Ralfs Braunfelds
rb17055

Praktiskās kombinatoriālās optimizācijas
Mazais praktiskais dabs

1. Problēmas apraksts

Šajā praktiskajā darbā izvēlējos ieviest optimizācijas algoritmu ***0-1 Knapsack*** problēmai:

- Tiek dota soma jeb *knapsack* ar kapacitāti *c*;
- Tiek doti *n* priekšmeti, kurus var ievietot somā;
- Katram priekšmetam *i* ir svars w_i un vērtība v_i;
- Katru priekšmetu var izvēlēties ievietot vai neievietot somā (katrs priekšmets vienā eksemplārā);
- Uzdevuma atrisinājums ir priekšmetu kopa, kuras kopējā vērtību summa v_i ir vislielākā un svaru summa w_i nepārsniedz somas kapacitāti *c.*

Šī problēma ir atrisināma *pseido-polinomiālā* laikā, izmantojot dinamisko programmēšanu jeb laikā *O(n * c)*, kā arī ir pierādīts, ka šī problēma ir *NP-pilna*, tātad šī problēma ir labs kandidāts optimizācijas algoritmu pielietošanai.

Šo problēmu izvēlējos optimizēt, izmantojot ģenētisko algoritmu.

2. Programmas apraksts

Optimizācijas algoritms tika rakstīts, izmantojot *Java 17*. Ar citām *Java* versijām netika testēts, bet vajadzētu darboties jebkuru jaunāku versiju un *Java 8*. *Java 17* var iegūt šeit: https://jdk.java.net/java-se-ri/17

Programmu ir paredzēts darbināt ar šādu komandu:

```
java –jar rb17055_md1.jar [parametri]
```

Darbinot programmu caur IntelliJ (nevis ar *jar* failu no komandrindas), krietni uzlabojas tās ātrdarbība, jo programma izpildes laikā veic daudzas konsoles izdrukas.

Programmas *help* izdruka ir pieejama, palaižot programmu bez neviena parametra, vai ar *help* kā pirmo parametru. Parametri programmai ir jāpadod konkrētā secībā un tie ir šādi:

- *input_file* – ceļš uz *knapsack* problēmas ieejas datiem (formātu skatīt zemāk);
- *iterations* – cik reizes tiks ģenerēta jauna populācija (i*nteger*);
- *population_size* – cik liela būs populācija katrā iterācijā (i*nteger*);
- *keep_rate* – varbūtība, ka bērns tiks saglabāts *as-is* un tiks ievietots nākamajā populācijā bez modifikācijām (*float*);
- *crossover_rate* – varbūtība, ka bērns tiks veidots kā *crossover* no diviem vecākiem, izmantojot vienmērīgo *crossover*;
- *mutation_rate* – mutācijas varbūtība katram priekšmetam. Ja varbūtība izpildās, tad:
    - Ja bērns nesatur šo priekšmetu, tad tas tiek pievienots;
    - Ja bērns satur šo priekšmetu, tad tas tiek izņemts.

Piemērs ievadei:

```
java –jar rb17055_md1.jar ./test_cases/uncorrelated_10000_1.txt 1000 1000 0.3 0.3 0.001
```

Ieejas faila formāts:

```
1  c
2  v1 w1
3  v2 w2
4  v3 w3
5  ...
6  vn wn
```

3. Algoritma apraksts

Ģenētiskais algoritms secīgi darbojas šādi:

