovať ich fitnes.[7].

Genetické programovanie Jedná sa o metódu generovania počítačových programov riešiacich daný problém. Princíp je veľmi podobný klasickému genetickému algoritmu, avšak jedincov v tomto prípade netvoria vektory optimalizovaných parametrov, ale celé programy. Program sa reprezentuje v stromovej štruktúre obsahujúcej funkcie, premenné a konštanty. Vhodnosť programu reprezentujeme opäť hodnotou fitness, určujúcou jeho schopnosť riešiť daný problém. Ďalšie kroky pozostávajú opäť z vytvorenia počiatočnej populácie, evaluácie a aplikácie genetických operátorov. Kríženie a mutácia sa vykonávajú nad štruktúrou stromu a vytvárajú tak nové programy, s cieľom nájsť dostatočne funkčný program.[7].

Evolučné stratégie Evolučné stratégie využívajú na reprezentovanie riešení špecificky vektor reálných čísel a kvalita sa ohodnocuje znovu pomocou fitness. Rozdiel oproti genetickému algoritmu je, že hlavnú úlohu tu zohráva mutácia namiesto kríženia. Mutáciu môžme vykonať pričítaním náhodne generovaného vektora v určitom rozmedzí. Ak takýto krok zlepší fitness riešenia, nahradíme ho a proces môžme opakovať. Algoritmus využíva autoadaptačné mechanizmy[6].

Swarm inteligence Jedná sa o skupinu algoritmov výraznejšie sa odlišujúcu od predošlých kategórií. Fungujúca na princípe interakcie jednotlivcov v prostredí. Spravanie jednotlivcov je stochastické, ovplyvnené vnímaním svojho okolia. Spadá sem napríklad optimalizácia hejnom častíc (angl. particle swarm optimization), inšpirovaná chovaním hejn vtákov pri hľadaní potravy. Alebo optimalizácia mravenčou kolóniou (angl. ant colony optimization) simulujúca spôsob hľadania potravy mravcami, napodobňujúca ich komunikáciu pomocou feromónovej stopy[15].

2.2.1 Genetický algoritmus

Základom môjho riešenia bude klasický genetický algoritmus, je jedným z najpoužívanejších evolučných algoritmov so širokým uplatnením. K opísaniu algoritmu používame terminológiu z biológie, uvedieme si definície jednotlivých prvkov algoritmu[7].

Populácia - množina kandidujúcich riešení.

Jedinec - jedno z kandidujúcich riešení, skladá sa z génov, hodnôt definujúcich konkrétne riešenie.

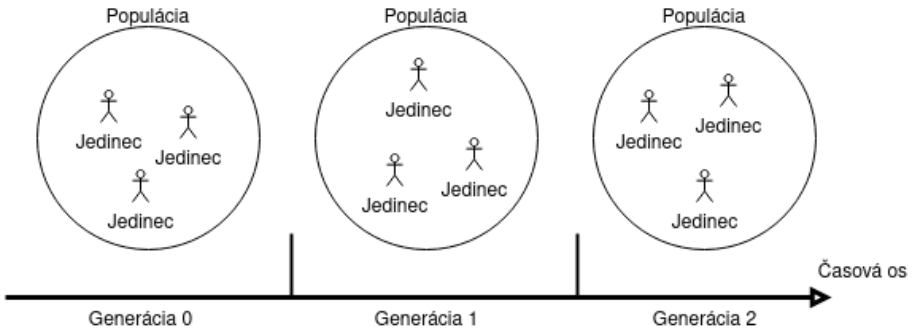
Evaluácia - proces, pri ktorom ohodnocujeme jedinca, zisťujeme ako dobré dané riešenie je. Výsledné hodnotenie sa nazýva fitnes jedinca a počítame ho cez fitness funkciu.

Selekcia - proces vyberania jedincov na základe ich fitnes, vybraní jedinci sa stanú rodičmi, zúčastnia sa reprodukcie, prenesenia svojich génov na nových jedincov.

Kríženie - proces rekombinácie génov rodičov a následné vytvorenie nových potomkov.

Mutácia - náhodná zmena nejakého z génov jedinca vykonávaná s určenou pravdepodobnosťou.

Generácie - po vykonaní všetkých krokov algoritmu, aktuálna populácia sa mení, pridávajú sa do nej novo vytvorení jedinci a odstraňujú sa starý, podľa návrhu všetci alebo len niektorý neselektovaný. Každým takýmto cyklom nám vznikne nová generácia jedincov, ktorú môžme označiť poradovým číslom.

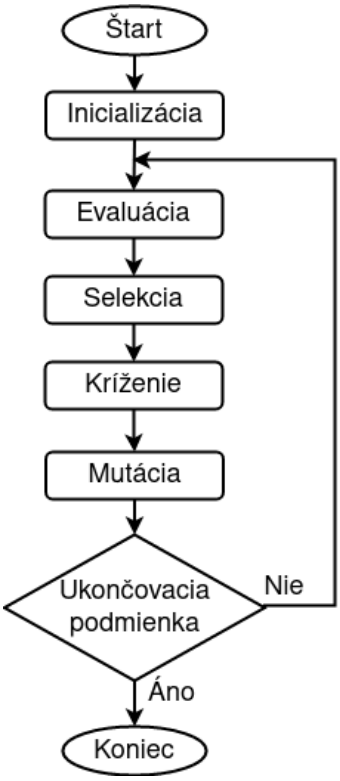


Obr. 2.1: Grafické zobrazenie termínov generácia,populácia a jedinec

Prvým krokom genetického algoritmu je **inicializácia**. Jedná sa o vytvorenie prvotnej populácie, vytvárame tak jedincov reprezentujúcich nejaké riešenia z množiny kandidátnych riešení. Na základe definície problému je potrebné definovať štruktúru jedinca. Podľa toho, koľko premenných potrebujeme na popis riešenia, bude jeho štruktúra obsahovať také množstvo génov. Potom môžme vygenerovať počiatočnú populáciu jedincov, ktorá má dôležitý dopad na efektívnosť GA. Jedinci sú generovaný väčšinou náhodne, je vhodné zabezpečiť ich rovnomerne rozloženie v priestore možných riešení. Veľkosť potrebnej počiatočnej populácie je závislá od komplexnosti problému a veľkosti prehľadávajúceho priestoru. Evaluáciou následne ohodnotíme jedincov, vypočítame ich fitness hodnotu.

Na hodnotenie jednotlivých riešení používame **fitnes funkciu**, ktorá nám udáva ako je dobré dané riešenie, a pre genetický algoritmus určuje pravdepodobnosť daného jedinca na reprodukciu.

Akonáhle algroitmus má počiatočnú populáciu, môže sa začať proces vyhľadávania dobrých riešení. Beh algoritmu je založený na spolupráci troch genetických operátorov, tie bežia v cykle až pokiaľ nedôjde k ukončeniu prehľadávania. Operátory sme už viackrát spomínali, prejdeme k ich podrobnejšiemu popisu.



Obr. 2.2: Diagram genetického algoritmu

2.2.2 Selekcia

Účelom selekcie je výber najlepších jedincov, ktorí podľa pravidiel evolúcie v prírode dostanú šancu prežiť a reprodukovať sa. Selekciu môžme rozdeliť do dvoch krokov a vybrať zvlášť rodičov, z ktorých sa vytvoria noví potomkovia. A potom zo všetkých vrátane nových jedincov vybrať tých, ktorí prežijú do ďalšej generácie. Ide o proces vyberania istej podmnožiny jedincov na základe ich fitnes. K tomuto procesu môžme pristupovať viacerými metódami. K výberu môžme pristúpiť deterministicky, kedy asi najzákladnejším spôsobom je použitie selekcie označovanej anglickým názvom *truncation selection* alebo tiež názvom elitárstvo. Populácia sa zoradí na základe ich fitness a vyberie sa jednoducho n najlepších. Takýto prístup však nemusí byť vždy vhodný, ďalšou možnosťou je pristúpiť k výberu stochastickým spôsobom. Princíp fungovania takejto metódy je, že každý jedinec má istú pravdepodobnosť výberu. Jedinci s lepším fitnes majú väčšiu pravdepodobnosť výberu, avšak nemajú výber zaručený. Výhoda takejto metódy je pridanie „šumu“ do procesu výberu, čo môže pomôcť pri vyhýbaní sa konvergencie do lokálneho optima[6][5].

Ruletový výber (rulet wheel selection) Ako vyplýva z názvu, je to metóda inšpirovaná ruletou, narozdiel od klasickej rulety, kedy každé číslo padá s rovnakou pravdepodobnosťou, pravdepodobnosť výberu jedinca je závislá od jeho fitnes hodnoty. Jedinci si medzi sebou rozdelia pomyselné ruletové koleso, čím lepšie je ich fitnes, tím väčšiu časť kolesa budú zaberať. Pravdepodobnosti výberu získame z relatívnej hodnoty fitness. Vypočítame celkový súčet všetkých fitnes hodnôt jedincov, fitnes jedinca vydelíme celkovým fitnes, výsledkom je jeho relatívne fitnes[7].

Rankový výber (rank selection) Tento typ selekcie zoraďuje jedincov od najhoršieho po najlepšieho podľa ich fitnes. Riešenia sú potom ohodnotené svojim poradovým číslom od 1 po N. Ďalej pravdepodobnosť výberu závisí na ich poradovom čísle reprezentujúcom ich fitnes. Následne sa nad jedincami najčastejšie vykoná ruletová selekcia. Pravdepodobnosť výberu dostaneme sčítaním všetkých poradových čísel a vydelením poradia jedinca ich celkovým súčtom. Tento prístup má svoje výhody v tom, že funguje bez úprav aj pri negatívnych fitnes hodnotách a prináša väčšiu náhodnosť do selekcie. Potláča tak problém zo strácajúcou sa diverzitou populácie. Môže však predĺžiť proces konvergencie a je výpočtovo náročnejší[15].

Turnajový výber (tournament selection) Výber sa uskutočňuje pomocou viacerých turnajov, tie môžu obsahovať dvoch a viacej jedincov. Jedinci sú do turnajov vybraní náhodne, vyhráva ten s najlepším fitnes. Víťazi sú selektovaný, turnaje sa tak vykonávajú dokedy nenazbierame určené množstvo selektovaných jedincov. Tento prístup je jednoduchý, výpočtovo efektívny a funguje pri negatívnych fitnes hodnotách. Je potrebné však nájsť vhodnú veľkosť turnajov pre daný problém a populáciu[13].

Elitárstvo (elitism) Jednoduchý koncept, ktorý je možné využívať s rôznymi spôsobmi selekcie. Jedná sa o automatickú selekciu stanoveného počtu najlepších jedincov na začiatku selekcie. Tak zabránime ich možnej strate pri stochastických metódach[15].

2.2.3 Kríženie

Operátor kríženia má na starosti rekombináciu jedincov za účelom vytvorenia nového potomka. Implementácia závisí od reprezentácie riešenia problému, jedinca. Vo väčšine prípadov je jedinec reprezentovaný reťazcom. Operácia potom spočíva v pospájaní vybraných úsekov reťazcov do nového reťazca. Kríženie môže byť jedno alebo viac bodové. Prebiehať bude tak, že sa reťazce rozdelia na segmenty v náhodne určenom bode alebo bodoch. Potom sa striedavo berú segmenty rodočov a postupne sa skladajú do nového reťazca. Tak vzniká jeden alebo dvaja potomci, ktorí nesú informácie získané od rodičov[4].

2.2.4 Mutácia

Operácia vykonáva úpravu nad jedným jedincom. Ide o zmenu génov jedinca náhodne a v určenom rozsahu. Dá sa použiť ako reprodukčný operátor v prípade klonovania jedného jedinca na upraveného potomka. Alebo sa použije ako doplnok kríženia k pridaniu novej informácie novo vzniknutému potomkovy. Úprava pomocou mutácie zahrňuje dva aspekty. Prvý je koľko génov jedinca bude zmenených. Druhý, aké veľké budú zmeny, nazývame tiež veľkosť kroku alebo vzdialenosť medzi riešeniami[15].

Kapitola 3

Návrh riešenia

Ďalej popíšem návrh môjho algoritmu pre riešenie TTP. Návrh obsahuje popis formátu vstupného zadania s ktorým bude algoritmus pracovať, výber spôsobu akým bude riešenie problému zakódované do chromozómu. Popis výpočtu fitnes hodnoty riešenia a návrh implementácie genetických operátorov. Následne sa tu uvádza ako budú tieto jednotlivé časti spojené do výsledného algoritmu, jeho rôznych nastavení. A popis, ako dané návrhy otestujem, porovnám medzi sebou.

