NONOGRAMI

Maša Juras

Maj 2022

1 Uvod

Za projektno nalogo sem si izbrala igro oz. logično uganko Nonogram, saj sem jo že od nekdaj zelo rada reševala. Najprej sem se še bolje spoznala s potekom igre, z možnimi algoritmi za reševanje uganke ter nato začela z implementacijo algoritma, ki bi za podani nonogram uspel najti rešitev. V programskem jeziku Python sem napisala kodo, ki z uporabo genetskega algoritma poskuša čimbolje rešiti podano logično uganko Nonogram.

2 Opis dela

Nonogrami so slikovne logične uganke. Poznamo črno-bele ter barvne nonograme, pri čemer so prvi načeloma bolj enostavni za reševanje in v tej projektni nalogi se ukvarjam z reševanjem le-teh. Imamo torej nepobarvano mrežo ter različne številke na dveh straneh okoli nje. Cilj te logične sestavljanke je, da uporabnik celice mreže pobarva (oz. pusti prazne) tako, kot mu to narekujejo številke, ki so zapisane na levi in zgornji strani mreže. Nonogram je lahko poljubne velikosti, praviloma velja, da večji kot je, težje ga je rešiti. Predvsem je težavnost odvisna od številk, ki določajo barvanje uganke. Ko uporabnik pobarva polja tako, kot mu to narekujejo številke, se mu razkrije skrita slika. To pomeni, da so kvadratki pobarvani tako, da se razkrije neka znana oblika. V kolikor igralec naredi kakšno napako, torej pobarva kakšen napačen kvadratek oz. kakšnega pozabi pobarvati, na koncu ne bo dobil lepe slike in bo na ta način videl, da uganke ni pravilno rešil.

Številke pri tej uganki predstavljajo neko obliko diskretne tomografije, ki meri koliko neprekinjenih vrstic izpolnjenih kvadratkov je v katerikoli vrstici ali stolpcu. Recimo podatki ”5 7 2” bi pomenili, da obstajajo nizi petih, sedmih in nazadnje še dveh zaporedno zapolnjenih kvadratkov, v točno tem vrstnem redu. Ob tem je potrebno poudariti oz. upoštevati še to, da je med njimi (temi bloki števil) vedno vsaj en prazen, torej nepobarvan kvadratek.

3 Algoritmi

Pregledala sem več možnih algoritmov za reševanje nonogramov ter se z vsakim malce seznanila, da sem se lažje odločila katerega bi bilo najbolje implementirati.

3.1 Depth first search - DFS

Verjetno vsem najbolj znan algoritem je "depth first search" algoritem. Pri tem algoritmu gre za generiranje vseh možnih rešitev za vsako posamezno vrstico, kot je prikazano tudi na spodnji sliki.

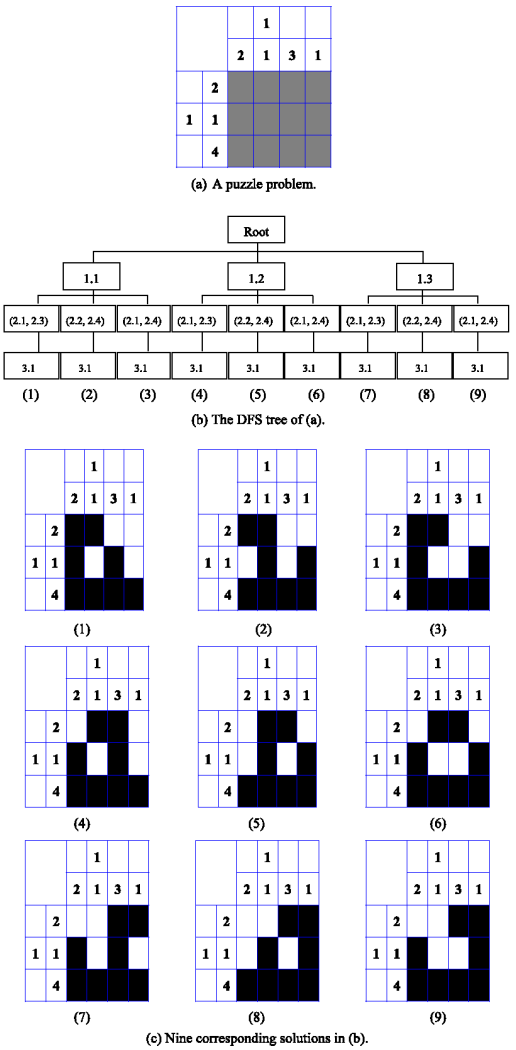

Figure 1: Primer DFS algoritma za reševanje nonograma

3.2 Soft computing algorithm

Delovanje tega algoritma temelji na generiranju permutacij možnih stanj. Po generiranju pa ustvarjena stanja primerja z omejitvami, ki so definirane za vsako posamezno vrstico oziroma stolpec, da preveri ali analizirano stanje ustreza vsem definiranim pogojem.