1. Katra elementa *fitness* tiek aprēķināts kā saturošo elementu vērtību v_i summa. Ja elementu svaru summa w_i pārsniedz somas kapacitāti *c*, tad elementa *fitness* ir 0.
2. Sākotnējās populācijas ģenerēšana. Katrs elements tiek veidots, nejauši izvēloties un ievietojot priekšmetus, līdz kārtējo priekšmetu vairs nevar ievietot somā.
3. Iterācijas skaita reizes tiek ģenerēta nākamā populācija. Katrs nākamās populācijas bērns tiek ģenerēts šādi:
    a. Ar *roulette wheel selection* tiek izvēlēts nejaušs vecāks no esošās populācijas. Tā kā tiek izmantots *roulette wheel selection*, tad vecāki ar lielāku *fitness* tiek izvēlēti biežāk;
    b. Ja izpildās *keep_rate* varbūtība, tad šis vecāks tiek ievietots nākamajā populācijā bez modifikācijām. Pretēji, tiek izpildīti nākamie soļi;
    c. Ja izpildās *crossover_rate* varbūtība, tad ar *roulette wheel selection* tiek izvēlēts nejaušs otrais vecāks. Tiek izveidots bērns ar vienmērīgo *crossover*. Katra priekšmeta iekļaušana tiek noteikta nejauši no viena vai otra vecāka.
    d. Tiek veikta bērna mutācija. Priekš katra priekšmeta, ja izpildās *mutation_rate* varbūtība, tad:
        i. Ja bērns šo priekšmetu jau satur, tas tiek izņemts;
        ii. Ja bērns šo priekšmetu nesatur, tas tiek ievietots.
4. Pēc katras kārtējās populācijas izveidošanas, tiek izdrukāts pašreizējais labākais atrastais risinājums;
5. Pēc visām iterācijas reizēm, tiek izdrukāts pašreizējais labākais *fitness* un atbilstošais risinājums.

4. Testu rezultāti

Pieejamo testa failu nosaukumi ir šādā formātā:

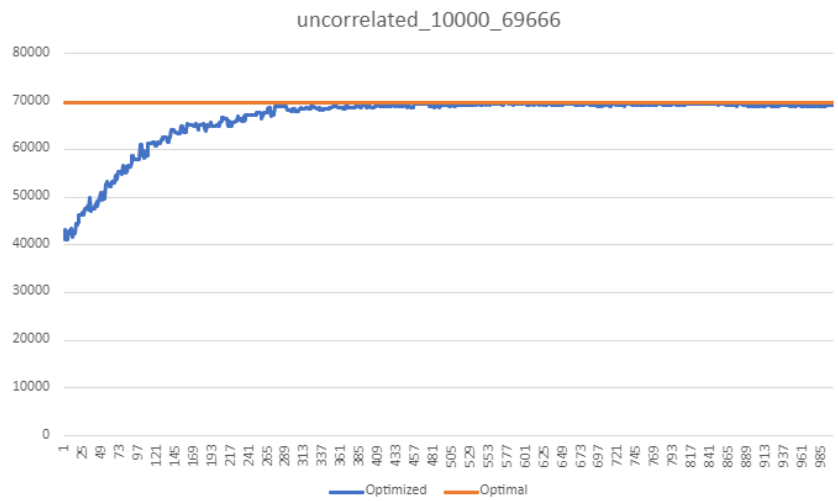
`<type>_<item_count>_<optimal_value>.txt`

- Type:
    - *correlated* gadījumā, kad starp priekšmeta vērtību un svaru ir pozitīvas korelācijas;
    - *Uncorrelated*, ja starp priekšmetu vērtību un svaru korelāciajs nav.
- Item count – priekšmetu skaits;
- Optimal value – problēmas īstais atrisinājums.

Visi redzamie testa faili tika iegūti no http://hjemmesider.diku.dk/~pisinger/codes.html un modificēti atbilstoši programmas vajadzībām.