3.1 Vstup

Zadanie konkrétneho TTP obsahuje pomerne veľa informácií, definuje sa zoznam miest, zoznam predmetov s parametrami a viacero ďalších hodnôt. Existuje databáza benchmarkových zadaní TTP, kde každý príklad je uložený zvlášť v súbore s definovanou štruktúrou. Navrhovaný algoritmus bude pracovať s týmito súbormi a ich formátom zadania. V článku[12], na ktorom sa podieľali aj autori TTP, sa opisuje spôsob vytvárania týchto príkladov. Autori využili existujúcu databázu príkladov pre problém obchodného cestujúceho, vygenerovali k zoznamu miest zoznam predmetov a doplnili zvyšné hodnoty.

Príklad vstupného súboru zo zadaním vyzerá nasledovne:

zadanie.ttp

```
PROBLEM NAME:  zadanie_1
KNAPSACK DATA TYPE: typ_1
DIMENSION: 4
NUMBER OF ITEMS:  5
CAPACITY OF KNAPSACK:  10
MIN SPEED:  0.1
MAX SPEED:  1
RENTING RATIO:  2
EDGE_WEIGHT_TYPE: CEIL_2D
NODE_COORD_SECTION (INDEX, X, Y):
1 2 2
2 8 2
3 8 8
4 2 8
```

```
ITEMS SECTION (INDEX, PROFIT, WEIGHT, ASSIGNED NODE NUMBER):
1 10 8 2
2 15 7 2
3 22 1 3
4 50 6 3
5 100 9 4
```

Algoritmus z takto formátovaného súboru načíta a uloží hodnoty parametrov zadania. K parametrom uvádzam vysvetlenia:

PROBLEM NAME - názov zadania.

KNAPSACK DATA TYPE - typ zoznamu predmetov, autori zadania tu zvyknú uvádzať, či hodnoty predmetov boli generované náhodne, bez kolerácie medzi sebou alebo naopak bolo použité nejaké špecifické generovanie.

DIMENSION - počet miest v danom zadaní.

NUMBER OF ITEMS - počet predmetov.

CAPACITY OF KNAPSACK - veľkosť kapacity batohu.

MIN SPEED - minimálna rýchlosť pohybu zlodeja za jednotku času.

MAX SPEED - maximálna rýchlosť pohybu zlodeja za jednotku času.

RENTING RATIO - cena prenájmu batohu na jednotku času

EDGE_WEIGHT_TYPE - typ hodnotenia vzdialenosti medzi mestami (2D, 3D priestor)

NODE_COORD_SECTION (INDEX, X, Y) - určuje začiatok zoznamu miest, vždy na novom riadku bude jeden záznam mesta zložený z troch hodnôt. Vzájomne oddelené medzerami bude číslom definovaný index mesta, jeho X a Y súradnice.

ITEMS SECTION (INDEX, PROFIT, WEIGHT, ASSIGNED NODE NUMBER) - určuje začiatok zoznamu predmetov, vždy na novom riadku bude jeden záznam predmetu zložený zo štyroch hodnôt. Odelené medzerami bude definovaný index predmetu, jeho hodnota, váha, index mesta kde sa predmet nachádza.

Vytváraná implementácia, ako vstup požaduje súbor v uvedenom formáte. Mestá aj predmety sa identifikujú pomocou ich indexov. Indexy sú celé čísla od 1 vzostupne a nesmú sa opakovať. Každý predmet má svoj originálny index, a tak sa môže nachádzať vždy iba v jednom meste. Algoritmus bude pracovať len s mestami v 2D priestore so súradnicami X a Y, pretože dostupné benchmarkové zadania sú tak vytvorené.

Spomínaná databáza zadaní je uverejnená na stránkach[1] univerzity v Adelaide (The University of Adelaide). Odtiaľ budem čerpať všetky vstupné testovacie súbory.

[1] https://cs.adelaide.edu.au/õptlog/CEC2014COMP_InstancesNew/

3.2 Kódovanie

Dôležitým počiatočným krokom pri evolučných algoritmoch je zvolenie vhodnej reprezentácie riešenia problému. Riešenia sa zakóduje do reťazca, ten označujeme ako chromozón alebo jedinec. Najpoužívanejším kódovaním je binárny reťazec, kedy každý gén je jeden bit, nula alebo jedna. Populárny je tiež Grayov kód, podobný binárnemu s rozdielom reprezentácie celých čísel binárnou hodnotou tak, aby susedné čísla sa odlišovali zmenou len jedného bitu. Na niektoré úlohy, kde je dôležité poradie prvkov používame reprezentáciu pomocou permutácie. Reprezentácia riešenia ovplyvňuje efektívnosť algoritmu, určuje nám štruktúru prehľadávacieho priestoru, má vplyv na následnú výpočtovú náročnosť. Ovplyvňuje možnosti implementácie genetických operátorov, ktoré pracujú s definovaným kódovaním a musia s ním vedieť jednoducho manipulovať.

TTP sa skladá z dvoch podproblémov, preto budeme reprezentovať riešenie plánu cesty po mestách a zároveň plán naberania predmetov. Výsledná reprezentácia bude spojenie dvoch reťazcov do jedného chromozónu, jedinca. Pre rozdielnosť týchto podproblémov, budem ich reprezentovať rozdielnym kódovaním a genetické operátory budú musieť byť implementované pre každé kódovanie rozdielne.

Reprezentácia plánu cesty, je prevzatá z TSP podproblému. Kódovaním musíme zachytiť také poradie miest z daného zoznamu, v akom budú postupne navštevované. Binárna či Grayova reprezentácia takéhoto problému je príliš neprirodzená. Mestá sú identifikované indexmi, celými číslami. Pri operáciách manipulujúcich s bitmi by vznikali rovnaké čísla na viacerých miestach cesty, iné čísla by zase chýbali. Podmienkov riešenia problému je navštíviť každé mesto a každé mesto len jeden krát, následne sa vrátiť do počiatočného mesta. Binárna reprezentácia by vyžadovala veľké množstvo opráv jednotlivých riešení, čo by bolo komplikované z hľadiska implementácie a tiež výpočtovo náročné riešenie.

Najprirodzenejší spôsob je **reprezentovanie pomocou permutácie**, riešením by tak bol reťazec indexov a tie by boli zoradené v poradí plánu cesty. Reťazec musí obsahovať všetky indexy miest, ale nesmú sa opakovať.

Plán cesty:

$$3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$$

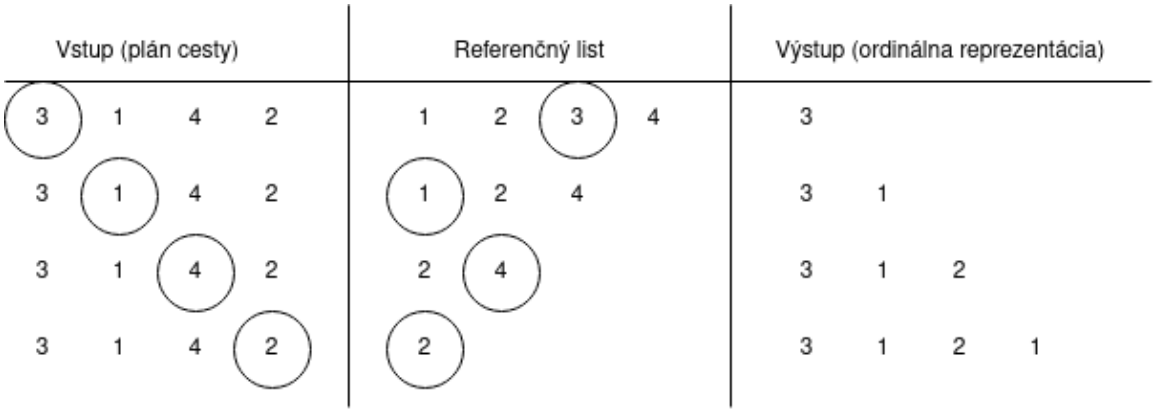
Reprezentácia permutáciou:

$$(3, 1, 4, 2)$$

Z tejto reprezentácie plánu cesty je priamo viditeľná plánovaná trasa, dobre sa s ňou pracuje pri počítaní fitness funkcie, tiež existuje pre ňu jednoduchá implementácia mutácie či lokálneho prehľadávania. Problematickejšia je pri krížení, klasické prepojenie vybraných častí permutácie od rodičov do novej opäť vedie k výskytu opakujúcich sa a chýbajúcich indexov. Pre jej prirodzenosť a fakt, že je často využívaná k reprezentácií riešenia TSP, budem ju využívať k zakódovaniu plánu cesty v mojom algoritme.

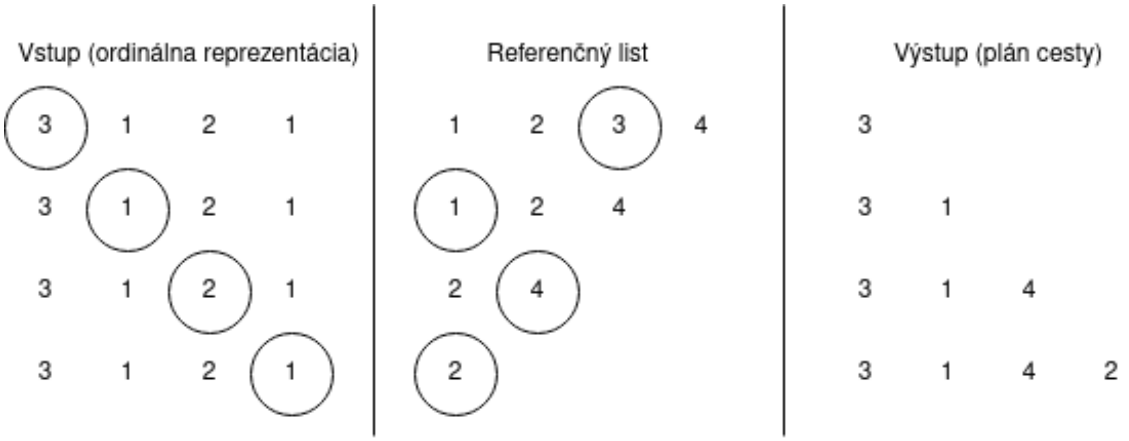
Ďalšou zaujímavou reprezentáciou vhodnou pre TSP je **ordinálna reprezentácia** (angl. ordinal representation). Táto reprezentácia umožňuje jednoduché kríženie bez porušenia permutácie. Z tohto dôvodu ju použijem k implementácií viacbodového kríženia[4]. Z tejto reprezentácie však nejde na prvý pohľad zistiť plán cesty a k výpočtu fitnes je potrebné ju dekódovať naspäť napríklad do podoby spomínanej permutácie. Viac detailov uvediem pri opise implementácie operátorov kríženia. K zakódovaniu do ordinálnej reprezentácie je potrebný referenčný zoznam indexov, určujúci ich poradie. Proces zakódovania funguje tak, že vyberieme prvý index z nášho plánu cesty, pozrieme sa na akom mieste sa

nachádza v referenčnom zozname. Odstránime ho z referenčného zoznamu a jeho poradové číslo, na ktorom sa nachádzal, vložíme na výstup. Zoberieme druhý index z plánu cesty, nájdeme ho v referenčnom zozname, odstránime ho z neho a jeho poradové číslo prilepíme na výstup. Takto pokračujeme až do konca[4].



Obr. 3.1: Prevod plánu cesty na ordinálnu reprezentáciu

Pri dekódovaní budeme postupne vyberať z referenčnej tabuľky indexy na poradovom mieste podľa hodnoty ordinálneho kódu. Indexy vkladáme opäť na výstup. Spätné dekódovanie na plán cesty je na obrázku 3.2.



Obr. 3.2: Prevod ordinálnej reprezentácie na plán cesty

Reprezentácia naberania predmetov je spôsob akým sa zakóduje riešenie druhého podproblému KP. Je potrebné zachytiť, ktoré predmety boli počas cesty pridané do batohu. Tento problém sa dá reprezentovať jednoducho ako binárny vektor[15]. Dĺžka vektoru bude rovnaká ako počet predmetov, ktoré existujú v danom zadaní. A každá binárna hodnota bude reprezentovať stav jedného predmetu: 0 znamená predmet nieje zbalený a 1 znamená predmet bude zbalený do batohu. Takáto reprezentácia umožňuje jednoduché mutácie aj kríženie, napriek tomu vďaka obmedzenej kapacite batohu budú pri týchto operáciách vznikať nevalidné riešenia. Pri zmene vektoru je potrebné skontrolovať podmienku kapacity a nevyhovujúce riešenia opraviť alebo následne penalizovať v fitnes hodnote. V mojom algoritme použijem binárny vektor na zakódovanie plánu vyzvedávania predmetov a nevyhovujúce riešenia budem opravovať odoberaním predmetov, kým podmienka kapacity batohu nebude splnená.