3.3 Genetic algorithm

Genetski algoritem uporablja veliko biološko pridobljenih tehnik, kot so recimo:

- dedovanje,
- naravna selekcija,
- rekombinacija,
- mutacija.

Vse zgoraj našteto je del procesa evolucije v naravi.

Algoritem deluje s tremi operatorji, in sicer *selekcija*, *crossover* ter *mutacija*. Začne s populacijo oz. generiranjem populacije. Nato se izračuna število omejitev (številk levo in zgoraj), ki zadoščajo igralni površini nonograma. Preveri se, ali je to število omejitev ustrezno veliko, ali pa je morda bilo narejenih preveč iteracij in ob tem rešitev ni bila najdena. Vseskozi torej za generiranje izvaja *selekcijo*, *crossover* in *mutacijo*. V vsakem koraku, ko se izračuna *fitness*, se tudi izvede nova iteracija. V kolikor kakšni omejitvi ni zadoščeno, se postopek ciklično ponavlja. Cilj je zato torej najti velikost števila omejitve, ki je dovolj dobra ter v končni fazi najti rešitev slikovne uganke.

4 Načrt dela

Najprej sem si na računalnik namestila vsa potrebna orodja in programe za izdelavo projektne naloge, in sicer sem uporabila kar Visual Studio Code ter programski jezik Python. Nato sem preučila zgoraj opisane obstoječe oz. možne algoritme za reševanje nonogramov ter se poglobila v genetskega. Na spletu sem poiskala tudi primere teh logičnih ugank, da sem nato lahko lažje začela s programiranjem.

V kodi sem to logično uganko zapisala kot dve tabeli, pri čemer ena hrani številke, ki so zapisane na levi strani nonograma ter nakazujejo kako je potrebno zapolniti posamezne vrstice, druga pa hrani številke, zapisane na vrhu logične sestavljanke, ki igralcu povedo, kako bodo v posameznem stolpcu pobarvani kvadratki.

Izbrala sem genetski algoritem in ga poskušala čimbolje implementirati. Najprej je bilo potrebno določiti oziroma izračunati *fitness*. To je številka, ki predstavlja število omejitev, ki zadoščajo igralni površini logične uganke. Torej, če imamo nonogram velikosti 10×10, lahko rečemo, da ima 20 omejitev, torej je $fitness = 20$. Številka izhaja iz tega, da ima tak nonogram 10 omejitev na

levi strani mreže ter 10 omejitev na zgornji strani. Izbrala sem tudi velikost populacije, ki sem jo dobila s poskušanjem, katera je dala najboljši rezultat. Ko sem poskusila to velikost nastaviti na 10, je algoritem deloval zelo slabo ter opravil ogromno iteracij že pri preprostih nonogramih. S poskušanjem sem ugotovila, da deluje precej bolje že kadar je ta številka večja od 100. Na primer pri preprostem nonogramu, katerega rešitev je oblika srca, sem ob nastavljeni visoki cifri za velikost populacije dobila pogosto tudi samo eno iteracijo, ko pa je bila velikost nastavljena na 10, sem dobila tudi 96 iteracij.

Kot je torej značilno za genetski algoritem, sem najprej generirala naključno populacijo možnih rešitev nonograma. Definirala sem funkcijo, ki izračuna oziroma oceni kolikšna je številka *fitness*-a. Nato pa se je začel cikel ponavljanja, kot je prikazano na spodnji sliki.