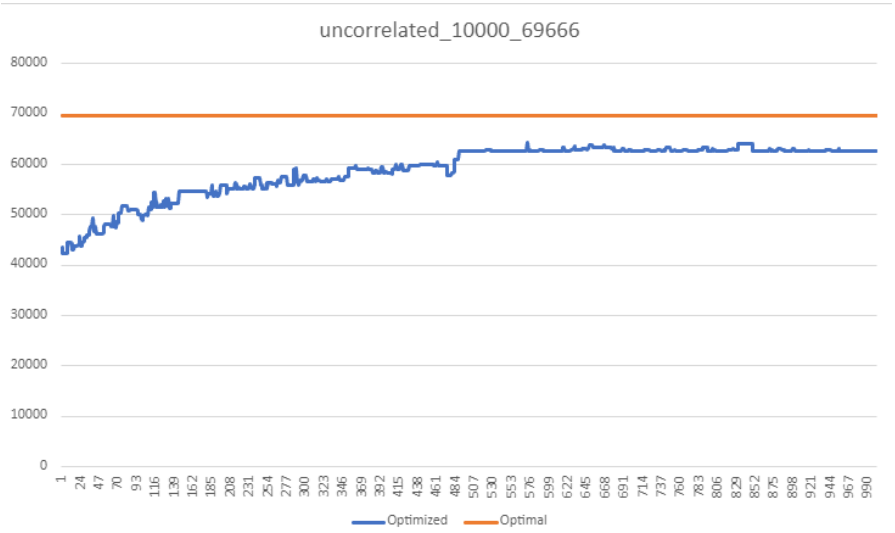
uncorrelated_10000_69666:

- *iterations:* 1000
- *population_size:* 1000
- *keep_rate*: 0.3
- *crossover_rate*: 0.3
- *mutation_rate*: 0
- Izpildes laiks: 2:28 min



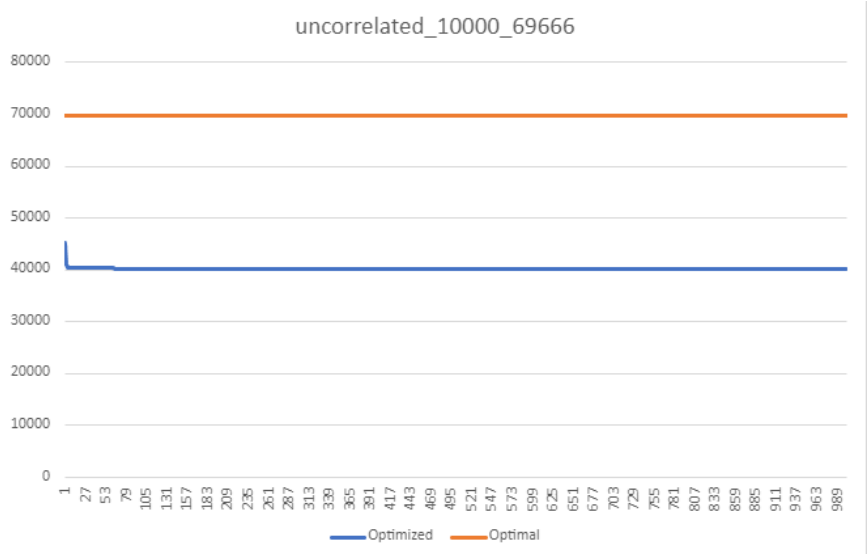
uncorrelated_10000_69666:

- *iterations:* 1000
- *population_size:* 1000
- *keep_rate*: 0.3
- *crossover_rate*: 0.3
- *mutation_rate*: 0.001
- Izpildes laiks: 2:26 min



uncorrelated_10000_69666:

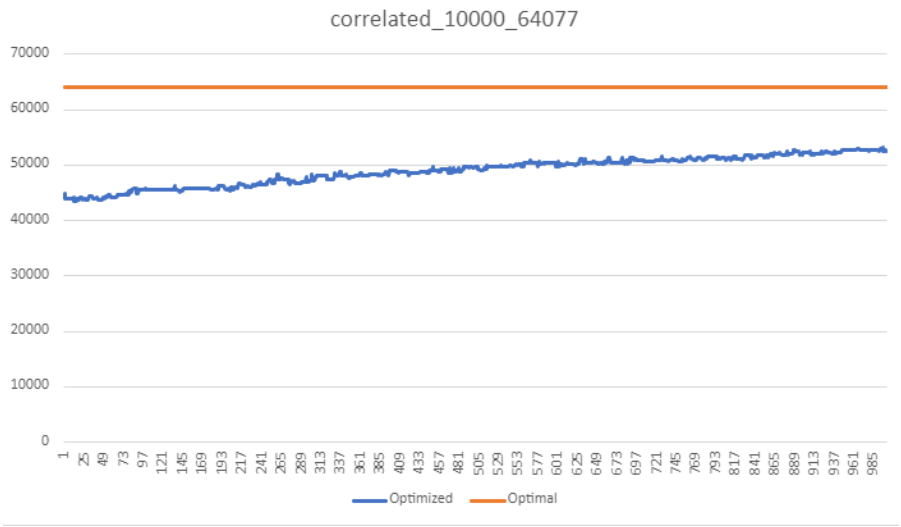
- *iterations:* 1000
- *population_size:* 1000
- *keep_rate*: 0.3
- *crossover_rate*: 0.3
- *mutation_rate*: 0.01
- Izpildes laiks: 2:26 min



Augstāk redzamajos testos varam pamanīt, cik svarīgi ir izvēlēties zemu *mutation_rate.* Vairākos Interneta resursos pamanīju, ka *mutation_rate* vērtībai parasti ir jābūt >= 0.01, bet *knapsack* problēmas gadījumā tam ir jābūt daudz zemākam. Tikai pie 0.001 optimizators spēja tuvoties optimālai vērtībai, savukārt, pilnībā izslēdzot mutāciju, optimizators gandrīz ieguva optimālo vērtību.

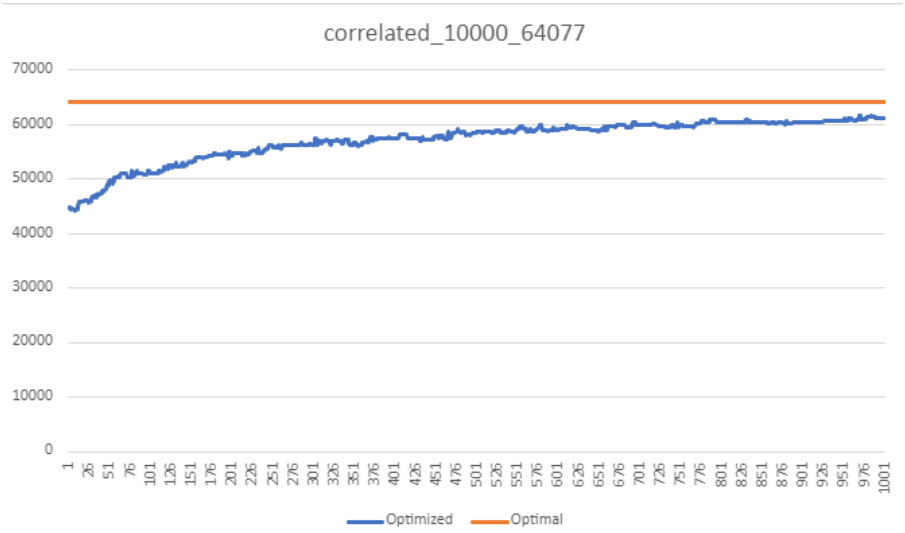
correlated_10000_64077:

- *iterations:* 1000
- *population_size:* 1000
- *keep_rate*: 0.3
- *crossover_rate*: 0.3
- *mutation_rate*: 0
- Izpildes laiks: 2:29 min



correlated_10000_64077:

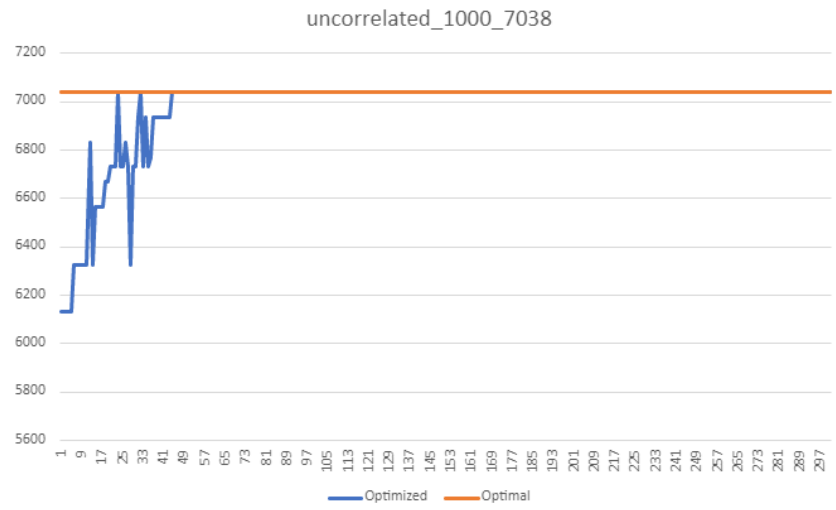
- *iterations:* 1000
- *population_size:* 1000
- *keep_rate*: 0.3
- *crossover_rate*: 0.3
- *mutation_rate*: 0.001
- Izpildes laiks: 2:38 min



Augstāk redzamajos testos varam pamanīt, ka korelētiem datiem, savukārt, mutācija palīdz optimizatoram ātrāk konverģēt uz optimālo vērtību.

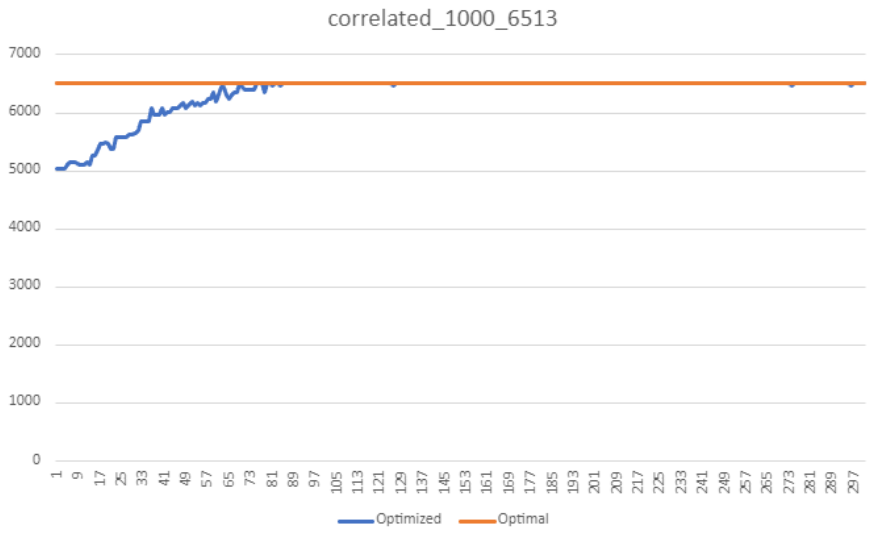
uncorrelated_1000_7038:

- *iterations:* 300
- *population_size:* 500
- *keep_rate*: 0.3
- *crossover_rate*: 0.1
- *mutation_rate*: 0
- Izpildes laiks: 2.1 sec



correlated_1000_6513:

- *iterations:* 300
- *population_size:* 500
- *keep_rate*: 0.3
- *crossover_rate*: 0.1
- *mutation_rate*: 0.001
- Izpildes laiks: 2.26 sec



Augstāk redzamajos testos varam pamanīt, ka, pie maziem priekšmetu skaitiem, nav grūti atrast parametrus, lai optimizators nonāktu līdz optimālai vērtībai.