Ak by zadanie obsahovalo 6 predmetov s indexami od 1 po 6, a konkrétne riešenie by do batohu zbalilo predmety 2, 3 a 6. Toto riešenie sa zakóduje do binárneho vektoru nasledovne:

$$(0, 1, 1, 0, 0, 1)$$

3.3 Fitnes funkcia

V optimalizačných problémoch je cieľom nájsť maximum alebo minimum matematickej funkcie určujúcej kvalitu riešenia problému. Nájdenie jej globálneho maxima alebo minima podľa definície problému znamená nájdenie vhodných vstupných argumentov k vyriešeniu problému. Takáto funkcia sa nazýva účelová, na jej základe sa vytvorí fitnes funkcia, ktorá bude určovať kvalitu jedincov v evolučných algoritmoch. Fitnes hodnota potom reprezentuje schopnosť jedinca prežiť a zúčastniť sa na procese kríženia. TTP hľadá riešenie, ktoré prinesie najväčší možný zisk zlodejovi. Účelová funkcia tohto problému je výpočet celkovej získanej čiastky zlodejom na konci jeho cesty, pričom sa zlodej môže dostať aj do záporných čísel, dlhu. Vzorec(2.1) je uvedený v TTP definíci. Hľadáme globálne maximum tejto funkcie, teda funkciu maximalizujeme. Túto účelovú funkciu som sa rozhodol použiť bez úprav priamo ako funkciu fitnes, veľkosť zisku zlodeja, bude určovať hodnotu fitnes riešenia v mojom algoritme.

Proces výpočtu fitnes hodnoty riešenia bude prebiehať nasledovne:

1. Na začiatku po načítaní vstupných dát sa vypočítajú vzájomné vzdialenosti medzi všetkým mestami. Vypočítané hodnoty sa uložia do matice vzdialeností k neskoršiemu použitiu. Mestá sa nachádzajú v 2D priestore s pridelenými súradnicami, vzdialenosť vypočítame pomocou Pytagorovej vety. Vzorec(3.1) ukazuje výpočet pre mestá s indexmi i a j, súradnicami X a Y.

$$d_{i,j} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2} \tag{3.1}$$

2. Pred výpočtom konkrétnej fitnes hodnoty sa prevedie kontrola podmienky kapacity batohu. Sčítajú sa hmotnosti zbalených predmetov, a ak došlo k prekročeniu povolenej kapacity vykoná sa oprava riešenia.

3. Aktuálna rýchlosť zlodeja v sa nastaví na maximálnu rýchlosť v_{max}, hmotnosť W a cena P v batohu sú na začiatku nulové. Taktiež celkový čas cesty $T = 0$.

4. V prvom meste sa na začiatku neberú žiadne predmety, vypočíta sa čas trvania cesty do druhého mesta $t_{1,2}$, vzorec(3.2). A pripočíta sa k celkovému času cesty T, vzorec(3.3).

$$t_{1,2} = \frac{d_{1,2}}{v} \tag{3.2}$$

$$T = T + t_{1,2} \tag{3.3}$$

5. Ďalej sa cyklicky prechádza z druhého až do posledného mesta a nakoniec z posledného naspäť do prvého. Ak sa v aktuálnom meste naberajú nejaké predmety do batohu, aktualizuje sa celková hmotnosť W a cena P v batohu. Vtedy je potrebné vypočítať novú rýchlosť zlodeja v.

$$v = v_{max} - ((\frac{W}{kapacita}) * (v_{max} - v_{min})) \tag{3.4}$$

   Potom sa môže prejsť do ďalšieho mesta, vypočíta sa čas cesty $t_{i,j}$ a aktualizuje sa celkový čas T. Podobne ako vo vzorcoch(3.2)(3.3).

6. Na konci cesty ak existujú predmety v počiatočnom meste, ktoré sa berú do batohu, pripočíta sa ich hmotnosť a cena. Vypočíta sa celkový zisk zlodeja, od celkovej ceny P v batohu sa odčíta celkový čas T vynásobený prenájmom.

$$zisk = P - (prenajom * T) \tag{3.5}$$

   Vypočítaný zisk sa rovná fitnes hodnote riešenia, jedinca.

3.4 Genetické operátory

3.4.1 Selekcia

Implementovať selekciu pre navrhovaný algoritmus je možné viacero spôsobmi. K vybraniu vhodného z nich implemntujem tri spôsoby selekcie a budem ich porovnávať v rámci testovania.

Výber najlepších N jedincov - najzákladnejšia implementácia selekcie, populácia sa zoradí od jedincov s najväčším fitness po najmenšie. Vyberieme požadovaný počet najlepších jedincov.

Ranková selekcia - jedinci sa zoradia od najmenšieho fitness po najväčšie, jeden najlepší jedinec, posledný v zozname, bude vybraný automaticky vždy na začiatku. Ďalej sa implementuje priradenie čísiel ranku a z nich výpočet pravdepodobností výberu. Rank je poradové číslo v zoradenom zozname jedincov začínajúci od 1. Pravdepodobnosť výberu daného jedinca je jeho rank vydelený celkovým súčtom všetkých rankov. Náhodným výberom jedincov s použitím vypočítaných pravdepodobností ako váh jednotlivých možností, sa vyberie požadovaný počet z populácie.

Turnajová selekcia, použije sa tiež s elitizmom, najlepší jedinec bude automaticky selektovaný na začiatku. Výber jedincov do turnaja bude náhodný pre všetkých s rovnakou pravdepodobnosťou. V turnaji bude selektovaný jeden víťaz s najlepším fitnes z účastníkov. Vhodná veľkosť turnaja bude závisieť od veľkosti populácie pri testovaní. Proces turnaja sa bude iterovať až kým bude selektované požadované množstvo jedincov.

3.4.2 Kríženie

Kríženie bude implementované nad dvoma vybranými jedincami a produktom budú dva nový potomci. Operácia kríženia sa vykoná na oboch reprezentáciách podproblémov rozdielne. Pri **reprezentácií plánu cesty (TSP)** bola zvolena implementácia viacbodového kríženia pomocou prekódovania do spomínanej ordinálnej reprezentácie.
Návrh implementácie bude nasledovný:

1. Prekódovanie rodiča 1 a rodiča 2 z reprezentácie permutáciou do ordinálnej reprezentácie.

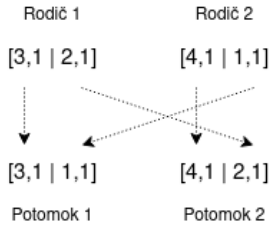
   Rodič 1
   $$(3, 1, 4, 2) \rightarrow [3, 1, 2, 1]$$
   Rodič 2
   $$(4, 1, 2, 3) \rightarrow [4, 1, 1, 1]$$

2. Vygenerujú sa náhodné body kríženia. Generovať sa bude najskôr počet bodov kríženia v danom rozmedzí podľa veľkosti zadania problému. Potom sa vygenerujú náhodne miesta kríženia v reťazci.

3. Vytvoria sa dva nové prázdne reťazce pre potomkov, do prvého potomka sa skopíruje časť reťazca od začiatku rodiča 1 po prvý bod kríženia, druhému potomkovi sa pridelí

časť od rodiča 2. Ďalšia časť od aktuálneho bodu kríženia po ďalší, prípadne po koniec reťazca, sa pridelí prvému potomkovi od rodiča 2 a druhému potomkovi od rodiča 1. Takto sa postupným skladaním častí od rodičov vzniknú kompletný potomkovia.

Pre ilustračný príklad sa kríženie vykoná s jedným bodom kríženia medzi druhým a tretím prvkom.



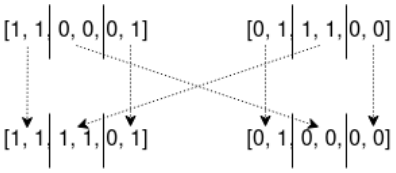
Obr. 3.3: Príklad jednobodového kríženia

4. Vzniknutý potomkovia sa prekódujú naspäť na permutáciu reprezentujúcu cestu zlodeja.

   Potomok 1
   $$[3, 1, 1, 1] \rightarrow (3, 1, 2, 4)$$
   Potomok 2
   $$[4, 1, 2, 1] \rightarrow (4, 1, 3, 2)$$

Výhodou tohto prístupu ku krížení permutácií je, že nedôjde k vzniku opakujúcim sa indexov, a teda porušeniu pravidiel pre plán cesty. Po prekódovaní do ordinálnej reprezentácie je samotné kríženie už veľmi jednoduché.

Implementácia kríženia pre **podproblém batohu (KP)** je o to jednoduchšia, že sa jedná o binárny vektor. Použijem aj v tomto prípade viacbodové kríženie, na začiatku sa vygeneruje náhodné množstvo bodov kríženia v stanovenom rozsahu. Vygenerujú sa pozície bodov a poskladajú sa nový jedinci.



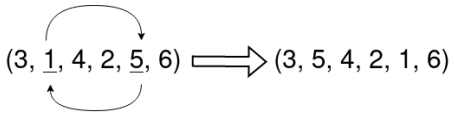
Obr. 3.4: Príklad viacbodového kríženia nad binárnym vektorom

Celkové kríženie a vznik nových jedincov pre TTP sa vykoná po krížením plánu cesty a plánu naberania predmetov ich rodičov. Novovzniknuté riešenia podproblémov sa spoja, vzniknú dva nový jedinci.

3.4.3 Mutácia

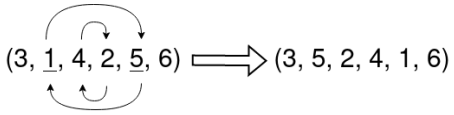
Mutácia vykonáva náhodnú zmenu chromozómu a vnáša nové informácie do prehľadávania. Spôsob jej implementácie bude takisto rozdielny u reprezentácií cesty a reprezentácií nabe-

rania predmetov.
Mutácia pre reprezentáciu **TSP** podproblemu sa vykoná zamenením dvoch indexov, miest, v permutácií. Náhodne sa vygenerujú dve pozície v rámci cesty, mestá na týchto pozíciach si zamenia poradie v akom ich zlodej navštívi.



Obr. 3.5: Príklad mutácie TSP podproblemu prehodením indexov

Implementujem aj druhý podobný spôsob mutácie pre TSP navrhovaný v článku[10], jedná sa o mutáciu reverznou sekvenciou, označuje sa skratkou **RSM** (z angl. Reverse Sequence Mutation). Opäť sa náhodne vyberú dva body v permutácií. Poradie indexov v sekvencií medzi týmito dvoma bodmi pretočí.



Obr. 3.6: Príklad mutácie TSP podproblemu reverznou sekvenciou

Mutácia podproblému **KP** bude taktiež veľmi jednoduchá, pretože sa jedná sa o zmenu (negáciu), jedného binárneho čísla z vektoru reprezentujúceho plán naberania predmetov. Miesto na ktorom dôjde ku mutácií sa vyberá náhodne, zmena spôsobí pribalenie alebo odstránenie predmetu na danom indexe z batohu.

Celkový operátor mutácie bude pracovať s jedincom nasledovne. Mutácia sa môže vykonávať s určenou pravdepodobnosťou. Táto hodnota je v rozmedzí 0 až 1, je nastavená pri spúštaní algoritmu. Generuje sa náhodné číslo z tohto rozmedzia a ak je väčšie, ako stanovená pravdepodobnosť mutácie, vykoná sa mutácia na danom jedincovi. Inak sa pokračuje ďalej a jedinec ostáva bez zmeny. Mutácia, ak prebehne, vykoná sa iba jeden krát u jedinca, buď v pláne cesty alebo pláne naberania predmetov. Čo sa taktiež rozhodne náhodne. Ak je teda hodnota pravdepodobnosti mutácie 0.5, je 50% šanca, že sa u jedinca vykoná jedna náhodná zmena informácie.

3.5 Evolučný algoritmus (1+1)

Najjednoduchšia varianta evolučného algoritmu označovaná skratkou **(1+1)EA** vykonáva evolúciu nad jedným jedincom a vytvára jedného potomka. Jedinec je reprezentovaný reťazcom a potomok sa z neho vytvorí náhodnou zmenou, mutáciou jedného alebo viacerých jeho génov. Vypočíta sa hodnota fitnes nového jedinca, ak je jeho fitnes väčšia od fitnes rodiča, potomok ho nahradí a rodič sa zahodí. V opačnom prípade ostáva rodič a zahodí sa potomok. Tento proces sa dookola opakuje, prevedie sa ďalšia náhodná zmena informácie nad jedincom, a ponechá sa iba, ak bolo dosiahnuté lepšie riešenie[16].

