Zpráva k 5. domácímu úkolu z předmětu MI-PAA

Jan Sokol
sokolja2@fit.cvut.cz

30. prosince 2018

Abstrakt

Je dána booleovská formule F proměnnných $X = (x_1, x_2, ..., x_n)$ v konjunktivní normální formě (tj. součin součtů). Dále jsou dány celočíselné kladné váhy $W = (w_1, w_2, ..., w_n)$. Najděte ohodnocení $Y = (y_1, y_2, ..., y_n)$ proměnných $x_1, x_2, ..., x_n$ tak, aby $F(Y) = 1$ a součet vah proměnných, které jsou ohodnoceny jedničkou, byl maximální.

Je přípustné se omezit na formule, v nichž má každá klauzule právě 3 literály (problém $3SAT$). Takto omezený problém je stejně těžký, ale možná se lépe programuje a lépe se posuzuje obtížnost instance (viz Selmanova prezentace v odkazech).
Obdobný problém, který má optimalizační kritérium ve tvaru "aby počet splněných klausulí byl maximální"a kde váhy se týkají klausulí, se také nazývá problém vážené splnitelnosti booleovské formule. Tento problém je lehčí a lépe aproximovatelný. Oba problémy se často zaměňují i v seriózní literatuře.

1 Úvod do problému

SAT je problém splnitelnosti. V logice vyjadřuje problém odpověď na otázku, zdali existuje zadanému výrazu (formuli) zapsaného ve výrokové Booleovské logice pomocí operací AND, OR a NOT přiřazení, při kterém bude výraz ohodnocen jako pravdivý.

$$x_1 \wedge x_2 \wedge x_3$$

Problém nazývaný "3SAT", "3CNFSAT", nebo "3-satisfiability"je definování speciálního případu, kdy určení splnitelnosti formule v konjunktivní normální formě, kde každá klauzule je omezena na nejvýše tři literály.

$$(x_1 \vee x_2 \vee x_3) \wedge (x_5 \vee x_6 \vee x7)$$

MAX-SAT problem je generalizace SAT problému. Je to problém určení maximálního počtu klauzulí, dané formule, které můžou být splnitelné pro nějaké přiřazení hodnot.

Vážený (Weighted) MAX-SAT se ptá, jaká může být maximální váha splnitelná pro všechny přiřazení, s tím že máme dány vážené hodnoty klauzulí.

2 Vstupní formát dat - DIMACS

Všechny testovací úlohy byly získány modifikací1 příslušných úloh z DIMACS formátu.

Formát DIMACS slouží pro standardizovanou reprezentaci logické formule pro řešení SAT problému. Pomocí DIMACS jsou definovány dva různé formáty uložení. První slouží pro reprezentaci formule v konjunktivní normální formě, a je označován jako DIMACS CNF. Druhý formát slouží pro reprezentaci obecné logické formule a je označován jako SAT - my budeme vycházet z toho prvního.

Formát je oproti originálnímu DIMACS CNF formátu poněkud upraven a vypadá takto:

```
c cnf with weights
p cnf 6 4
w 2 4 6 8 4 2
1 2 -3 0
-2 4 0
1 4 5 6 0
```

Díky tomu jsou do CNF přidány též váhy jednotlivých proměnných. Tyto hodnoty jsou obsaženy na řádku (začínajícím s w) pod definicí problému (začínajícím s p).

Z prezentace Stochastic Search And Phase Transitions: AI Meets Physics víme, že nejtěžší 3SAT problémy mají poměr 4.3(poměr klauzulí ku počtu proměnných). Na tyto instance se budeme nejvíce zaměřovat.

3 Práce s omezujícími podmínkami

Největším rozdílem od problému batohu zde jsou přísné omezující podmínky, které prostor dělají přetržitým a hůře dostupným.

Proto jsem musel omezující podmínky relaxovat. Do dalších generací procházejí i ti jedinci, kteří podmínky porušují. Až v poslední generaci jsou vybráni ti nejlepší jedinci, kteří podmínky splňují.