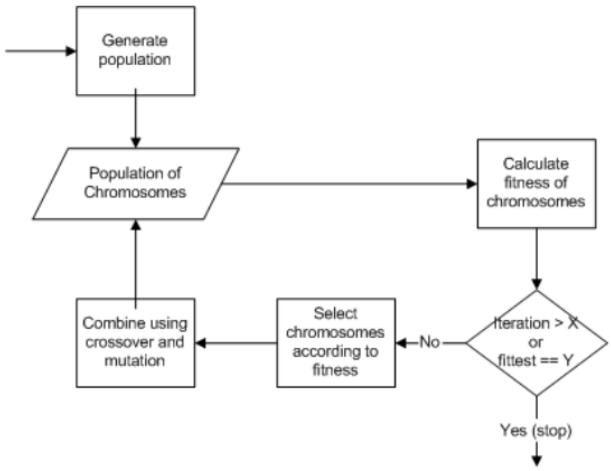

Figure 2: Diagram poteka genetskega algoritma

Ustvarja se nova populacija s ponavljanjem *selekcije*, *crossover*, *mutacije* in *sprejemanja*. S *selekcijo* algoritem najprej izbere dve rešitvi iz populacije, glede na njihovo vrednost *fitness*. Torej, boljša kot je cifra, več je možnosti, da bo izbrana. Te dve izbiri nato poimenujem za starša. Z verjetnostjo *crossover* gremo preko staršev, da dobimo novega potomca. Ta nastane tako, da zgeneriram naključno točko in vse pred to točko potomec podeduje po prvem staršu, vse za to točko pa potomec podeduje po drugem staršu. V vsakem koraku tudi preverimo ali je zadoščeno vsem omejitvam ali je potrebno še naprej iskati rešitev. Nato z verjetnostjo *mutacije* mutiram novega potomca z vsako pozicijo v začetni populaciji. Znova preverjam pogoje oz. omejitve.

Na koncu se še, kot že omenjeno, preveri, ali so vsi pogoji oz. vse omejitve izpolnjene in v kolikor so, algoritem vrne izrisano rešitev nonograma. Poleg tega sem naredila tudi števca, ki nakoncu podata čas izvajanja v sekundah ter koliko iteracij je bilo potrebnih, da je algoritem prišel do rešitve, torej pobarvanja

uganke.

5 Možne izboljšave

Na začetku programa sta definirani številki za verjetnosti *crossover* in *mutacije*. S testiranjem raznoraznih kombinacij sem ocenila, katere so tiste številke, ki vrnejo najboljše rezulate. To se bi za še bolj optimalne rezultate seveda dalo še izboljšati, vendar tudi dobljeni rezultati niso slabi.

V primeru, da moja implementacija genetskega algoritma prejme nek nerešljiv nonogram, se bo koda izvajala v neskončnost. Dobro bi bilo dodati neko omejitev, ki bi po določenem času ali pa po določenem številu iteracij uporabniku javila, da uganka ni rešljiva.

6 Rezultati

Moj program za reševanje nonograma velikosti 9×9 v povprečju porabi 2.2 iteracije, da pride do pravilne končne rešitve. Za eno iteracijo program vedno porabi manj kot 0.1 sekunde časa.

Manjše nonograme ta implementacija reši zelo hitro, pri večjih pa se čas reševanja občutno poveča, vendar vseeno na koncu najde rešitev.

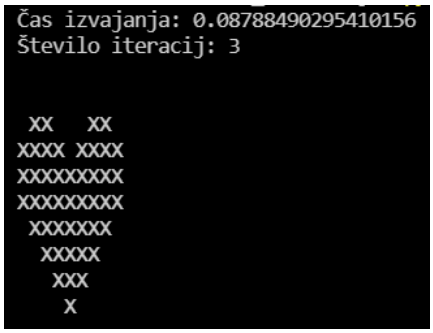

Figure 3: Prikaz rešitve preprostega nonograma 9×9

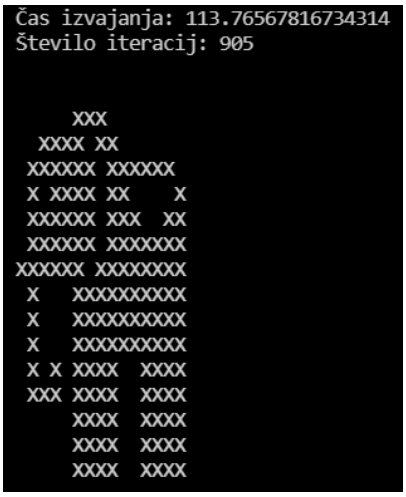

Figure 4: Prikaz rešitve zahtevnejšega nonograma 15×15