Pre TTP bude návrh implementácie (1+1)EA nasledovný. Jedna iterácia prebehne nad jedincom s danou veľkosťou kroku, vzniká jeden nový jedinec. Náhodne sa rozhodne pre úpravu buď plánu cesty alebo plánu naberania predmetov. Potom sa na danom reťazci,

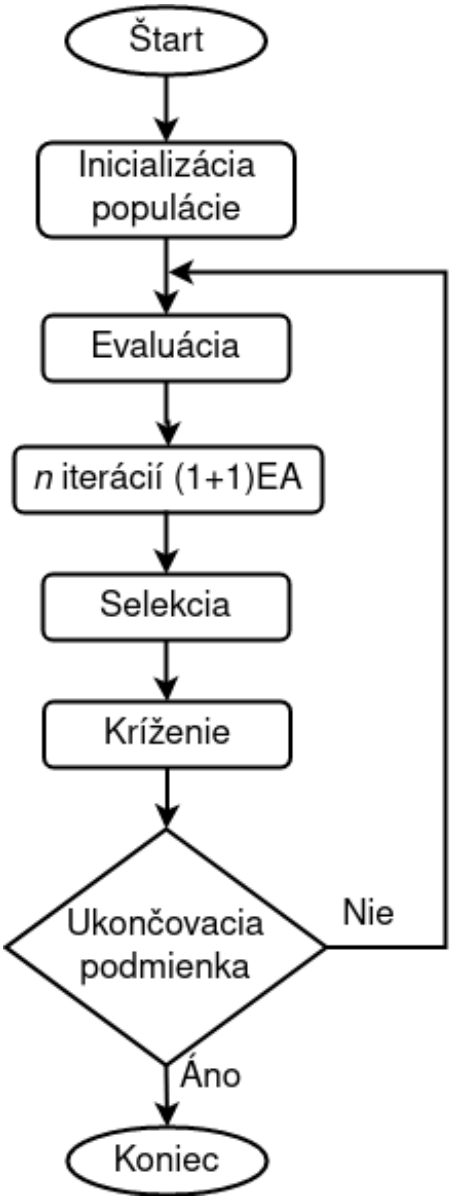
reprezentácií vybraného podproblému, vykoná toľko zmien, aká je stanovená veľkosť kroku. Vykonanie zmeny sa implementuje rovnako, ako v prípade mutácie, pre plán cesty sa vykoná mutácia reverznou sekvenciou a v pláne naberania predmetov sa neguje náhodné binárne číslo. Nakoniec sa skontroluje, či má väčšiu hodnotu fitnes pôvodný alebo upravený jedinec. Lepší ostáva a je výstupom algoritmu alebo sa na nom vykonajú ďalšie iterácie.

(1+1)EA implementujem spolu s GA môžu bežať samostatne alebo sa spolu skombinovať. Použiť (1+1)EA v rámci GA je možné, ako lokálne prehľadávanie pre vylepšenie jedincov alebo, ako náhradu mutácie. Tieto možnosti otestujem a navzájom porovnám.

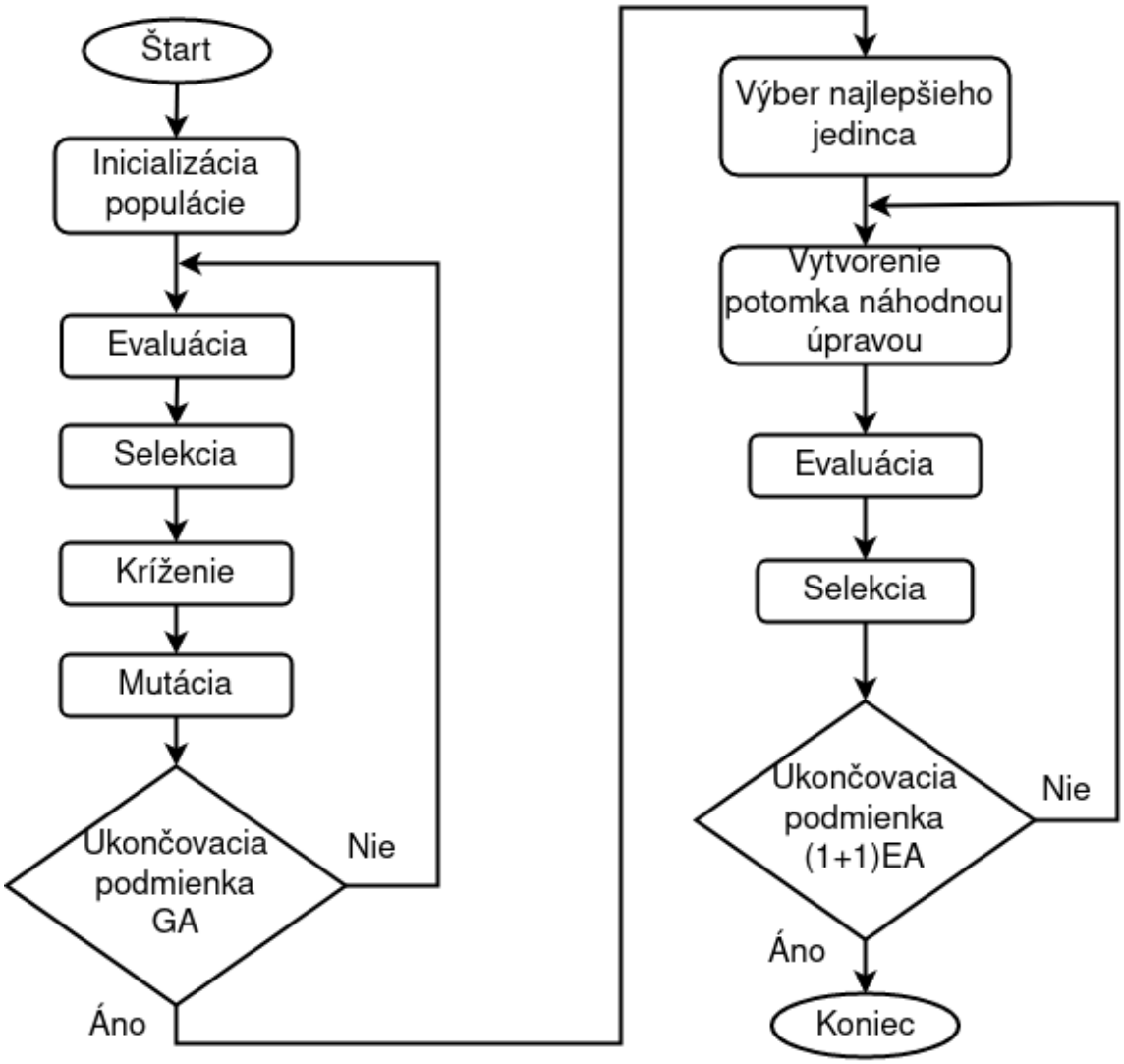
3.6 GA a hybridný GA

Výsledný genetický algoritmus bude využívať navrhnuté kódovanie, fitnes funkciu a operátory. Na začiatku načíta dáta zo vstupného súboru, spracuje ich a uloží do vhodných štruktúr. Ďalšími vstupnými parameterami budú nastavenia GA. Užívateľ nastaví koľko iterácií GA sa vykoná, veľkosť populácie, pravdepodobnosť mutácie. Alebo detailnejšie nastavenia, ako rozsah pre množstvo bodov kríženia. Algoritmus na začiatku vytvára počiatočnú generáciu stanovenej veľkosti s náhodne vygenerovanými riešeniami problému. Potom sa začne proces vyhľadávania optimálneho riešenia, algoritmus bude postupne vykonávať operáciu selekcie, kríženia a mutácie. Tento proces sa vykoná stanovený počet iterácií a algoritmus sa ukončí. Výsledkom je najlepší jedinec, najlepšie nájdené riešenie problému.

Okrem tejto implementácie GA, otestujem jeho kombináciu z navrhnutým (1+1)EA. Takúto spoluprácu algoritmov môžme označiť ako hybridný genetický algoritmus, viď obr.3.7. (1+1)EA použijem pre lokálne prehľadávanie na vylepšenie jedincov medzi iteráciami GA ako náhradu za operátor mutácie. Tiež je možné použitie oboch algoritmov v rámci vyhľadávania tak, že riešenie vyhľadané pomocou GA sa skúsi vylepšiť ešte niekoľkými iteráciami (1+1)EA pre vyhľadanie lokálneho maxima. Takúto kombináciu tiež otestujem, môžme ju označiť ako **GA+(1+1)EA**, viď obr.3.8.



Obr. 3.7: Hybridný GA

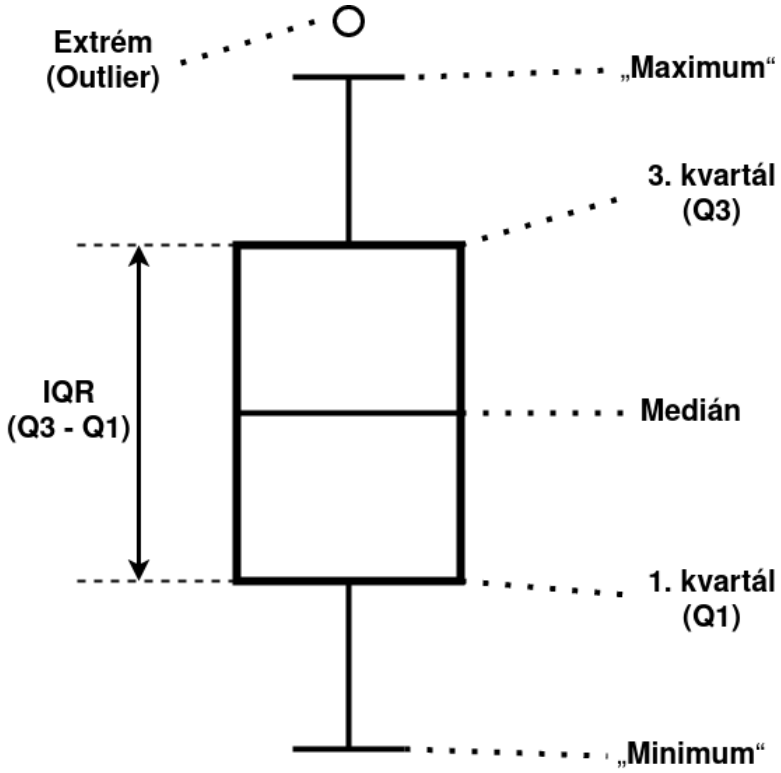


Obr. 3.8: Algoritmus GA + (1+1)EA

3.7 Testovanie algoritmu

Navrhnutý algoritmus je stochastický, nedeterministický, a pri viacerých spusteniach s rovnakým vstupom bude prichádzať k rozdielnym výstupom. Aby bolo možné rôzne nastavenia algoritmu navzájom porovnávať, je potrebné každé nastavenie spustiť niekoľko krát nad rovnakým vstupom. Potom testované nastavenia dokážeme porovnať napríklad na základe vypočítaného mediánu z výstupov.

Pre grafické porovnanie variácií algoritmu budem používať krabicový graf (angl. Box plot). Umožňuje vzájomné porovnávanie skupín dát, zobrazuje medián, kvartily, maximálnu a minimálnu hodnotu. Prvý dolný kvartil sa rovná 25. percentilu dát, bude oddeľovať najmenších 25% hodnôt od zvyšku. Druhý kvartil je medián rovná sa 50. percentilu dát, nachádza sa v strede zoradených dát. Tretí kvartil sa rovná 75. percentilu. Ak sa v dátach nachádzajú nejaké extrémne hodnoty, výrazne oddelené zvyknú sa samostatne vyznačovať ako body[14].



Obr. 3.9: Popis častí krabicového grafu (Box plot)

Výpočty rozsahov pre krabicový graf z obrázku 3.9, (IQR z anglického InterQuartile Range)[8]

„Minimum“ $Q1 - 1.5 * IQR$

1. kvartil 25. percentil

Medián 50. percentil

3. kvartil 75. percentil

„Maximum“ $Q3 + 1.5 * IQR$

Pri testovaní sa stanový počet spustení algoritmu nad jedným vstupným benchmarkovým súborom. Z výstupných dát, čo sú najlepšie nájdené hodnoty fitnes z daných spustení, sa vykresli krabicový graf. Výsledné grafy rôznych implementácií vyhľadávania, či nastavení parametrov sa použijú ku porovnaniu dosiahnutých výsledkov, výberu najlepšieho návrhu.

Náhodné prehľadávanie môže pomôcť ako referencia pri zisťovaní efektívnosti algoritmu. Implementujem teda veľmi jednoduchý spôsob prehľadávania, pri ktorom budem opakovane generovať náhodných jedincov a ukladať doposiaľ najlepšieho, s najväčším fitness. Takéto čisto náhodné prehľadávanie sa potom môže porovnať s navrhnutým evolučným algoritmom na rovnakom celkovom počte vytvorených jedincov.