4 Cenová (fitness) funkce

Když porovnáváme 2 stavy ze stavového prostoru úlohy, potřebujeme funkci, která umí jednotlivé stavy kvalitativně ocenit. Na rozdíl od problému batohu si však u váženého 3 SATu nevystačíme s pouhým porovnáním dosažené váhy. Musíme ale v zřetel brát i podmínku splnitelnosti formule. Navržena byla proto následující cenová funkce:

$$h(X) = ln(k \cdot e^{\frac{C_X}{C_{total}}} - (1-k) \cdot e^{\frac{W_X}{W_{total}}})$$

kde C_X je počet splněných klauzulí stavu X, C_{total} počet klauzulí formule F, W_X váha stavu X a W_{total} suma vah všech proměnných formule F.

k je zde koeficient chtěné vlastnosti. Pro hodnotu 0 preferujeme pouze co nejvyšší součet vah, pro hodnotu 1 bychom preferovali pouze co nejvíce splněných klauzulí.

5 Výběr jazyka

Pro svou implementaci problému batohu jsem si vybral jazyk Python. Ačkoli to je jazyk interpretovaný a nečekal jsem závratné rychlosti výpočtů, mojím výběrem byl pro to, že jsem jazyk znal a pro jakýkolik koncept je pro mne nejrychlejší.

6 Testovací Hardware

Všechny testy byly prováděny na cloudové linuxové instanci v AWS, běžící na Red Hat Enterprise Linux 7. Velikost instance byla: 2 Core CPU / 8 GB RAM, v názvosloví AWS **m4.large**.

7 Měření výpočetního času

Výpočet běhu funkce je řešen tak, že je spočten strojový čas před během funkce, a také po něm. Tyto časy jsou od sebe odečteny a je vrácen čas v ms.

```
def timing(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        measured_time.append(
          {'type': f.__name__,
           'time': (time2-time1)*1000.0})
        return ret
    return wrap
```

7.1 Popis algoritmu

Zde uvádím úryvek kódu řídící evoluci. Celý funkční kód je k dispozici v repozitáři.

Nejdříve vytvoříme počáteční populaci. Potom v n generacích běžíme nasledovné:

- vytřídíme nejlepší jedince dle fitness funkce necháme je soupeřit v turnaji,
- dle nastaveného elitismu převezmeme x jedinců z minulého kola do tohoto,
- vyplníme novou generaci novými jedinci, kteří jsou:
  - kříženci dvou náhodných minulých jedinců,
  - nebo jeden náhodný jedinec.
- mutujeme náhodné bity těchto nových jedinců
- ověříme, zda jedinec je validní (náklad batohu je menší, než je jeho kapacita).

```
# Create initial population
population = create_population(self.population_size,
                               size)

# Run n generations
for generation in range(0, self.generations):
  sorted_population = sort_population(population)

  # Selection
  new_population = self.tournament(population,
                       self.tournament_count,
                       self.tournament_pool_size )

  # Elitsm
  del sorted_population[self.elitism_count:]

  new_population.extend(sorted_population)
  sorted_population = sort_population(new_population)

  # Fill population with new children
  while len(new_population) != self.population_size:
      child = []
      # Crossover
      if odds_are(self.xover_probability):
          in1 = self.random_individual(new_population)
          in2 = self.random_individual(new_population)

          child = self.crossover_single(in1, in2)
      else:
          # Just pick random individual
          child = deepcopy(random_individual(population)

      # Mutation
      child = self.mutator_random_inverse(child,
                           self.mutation_probability)

      # Check if mutated/crossed individual is valid
      if self.constraint_fn(child):
          new_population.append(child)
```

7.2 Experimenty s nastavením parametrů

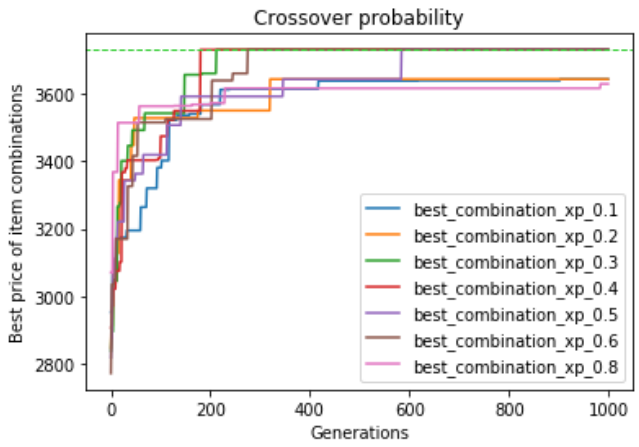
Různě jsem nastavoval parametry genetického algoritmu - pravděpodobnost křížení, pravděpodobnost mutace, elitismus, velikost turnaje, počet turnajů a počet generací. Z grafů jsem poté usoudil, které hodnoty jednotlivých parametrů jsou nejlepší.