Výpočtová náročnosť Aby sa mohla porovnávať efektívnosť algoritmov a ich nastavení, je potrebné zabezpečiť, aby ich výpočet mal podobnú časovú zložitosť. Pre GA je jeho komplexnosť závislá od veľkosti populácie s ktorou pracuje, s počtom generácií, ktoré sa budú simulovať, a veľkosti riešenia problému, jedinca. Komplexnosť zadania a reprezentácie jedinca bude rovnaká pre všetky varianty spustené nad jedným vstupom. Zjednodušene budem teda časovú zložitosť pre GA reprezentovať ako súčin veľkosti populácie a počtu generácií, toto číslo reprezentuje počet jedincov, ktorých sa budú v rámci behu evaluovať. Zložitosť (1+1)EA bude predstavená množstvom vykonaných iterácií, čo opäť znamená množstvo jedincov, ktorý budú počas behu vytvorený. Podobne tak bude pre náhodné prehľadávanie generovaním N náhodných jedincov. Navzájom porovnávané algoritmy sa budú spúšťať v nastaveniach, kedy **počet evaluacií jedincov** v rámci behu bol rovnaký alebo aspoň podobný.

Kapitola 4

Implementácia

Navrhnutý algoritmus budem implementovať v programovacom jazyku **Python**. Tento jazyk som si kvôli tomu, že umožňuje jednoduchú a efektívnu implementáciu návrhu. Je to vysokoúrovňový jazyk so širokou škálou použití. Tiež poskytuje knižnice pre jednoduchú vizualizáciu výsledkov algoritmu. V oblasti umelej inteligencie a strojového učenia je tento jazyk momentálne veľmi populárny. Používam verziu jazyku Python 3.8.5.

Využívať budem niekoľko základných knižníc, ktoré Python poskytuje. Implementácia genetických operátorov a evolučného algoritmu je vytvorená podľa uvedeného návrhu, neboli použité žiadne špeciálne knižnice pre evolučné výpočty či genetické algoritmy.
Medzi použité knižnice patrí **NumPy** pre prácu z vektormi a permutáciami, **Random** ku generovanie náhodných javov, **Logging** implementujúci vytváranie logovacích správ, **Matplotlib** pre vizualizáciu výsledkov algoritmu, **Json** k práci z dátami v json konfiguračnom súbore.

4.1 Diagram tried

Pri implementácií som využíval princípy objektovo orientovaného programovania. Uvádzam diagram tried zobrazujúci model implementácie algoritmu 4.1.

Trieda ***TtpParam*** slúži na načítanie zadania problému, jej objekt sa vytvára na začiatku spustenia programu. Metóda *readDataSet()* otvorí TTP súbor, načíta dáta zadania, uloží ich do svojich atribútov. Zavolaním *calculateDistanceMatrix()* po načítaní dát sa vypočítajú vzdialenosti medzi mestami a uložia sa do atribútu *distanceMatrix*. Obsahuje metódy k získaniu rôznych informácií zo zadania ako *getValueOfItem()* pre hodnotu predmetu, *getWeightOfItem()* pre hmotnosť predmetu, *getDistance()* pre vzdialenosť medzi mestami.

Trieda ***Generator*** obsahuje metódy pre náhodné generovanie jedincov. Objekt tejto triedy sa prepája ku jednému objektu triedy *TtpParam*, odkiaľ čerpá dáta o mestách a predmetoch. Metóda *generateXY()* generuje reťazec X, náhodné riešenie TSP a reťazec Y riešenie KP. Metóda *generateGeneration()* slúži k inicializácií celej počiatočnej generácií jedincov.

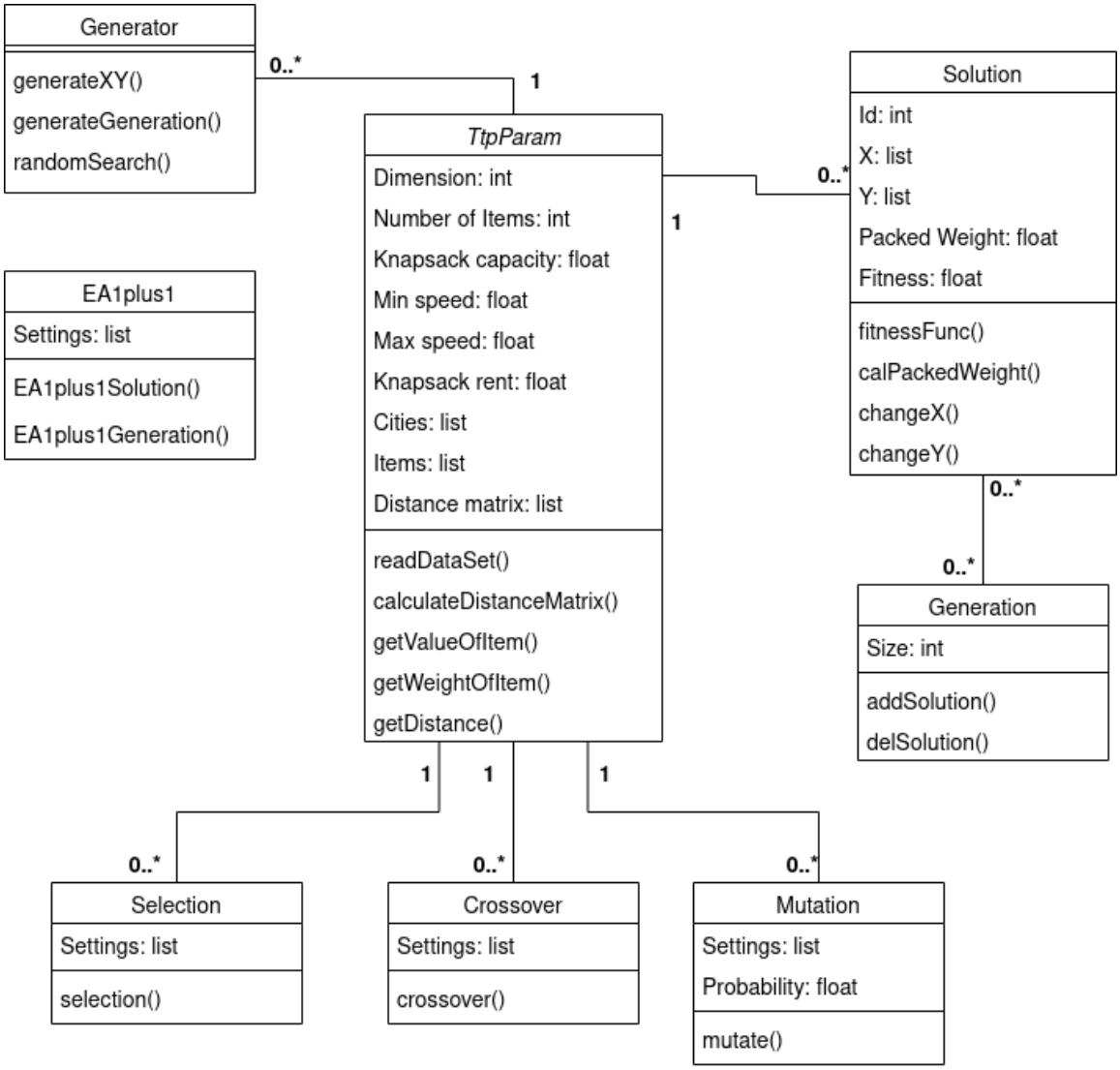
Objekty triedy ***Solution*** budú reprezentovať jedno konkrétne riešenie problému, jedného jedinca. Ten je identifikovaný pomocou atribútu *Id*, atribúty *X* a *Y* sú reťazce repre-

zentujúce riešenia podproblémov, do *PacketWeight* sa ukladá celková hmotnosť predmetov v batohu a atribút *Fitness* obsahuje hodnotu fitnes riešenia. Metóda *fitnessFunc()* prepočíta a aktualizuje fitnes hodnotu, volá sa pri každej zmene riešenia. Podobne pomocou *calPackedWeight()* sa prepočíta hmotnosť v batohu.

Trieda ***Generation*** reprezentuje zoznam jedincov, zoznam objektov triedy *Solution*, používa sa na vytváranie generácií v rámci GA. V atribúte *Size* je zapísané množstvo jedincov v generácií. Má metódy na pridanie *addSolution()* a odstránenie *delSolution()* jedincov.

Genetické operátory sú taktiež reprezentované triedami ***Selection***, ***Crossover***, ***Mutation***. Aj keď sa jedná skôr o funkcie, pre prehľadnosť sú implementované v rámci príslušných tried ako ich metódy. Majú svoj objekt triedy *TtpParam*, kde pristupujú k parametrom zadania. Obsahujú metódy vykonávajúce selekciu, kríženie a mutáciu podľa popísaných návrhov.

Trieda **EA1plus1** obsahuje metódy implementujúce (1+1)EA nad jedným jedincom *EA1plus1Solution()*, či nad celou generáciou *EA1plus1Generation()*.



Obr. 4.1: Implementácia – diagram tried

4.2 Spustenie a nastavenie algoritmu

Algoritmus sa dá jednoducho spustiť v príkazovom riadku príkazom:

```
python3 main.py
```

Všetka konfigurácia a nastavovanie parametrov sa vykonáva v súbore **config.json**. Program po spustený všetky potrebné údaje načíta odtiaľ.
V konfiguračnom súbore je možné zvoliť aký algoritmus sa má spustiť, nastaviť jeho hodnoty parametrov, ktorá implementácia operátora sa má zvoliť, cesta k zdrojovému súboru so zadaním, zapnutie/vypnutie kreslenia grafov pri výstupe.

Implementácia algoritmov je rozdelená v súboroch:

main.py spúšťanie programu, načíta dáta z konfiguračného súboru a spúšťa požadované nastavenie.

algMethods.py definície funkcií implementujúcich rôzne testované algoritmy.

myEA.py obsahuje definície tried pre evolučné algoritmy, implementácia tried pre jedincov, generáciu, genetické operátory atď. Tieto triedy sa potom využívajú k zostaveniu spomínaných testovaných algoritmov.

myIO.py obsahuje triedu *TtpParam* k načítaniu zadania, sú tu tiež definované loggery.

Počas behu sa vytvárajú viaceré logovacie súbory zaznamenávajúce informácie o behu algoritmu, v kóde je možné nastaviť úroveň vážnosti logovania.

Kapitola 5

Testovanie

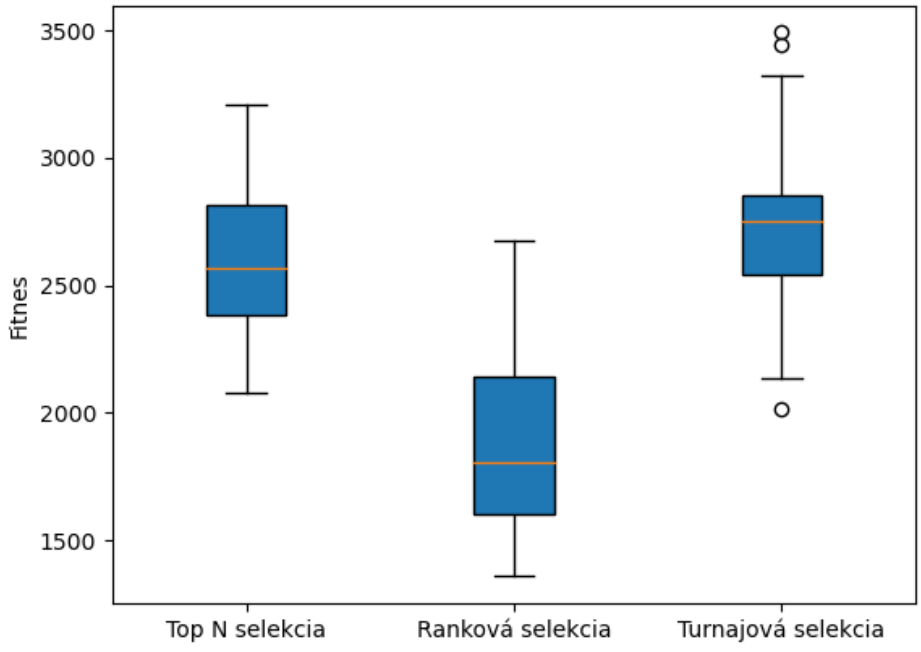
Testovaním porovnám rôzne nastavenia algoritmu medzi sebou, tak budem môcť vybrať najvhodnejšie z nich. Každé nastavenie bude spustené 30 krát, budú vykreslené krabicové grafy z výstupných najlepších hodnôt fitnes pre dané spustenia. U testov sú vždy uvedené základné nastavenia algoritmu a výsledné hodnoty mediánov. Najskôr budem testovať nastavenia GA, potom jeho kombinácie s (1+1)EA.

5.1 Porovnávanie operátorov selekcie

Ako prvé otestujem implementácie operátorov selekcie, pri základnom nastavení genetického algoritmu. Porovnávať budem selekciu výberom najlepších N, rankovú selekciu a turnajovú selekciu (veľkosť turnaja = 5). Výsledný krabicový diagram je na obrázku 5.1.

```
Nastavenie:
    Dátový súbor: eil51_n50_bounded-strongly-corr_01.ttp (mestá:51 , predmety:50)
    Algoritmus: GA
    Populácia: 400
    Počet generácií: 250
    Pravdepodobnosť mutácie: 0.5
    Behy algoritmu: 30

Výsledné mediány:
    Top N selekcia: 2565.44
    Ranková selekcia: 1807.82
    Turnajová selekcia: 2753.605
```



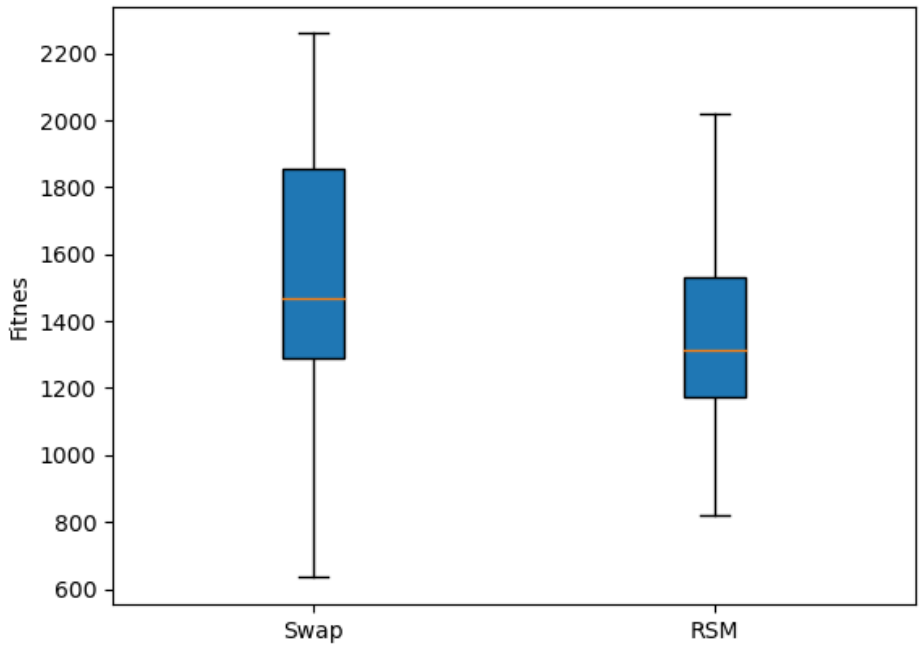
Obr. 5.1: Krabicový graf - testovanie operátorov selekcie

5.2 Testovanie mutácie

U mutácie som porovnával jej implementáciu pre plán cesty, v prvom prípade sa jednalo o prehodenie pozície dvoch indexov (Swap), druhý spôsob invertoval poradie vo vybranej sekvencií (RSM). Na pláne naberania predmetov prebiehala mutácia v oboch prípadoch rovnako. Jednoduchá zámena pozícií dosiahla mierne lepšie výsledky, ako ukazuje obrázok 5.2.

```
Nastavenie:
    Dátový súbor: eil51_n50_bounded-strongly-corr_01.ttp (mestá:51 , predmety:50)
    Algoritmus: GA
    Selekcia: Turnajová
    Populácia: 400
    Počet generácií: 250
    Pravdepodobnosť mutácie: 1
    Behy algoritmu: 30

Výsledné medián y:
    Swap: 1467.85
    RSM: 1315.675
```



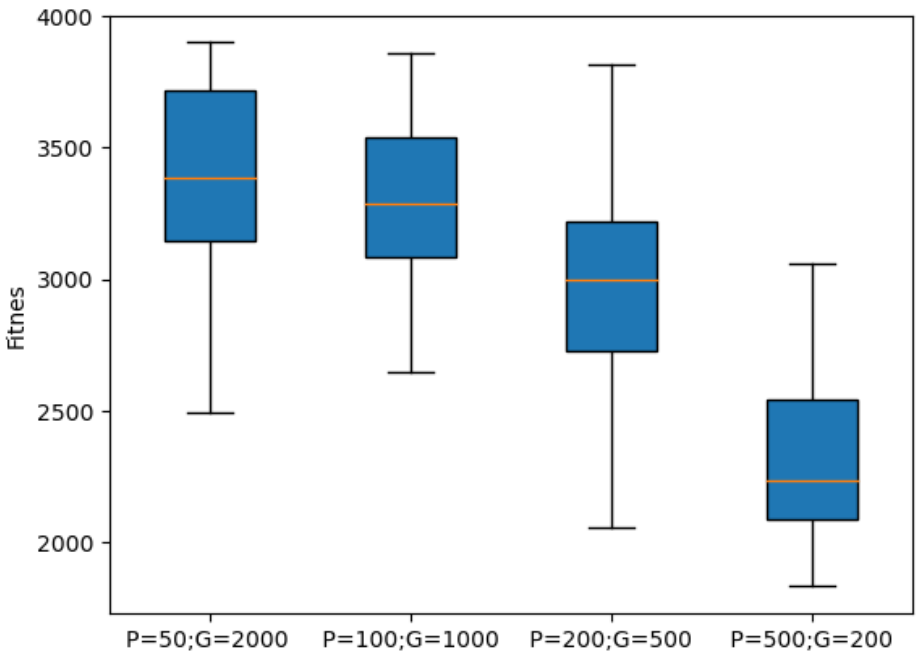
Obr. 5.2: Krabicový graf - testovanie operátorov mutácie

5.3 Pomer veľkosti populácie ku počtu generácií

Otestovaním rôznych pomerov veľkosti populácie a počtu simulovaných generácií, som hľadal najvýhodnejšie nastavenie pre daný algoritmus. Výsledné porovnanie na obrázku 5.3.

```
Dátový súbor: eil51_n50_bounded-strongly-corr_01.ttp (mestá:51 , predmety:50)
Behy algoritmu: 30
Nastavenie:
    Algoritmus: GA
    Selekcia: Turnajová
    Pravdepodobnosť mutácie: 0.5

Výsledné mediány:
    Populácia 50, Generácie 2000: 3384.1
    Populácia 100, Generácie 1000: 3284.22
    Populácia 200, Generácie 500: 2998.51
    Populácia 500, Generácie 200: 2236.86
```



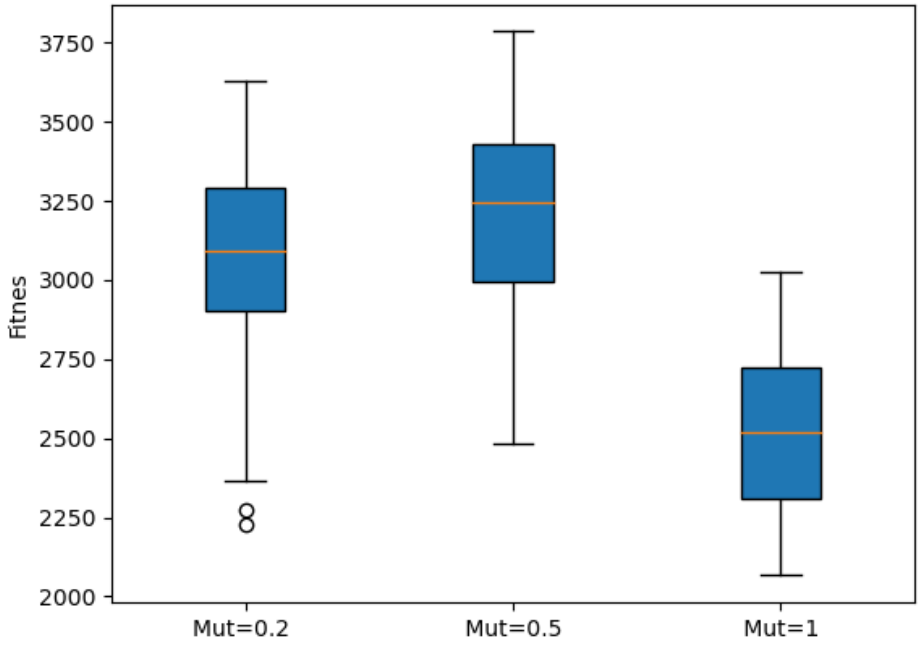
Obr. 5.3: Krabicový graf - pomeru populácie ku počtu generácií

5.4 Pravdepodobnosť mutácie

Pravdepodobnosť mutácie je číslo od 0 po 1 určujúce s akou pravdepodobnosťou sa vykoná mutácia na novovzniknutom jedincovi, porovnám vplyv jej veľkosti na GA, výsledky sú na obrázku 5.4.

```
Dátový súbor: eil51_n50_bounded-strongly-corr_01.ttp (mestá:51 , predmety:50)
Behy algoritmu: 30
Nastavenie:
    Algoritmus: GA
    Selekcia: Turnajová
    Populácia: 80
    Počet generácií: 1250

Výsledné mediány:
    Pravdepodobnosť mutácie 0.2: 3094.045
    Pravdepodobnosť mutácie 0.5: 3244.965
    Pravdepodobnosť mutácie 1: 2518.975
```



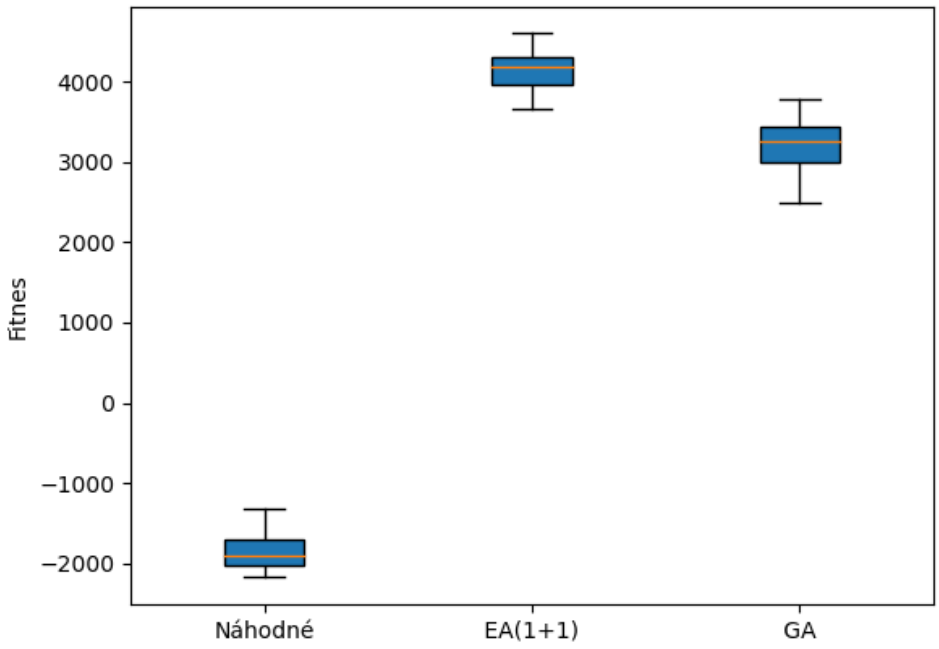
Obr. 5.4: Krabicový graf - vplyv pravdepodobnosti mutácie

5.5 GA vs (1+1)EA

Podľa predošlých testov GA použijem jeho najvýhodnejšie nastavenie a porovnám ho s implementáciou (1+1)EA. Do výsledkov pridám aj čisto náhodné prehľadávanie s rovnakým počtom vytvorených jedincov, pre lepšie znázornenie efektívnosti algoritmov. Na obrázku 5.5 je vidieť, že jednoduchý (1+1)EA dosiahol lepších výsledkov ako GA.

```
Dátový súbor: eil51_n50_bounded-strongly-corr_01.ttp (mestá:51 , predmety:50)
Behy algoritmu: 30
Nastavenie GA:
    Selekcia: Turnajová
    Populácia: 80
    Počet generácií: 1250
    Pravdepodobnosť mutácie: 0.5
Nastavenie (1+1)EA:
    Iterácie: 100000
    Rozsah veľkosti kroku: 1 - 5
Nastavenie náhodného prehľadávania:
    Iterácie: 100000

Výsledné mediány:
    Náhodné: -1908.69;   (1+1)EA: 4191.43;   GA: 3244.97
```