Křížení

7.3 Pravděpodobnost křížení

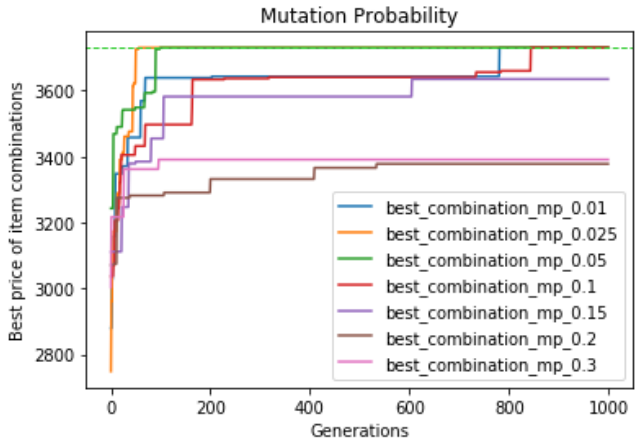
Doporučená pravděpodobnost křížení je 80-95% (Beasley, 1993). Pomocí selského rozumu také lze usoudit, že malá pravděpodobnost křížení snižuje rychlost konvergence a v případě velmi malé hodnoty vůbec zastaví genetický algoritmus. Avšak pravděpodobnost 100% taky není dobrá, jelikož //vždy// ztrácíme rodiče bez jistoty objevu lepšího potomka (algoritmus by měl zachovat "náhodnost"a tím i divergenci populace). Proto byly testovány následující hodnoty: 70%, 75%, 80%, 85%, 90%, 95%, 97%.

Osobně hodnotím 0.3 až 0.4 jako nejlepší hodnotu pravděpodobnosti křížení. Při pravděpodobnosti 0.7 a více ani nebylo možné dosáhnout optimální hodnoty.



7.4 Pravděpodobnost mutace

Na grafu velmi dobře vidíme, že tato hodnota by měla být nízká. Osobně hodnotím 0.025 až 0.1 jako nejlepší hodnotu pravděpodobnosti mutace. Při pravděpodobnosti 0.7 a více ani nebylo možné dosáhnout optimální hodnoty.

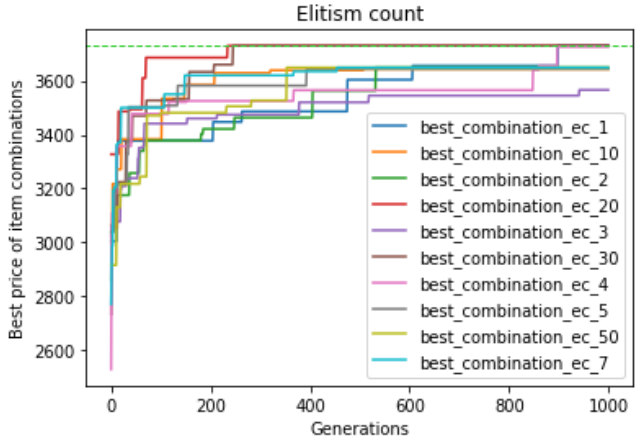


7.5 Elitismus

Vysoký elitismus zvýhodňuje kvalitnější jedince a znevýhodňuje slabší jedince, kteří ale v sobě mohou nést lepší informaci.

Pokud toto nastavíme na vysokou hodnotu, velmi často uvízneme v lokálním maximu/minimu.

Na grafu vidíme, že elitismus se zdá být nejlepší při hodnotě 5.