Obr. 5.5: Krabicový graf - porovnanie náhodného prehľadávania, GA, (1+1)EA

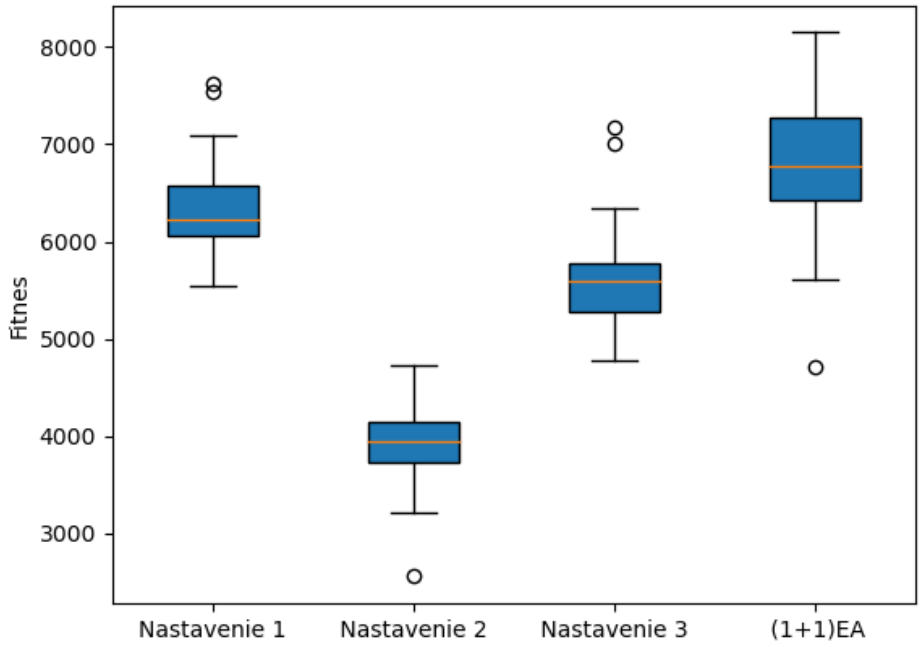
5.6 Hybridný GA

Otestujem ešte kombináciu GA a (1+1)EA do jedného algoritmu. (1+1)EA nahradí operátor mutácie, namiesto neho bude vnášať nové informácie do vyhľadávania. Narozdiel od mutácie, (1+1)EA zmeny zachováva len ak vylepšujú fitnes. Najdôležitejšie parametre, ktoré otestujem bude veľkosť populácie, počet generácií a množstvo iterácií (1+1)EA v rámci jednej generácie pre každého jedinca. Výpočtová náročnosť sa bude dať vyjadriť ako súčin týchto parametrov.

Tiež sa použije iný vstupný dátový súbor, nový vstup má o 100 predmetov viac ako predošlý, pre zvýšenie náročnosti.

```
Dátový súbor: eil51_n150_bounded-strongly-corr_01.ttp (mestá:51 , predmety:150)
Behy algoritmu: 30
Nastavenie 1
    Selekcia: Turnajová
    Populácia: 12
    Počet generácií: 50
    Iterácie (1+1)EA vrámci generácie: 200
    Rozsah veľkosti kroku EA: 1 - 3
Nastavenie 2
    Selekcia: Turnajová
    Populácia: 24
    Počet generácií: 20
    Iterácie (1+1)EA vrámci generácie: 250
    Rozsah veľkosti kroku EA: 1 - 3
Nastavenie 3
    Selekcia: Turnajová
    Populácia: 12
    Počet generácií: 25
    Iterácie (1+1)EA vrámci generácie: 400
    Rozsah veľkosti kroku EA: 1 - 3

Výsledné mediány:
    Nastavenie 1: 6239.39
    Nastavenie 2: 3945.23
    Nastavenie 3: 5592.55
    (1+1)EA: 6779.47
```



Obr. 5.6: Krabicový graf - porovnanie nastavení hybridného GA

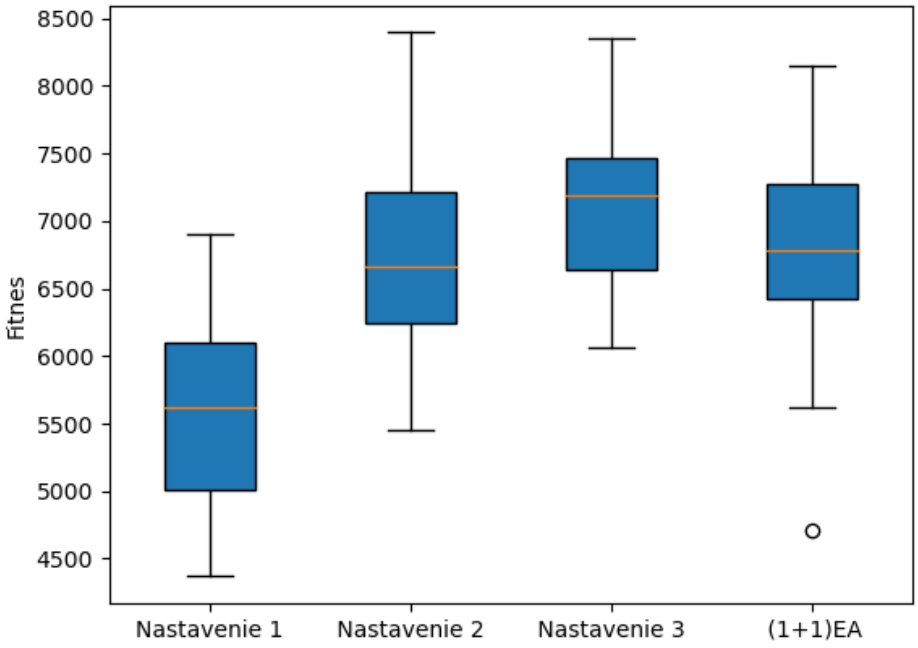
5.7 GA + (1+1)EA

Na záver ešte otestujem použitie obidvoch algoritmov GA a (1+1)EA za sebou. Využijem klasický GA a jeho najlepšie nájdené riešenie sa následne pokúsim vylepšiť použitím (1+1)EA. Porovnám výsledky oproti samostatnému použitiu (1+1)EA s rovnakým počtom evaluacií, ktorý v porovnaní z ostatnými má zatiaľ najlepšie výsledky.

```
Dátový súbor: eil51_n150_bounded-strongly-corr_01.ttp (mestá:51 , predmety:150)
Behy algoritmu: 30
Nastavenie 1
    Selekcia: Turnajová
    Populácia: 80
    Počet generácií: 1250
    Pravdepodobnosť mutácie: 0.5
    Rozsah veľkosti kroku EA: 1 - 5
    Iterácie (1+1)EA nad najlepším: 20000
Nastavenie 2
    Selekcia: Turnajová
    Populácia: 60
    Počet generácií: 1000
    Pravdepodobnosť mutácie: 0.5
    Rozsah veľkosti kroku EA: 1 - 5
    Iterácie (1+1)EA nad najlepším: 60000
Nastavenie 3
```

```
    Selekcia: Turnajová
    Populácia: 32
    Počet generácií: 625
    Pravdepodobnosť mutácie: 0.5
    Rozsah veľkosti kroku EA: 1 - 5
    Iterácie (1+1)EA nad najlepším: 100000

Výsledné mediány:
    Nastavenie 1: 5624.34
    Nastavenie 2: 6665.86
    Nastavenie 3: 7191.12
    (1+1)EA): 6779.47
```



Obr. 5.7: Krabicový graf - porovnanie nastavení GA nasledovaného (1+1)EA

Kapitola 6

Zhodnotenie výsledkov

Testovaním GA som sa snažil nájsť najlepšie nastavenie pre tento algoritmus. Z implementácie selekcie zaostávala ranková selekcia, selekcia najlepších N a turnajová dosahovali podobné výsledky, viď graf 5.1. Ďalej som sa rozhodol využívať turnajovú selekciu. Zmena varianty implementácia mutácie nemala výrazný vplyv na výsledok. Dôležitým faktorom bol pomer veľkosti populácie ku počtu generácií a pravdepodobnosť mutácie. GA dosahoval lepšie výsledky ak bola veľkosť populácie niekoľko násobne menšia od počtu generácií, graf 5.3. Testovanie pravdepodobnosti mutácie ukázalo dobré výsledky na hodnote 0.5. Napriek tomu takto nastavený GA zaostával v porovnaní za jednoduchým (1+1)EA, oba však dosiahli výrazne lepších hodnôt ako implementácia čisto náhodného prehľadávania, graf 5.5 . Následne som sa pokúšal skombinovať tieto dva algoritmy k dosiahnutiu lepších výsledkov, ich kombinácia v hybridnom GA stále zaostáva za samotným (1+1)EA, graf 5.6 . Nakoniec som testoval spustenie GA, po jeho ukončení sa k vylepšeniu jeho výsledného riešenia použil (1+1)EA, toto nastavenie som nazval GA + (1+1)EA. Pri kombinácií kedy väčšia časť výpočtov bola prenechaná na (1+1)EA, toto nastavenie mierne prekonalo samotný (1+1)EA, graf 5.7. Preto som túto kombináciu zvolil ako najlepšie nájdené nastavenie, na obrázku 3.8 je diagram popisujúci vybraný algoritmus GA + (1+1)EA.

Ďalej spustím vybrané najlepšie nastavenie navrhnutého algoritmu na základe predošlých vykonaných testov, zaznamenám jeho výsledky pre viaceré vstupné súbory. Na benchmarkových zadaniach pre TTP s ktorými pracujem bolo otestovaných viacero algoritmov, môžem teda porovnať, ako som sa priblížil k zverejneným najlepšie nájdeným riešeniam.

Hodnoty najlepších nájdených riešení pre daný vstupný súbor som prebral z práce[9], ktorá bola publikovaná roku 2022 v GECCO (Genetic and Evolutionary Computation Conference). V nej sa uvádzajú doposiaľ najlepšie nájdené riešenia a mnohé boli v rámci práce prekonané. Tieto hodnoty použijem ako referenčné pre môj algoritmus.

Vstupný súbor	**GA+EA(1+1) medián**	**Náhodné prehľadávanie medián**	**Najlepšie referenčné[9]**
eil51_n50_bounded-strongly-corr_01	4377.225	-1349.64	4465
eil51_n150_bounded-strongly-corr_01	7000.14	-25265.06	8293.8
eil51_n50_uncorr_01	3011.34	-6043.70	3227.1
eil51_n150_uncorr_01	6278.4	-29263.88	7854.2
pr152_n151_bounded-strongly-corr_01	2034.29	-109909.36	11117.4

Tabuľka 6.1: Porovnávanie výsledkov algoritmu.

Nastavenie algoritmu:

```
Vstupné dáta:
    eil51_n50_bounded-strongly-corr_01 - Mestá: 51, Predmety: 50
    eil51_n150_bounded-strongly-corr_01 - Mestá: 51, Predmety: 150
    eil51_n50_uncorr_01 - Mestá: 51, Predmety: 50
    eil51_n150_uncorr_01 - Mestá: 51, Predmety: 150
    pr152_n151_bounded-strongly-corr_01 - Mestá: 152, Predmety: 151

Počet behov algoritmu k výpočtu mediánu: 30

Algoritmus GA + (1+1)EA
    Selekcia: Turnajová
    Populácia: 40
    Počet generácií: 500
    Pravdepodobnosť mutácie: 0.5
    Rozsah veľkosti kroku EA: 1 - 5
    Iterácie (1+1)EA nad najlepším: 100000

Náhodné prehľadávanie:
    Iterácie: 120000
```

Z výsledkov v tabuľke 6.1 je vidieť, že algoritmus nájde relatívne dobré riešenia pri menších zoznamoch miest a predmetov. Pri väčších sadách má už problémy, s časti to môže byť spôsobné, že jeho nastavovanie bolo vykonané na testovaní menších vstupov a tím, že (1+1)EA je náchylné k zaseknutiu sa na lokálnom maxime. Napriek tomu v porovnaní z výsledkami náhodného prehľadávania sa ukazuje efektivita takéhoto prehľadávania založeného na evolúcií.

Kapitola 7

Záver

Cieľom tejto práce bolo zoznámenie sa z evolučnými algoritmami a ich použitím k riešeniu optimalizačných úloh. Zoznámenia sa s problémom cestujúceho zlodeja (TTP), jeho vlastnosťami a možnými spôsobmi riešenia. Následné navrhnutie vlastného prístupu k riešeniu daného problému s využitím evolučných algoritmov. Navrhnutie viacerých prístupov k implementácií niektorých častí algoritmu. Daný návrh implementovať, otestovať v rôznych nastaveniach, zhodnotiť dosiahnuté výsledky. Na začiatku práce uvádzam TTP, jeho relevantnosť, spôsob akým vznikol. K porozumeniu TTP je potrebné porozumieť jeho podproblémom, problému obchodného cestujúceho a problému batohu, preto predstavujem aj tieto problémy. Uvádzam obvyklú definíciu pre TTP, jeho parametre a potrebné výpočty. Ďalej sa v práci nachádzajú základné teoretické informácie ku evolučným algoritmom, špeciálne ku genetickému algoritmu (GA).

Sekcia návrhu algoritmu obsahuje popis zvoleného prístupu k riešeniu problému, začína sa stanovením formátu vstupných dát, parametrov problému. K TTP existujú benchmarkové súbory obsahujúce zadania tohto typu o rôznych veľkostiach, majú stanovený svoj formát, ktorý predstavujem a algoritmus s ním pracuje. Ďalšou úlohou bolo navrhnúť spôsob reprezentovania konkrétneho riešenia problému ako reťazec dát alebo jedinca/chromozón v terminológií GA. Potom bola navrhnutá funkcia, ktorá ohodnocuje kvalitu riešenia, jedinca, v prípade TTP nám ide o maximalizovanie zisku zlodeja. Zvolenie fitnes funkcie bolo teda jednoduché a jednalo sa o výpočet zisku. Nasledoval návrh implementácie genetických operátorov nad danou reprezentáciou, zakódovaním riešenia problému. Návrh bol zameraný na implementáciu klasického GA, ale tiež na algoritmus (1+1)EA, a následné spojenie týchto algoritmov do jedného.

Implementácia bola vykonaná v jazyku Python. K reprezentácií vstupného zadania, jedincov, generácií, generátora riešení, genetických operátorov som vytvoril triedy obsahujúce potrebné atribúty a metódy. Tie potom využívam na poskladanie navrhnutého algoritmu. Výsledný program je možné spustiť vo viacerých nastaveniach a všetky sa definujú v rámci konfiguračného súboru.

Veľkou časťou práce bolo testovanie algoritmu a jeho nastavení. Na výkonnosť GA mal najväčší vplyv výber správneho pomeru veľkosti populácie a počet vykonaných iterácií, teda simulovaných generácií. Okrem toho sa testovali rôzne implementácie operátorov a pravdepodobnosť mutácie. Otestoval sa výkon (1+1)EA prehľadávania, ten napriek svojej jednoduchosti bol efektívnejší pri riešení TTP než klasický GA. Skúšal som spojenie týchto algoritmov v dvoch variantách, k najlepším výsledkom som sa dostal použitím klasického GA na začiatku s malou časťou výpočtovej kapacity, a použití (1+1)EA na jednom nájdenom riešení s využitím zvyšnej väčšej časti výpočtovej kapacity. Túto kombináciu

som použil v závere na porovnanie výsledkov z verejnými najlepšími riešeniami pre dané vstupy. Výsledný algoritmus podľa očakávaní pri porovnávaní nebol schopný nájsť najlepšiu známu hodnotu v rozumnom čase, hlavne pre zložité vstupné zadania. Dostatočne sa však priblížil, a oproti čisto náhodnému prehľadávaniu ukázal svoju vysokú efektivitu. Z čoho môžem urobiť záver, že sa jedná o jednoduchý spôsob riešenia zložitých problémov s večším prehľadávacím priestorom, v prípadoch ak nieje potrebné vyhľadať vždy najlepšie riešenie.

K možným vylepšeniam algoritmu v budúcnosti existuje množstvo prístupov k implementácií jednotlivých genetických operátorov, ktoré by mohli ovplyvniť jeho výkonnosť. Taktiež existuje veľa spôsobov implementácie lokálnych prehľadávaní pre podproblémy TTP, ich vhodné zakomponovanie do algoritmu napríklad pri generovaní počiatočnej populácie, pre vytvorenie lepšieho štartu GA, by malo mať pozitívni efekt na jeho výkonnosť. Algoritmus je teda možné ďalej upravovať a vylepšovať ho.

Literatúra

[1] Ali, H., Zaid Rafique, M., Shahzad Sarfraz, M., Malik, M. S., Alqahtani, M. A. et al. A novel approach for solving travelling thief problem using enhanced simulated annealing. *PeerJ Computer Science*. 2021, zv. 7. DOI: 10.7717/peerj-cs.377.

[2] Alridha, A., Salman, A. M. a Al Jilawi, A. S. The Applications of NP-hardness optimizations problem. *Journal of Physics: Conference Series*. IOP Publishing. mar 2021, zv. 1818, č. 1, s. 012179. DOI: 10.1088/1742-6596/1818/1/012179. Dostupné z: https://dx.doi.org/10.1088/1742-6596/1818/1/012179.

[3] Bonyadi, M. R., Michalewicz, Z. a Barone, L. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In: *2013 IEEE Congress on Evolutionary Computation*. 2013, s. 1037–1044. DOI: 10.1109/CEC.2013.6557681.

[4] Hynek, J. *Genetické algoritmy a genetické programování*. 1. vyd. Praha: Grada, 2008. Průvodce. ISBN 978-80-247-2695-3.

[5] Katoch, S., Chauhan, S. S. a Kumar, V. A review on Genetic Algorithm: Past, present, and future. *Multimedia Tools and Applications*. 2020, zv. 80, č. 5, s. 8091–8126. DOI: 10.1007/s11042-020-10139-6.

[6] Kenneth, A. D. J. Evolutionary computation: a unified approach. Springer. 2007, č. 3. ISSN 1389-2576.

[7] Kvasnička, V. *Evolučné algoritmy*. 1. vydanie. Bratislava: Slovenská technická univerzita v Bratislave vo Vydavateľstve STU, 2000. Edícia vysokoškolských učebníc. ISBN 80-227-1377-5.

[8] Mcleod, S. *Box plot explained: Interpretation, examples and comparison*. Feb 2023. Dostupné z: https://www.simplypsychology.org/boxplots.html.

[9] Nikfarjam, A., Neumann, A. a Neumann, F. On the Use of Quality Diversity Algorithms for the Traveling Thief Problem. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: Association for Computing Machinery, 2022, s. 260–268. GECCO '22. DOI: 10.1145/3512290.3528752. ISBN 9781450392372. Dostupné z: https://doi.org/10.1145/3512290.3528752.

[10] Otman, A., Abouchabaka, J. a Tajani, C. Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem. *Int. J. Emerg. Sci.* Marec 2012, zv. 2.

[11] Pisinger, D. Linear Time Algorithms for Knapsack Problems with Bounded Weights. *Journal of Algorithms*. 1999, zv. 33, č. 1, s. 1–14. DOI: https://doi.org/10.1006/jagm.1999.1034. ISSN 0196-6774. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0196677499910349.

[12] Polyakovskiy, S., Bonyadi, M. R., Wagner, M., Michalewicz, Z. a Neumann, F. A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2014, s. 477–484. GECCO '14. DOI: 10.1145/2576768.2598249. ISBN 9781450326629. Dostupné z: https://doi.org/10.1145/2576768.2598249.

[13] Prayudani, S., Hizriadi, A., Nababan, E. B. a Suwilo, S. Analysis Effect of Tournament Selection on Genetic Algorithm Performance in Traveling Salesman Problem (TSP). *Journal of Physics: Conference Series*. IOP Publishing. jun 2020, zv. 1566, č. 1, s. 012131. DOI: 10.1088/1742-6596/1566/1/012131. Dostupné z: https://dx.doi.org/10.1088/1742-6596/1566/1/012131.

[14] Williamson, D., Parker, R. a Kendrick, J. The box plot: A simple visual method to interpret data. *Annals of internal medicine*. Júl 1989, zv. 110, s. 916–21. DOI: 10.1059/0003-4819-110-11-916.

[15] Zelinka, I. *Evoluční výpočetní techniky : principy a aplikace*. 1. české vyd. Praha: BEN, 2009. ISBN 978-80-7300-218-3.

[16] Zhang, Y. S. a Hao, Z. F. Runtime Analysis of (1+1) Evolutionary Algorithm for a TSP Instance. In: Panigrahi, B. K., Das, S., Suganthan, P. N. a Dash, S. S., ed. *Swarm, Evolutionary, and Memetic Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 296–304. ISBN 978-3-642-17563-3.

Príloha A

Návod k spusteniu implementácie algoritmu

K práci sú priložené zdrojové súbory s implementáciou algoritmu, nachádzajú sa v priečinku *src*:

```
src/main.py
src/algMethods.py
src/myEA.py
src/myIO.py
```

V priečinku *src/data* sa nachádzajú vstupné zadania TTP používané v práci:

```
src/data/eil51_n50_bounded-strongly-corr_01.ttp
src/data/eil51_n50_uncorr_01.ttp
src/data/eil51_n150_bounded-strongly-corr_01.ttp
src/data/eil51_n150_uncorr_01.ttp
src/data/pr152_n151_bounded-strongly-corr_01.ttp
```

Všetky nastavenia algoritmu sa pred spustením vykonávajú v súbore *src/config.json*. Ďalej uvediem kde sa nastavujú najdôležitejšie parametre v konfiguračnom súbore:

- Nastavenie cesty k vstupnému TTP súboru so zadaním:

  ```
  "dataFilePath": "data/eil51_n50_uncorr_01.ttp"
  ```

- Voľba jednej z uvedených variácií algoritmov k vyhľadávaniu riešenia TTP (GA, hybridný GA, (1+1)EA, náhodné prehľadávanie, GA + (1+1)EA):

  ```
  "_commentMods": "Availible mods: 1 - GA, 2 - hybrid GA, 3 - EA1plus1,
  4 - Random search, 5 - GA + EA",

  "mode" : 5,
  ```

- Voľba spôsobu implementácie selekcie (výber najlepších N, ranková selekcia, turnajová selekcia):

```
"_commentSelections": "Availible selections: 1 - top N, 2 - rank,
3 - tournament",

"selection" : 1,
```

- Nastavenie parametrov týkajúcich sa genetického algoritmu. A to je veľkosť populácie, počet generácií, pravdepodobnosť mutácie, počet bodov kríženia:

```
"geneticAlg":{
      "population" : 20,
      "numOfGenerations" : 40,
      "mutationRate": 0.5,
      "crossPointsSetting": {
          "minX": 1,
          "maxX": 5,
          "minY": 1,
          "maxY": 5
        }
},
```

- Nastavenie parametrov (1+1)EA, čím je počet iterácií a rozsah veľkosti kroku:

```
"EA1plus1": {
    "iterations" : 10,
    "minStep": 1,
    "maxStep": 8
},
```

- Nastavenie počtu iterácií pre náhodné prehľadávanie:

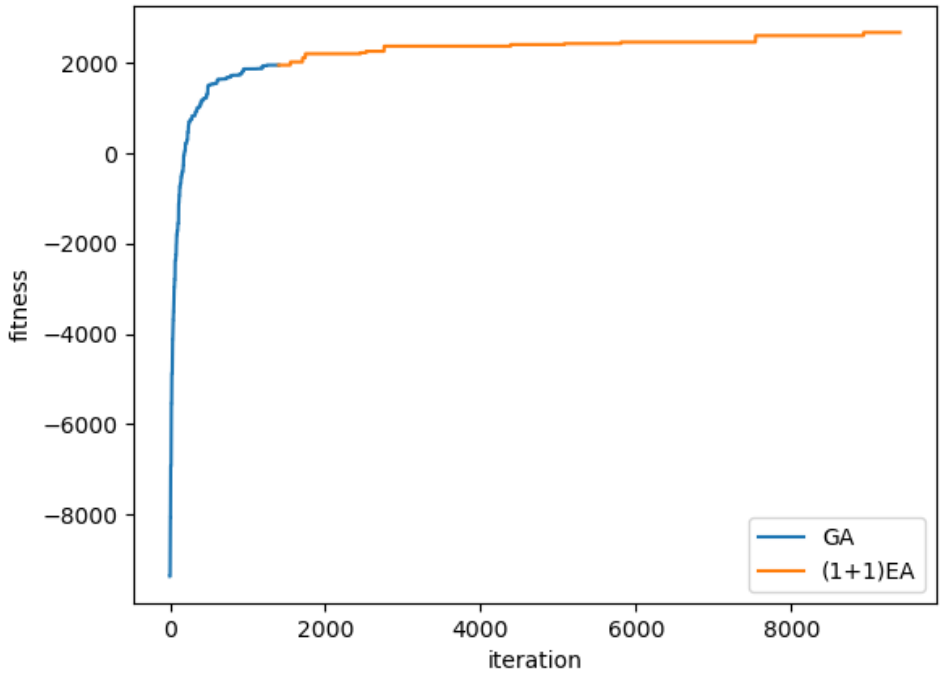
```
"randSearch": {
    "iterations" : 1200
},
```

- Vypnutie/zapnutie zobrazovania grafu s vývojom najlepšej fitnes hodnoty, viď obr.A.1:

```
"showPlot": true
```

- Pre zobrazenie krabicového diagramu k danému nastaveniu je potrebné ho spustiť niekoľko krát za sebou. Takýto mód programu je možné nastaviť zapnutím štatistického módu a nastavením počtu spustení (takéto spustenie môže byť výpočtovo veľmi náročné s dlhým časom výpočtu):

```
"statisticMode": {
        "on" : false,
        "numOfRuns" : 30,
        "showBoxPlot" : true
},
```



Obr. A.1: Vývoj fitnes hodnoty s počtom iterácií pre nastavenie GA + (1+1)EA

Po nastavení konfiguračného súboru sa program jednoducho spustí cez príkazový riadok. V adresári */src* zadáme príkaz:

```
/src$ python3 main.py
```