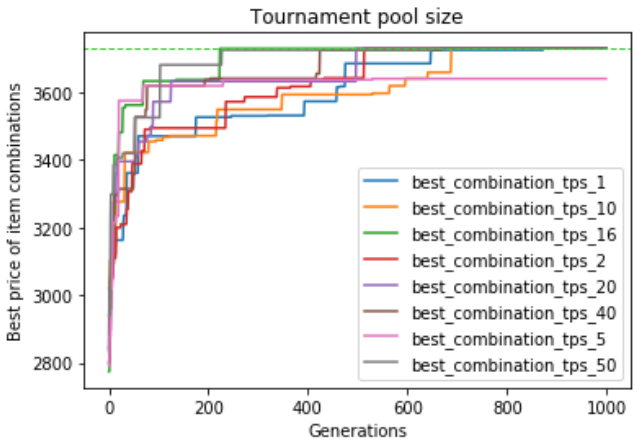
7.6 Velikost turnaje

Turnajem se vybírají jedinci ke křížení. Do každého turnaje vstupuje několik náhodných jedinců z celé populace.

Tento parametr se chová obdobně, jako elitimus, při velkých turnajích jsou znevýhodněni slabší jedinci.

Pokud toto nastavíme na vysokou hodnotu, velmi často uvízneme v lokálním maximu/minimu.

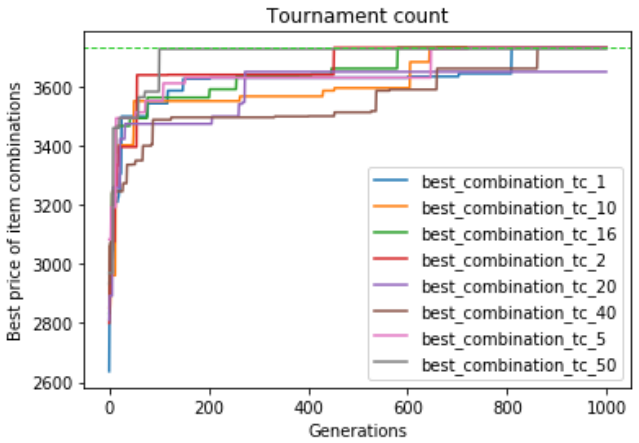
Na grafu vidíme, že velikost turnaje se zdá být nejlepší při hodnotě 2.



7.7 Počet turnajů

Turnajem se vybírají jedinci ke křížení. Z každého turnaje je vybírán vždy jeden, tudíž počet turnajů nepřímo určuje, kolik jedinců bude vybráno ke křížení do nové populace.

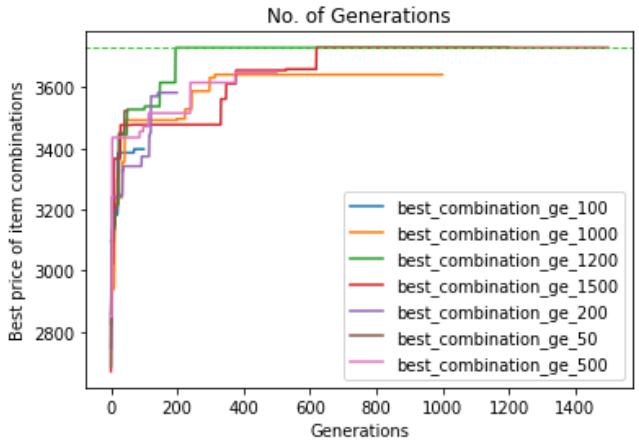
Na grafu vidíme, že počet turnajů se zdá být nejlepší při hodnotě 20 až 50.



7.8 Počet generací

Počtem generací jsem zjišťoval, kolik nejméně generací je potřeba, dokud nedojde ke konvergenci, pro instance s 30 předměty.

Jako moment pro ukončení evoluce jsem vybral 1000 generací, ikdyž jak vidíme níže na grafu, tato hodnota má poměrně velkou rezervu.



8 Shrnutí a závěr

Pokročilá iterativní metoda je nesmírně efektivní při velkých počtech předmětů v instanci.

Přesnost se pohybuje kolem 90-100% (tj, relativní chyba většinou menší, než 10%) a čas strávený výpočtem je několikařádově menší, než při použití jiných metod.

Implementace byla úspěšná a napsána pro jednoduchou výměnu batůžku na SAT.
Pro vytváření grafů bylo využito Python notebooku, který je přiložen v adresáři **report/**. Grafy jsou vykresleny pomocí knihovny **mathplotlib**.

9 Reference