Универзитет у Београду

Машински факултет

Мастер смер ИНДУСТРИЈА 4.0

Рачунарска интелигенција

Пројекат: Примена генетских алгоритама у решавању проблема терминирања технолошких процеса

Професори:

проф. др. Александар Картељ

асистент Денис Аличић

Студенти:

Дејан Благојевић 4004/19

Огњен Збиљић 4002/20

Немања Јанев 4004/20

Увод

Промене са којима се суочавају компаније на почетку овог века резултат су утицаја више фактора, од којих је један глобализација. Како би компанија ефикасно одговорила на потражњу глобалног тржишта, није више довољно да њен производни систем само прави огромну количину производа, већ је потребно да то може извршавати уз велику варијацију врста производа. То условљава да се њихове машине које чине производни систем мора променити из наменских машина које су могле изводити искључиво један посао у машине које имају могућност да обављају више послова. Ту настаје проблем организације и доделе послова машинама, јер систем у којем се планира да једна машина ради стално само један те исти део посла на сваком производу, једноставно не би био ефикасан и исплатив, јер је сада могуће тај део посла распоредити на више машина. Али како одабрати којој машини доделити који посао, у ком тренутку и којим редоследом, а да је време за које ће се испунити неки предвиђени план производње, најмање? Оптимизациони алгоритам, који је тема овог рада, пружиће одговор на ово питање.

Овај проблем (проблем терминирања, енг. *scheduling*) је популаран за учење и вежбање еволутивних алгоритама, па има доста доступних функционалних решења овог проблема. Пример алгоритма који кодира ово решење пронађен је на сајту Групе програмера [2]. Овај алгоритам је кодиран преко класа по чему се разликује од алгоритма који је предмет овог пројектног задатка.

Као улазни подаци за алгоритам који се морају дефинисати у поставци проблема су:

1. Расположиви број машина у систему
2. Број послова који је потребно одрадити
3. Структура сваког посла, тј. на којим машинама се треба одрадити посао и којим редом
4. Време које је потребно да сваки посао проведе на појединачној машини
5. Време транспорта делова између машина

Излаз из алгоритма представља најоптималнији распоред послова по машинама, по критеријуму потребног времена за завршетак свих послова.

Опис решења

Терминирање (енгл. *scheduling*) представља примену оптимизационих метода и метода одлучивања у циљу планирања и оптималног временског распоређивања активности производно-технолошких ентитета. У овом пројектном задатку под појмом терминирање се подразумева терминирање флексибилних машина алатки у складу са усвојеним критеријумима. У процесу терминирања флексибилних машина алатки свака операција се додељује одговарајућој машини алатки. Као резултат се добија редослед операција делова на одговарајућим машинама алаткама.

Најчешћи проблем терминирања машина алатки се своди на одређивање максималног производног времена обраде свих делова (енгл. *makespan*).

Примена генетичких алгоритама у оптимизацији плана терминирања

У циљу отимизације технолошког процеса и плана терминирања често се користе генетички алгоритми. Да би се извршила оптимизација плана терминирања потребно је правилно извршити сваки од следећих 5 корака:

- генерисање јединки у иницијалној популацији;
- евалуација функције циља и иницијализација параметара генетичких алгоритама;
- селекција;
- укрштење и
- мутација.

Генерисање јединки у иницијалну популацију

Приликом генерисања јединки у иницијалну популацију код оптимизације плана терминирања важно је напоменути да се посматра укупан процес за све делове који учествују у плану терминирања.

Иницијална популација за генетички алгоритам се састоји од једног стринга тј. хромозома (план терминирања).

Дужина хромозома за план терминирања одређена је бројем делова, тј. дужина хромозома одговара броју делова који учествују у терминирању.

1	3	0	4	2

Слика 1: Стринг за план терминирања (јединка у популацији)

```
# generisanje inicijalne populacije

#np.random.seed(6)

population_list=[] #lista gde popunjavamo populaciju
for i in range(population_size): #range(population_size) #trebalo bi populaciju da popunis u skladu sa velicinom poplacijue
   generisana_jedinka=list(np.random.permutation(br_poslova)) #pravimo permutaciju od M poslova(Jobs)
   population_list.append(generisana_jedinka)

populacija = population_list #nama je populacija u stvari lista prioriteta obrade
populacija
```

Прорачун функције циља

Алгоритам функције циља за прорачун вредности користи податке из више променљивих, углавном листи и матрица које су попуњене псеудо-случајним бројевима и чије димензије зависе од броја послова, броја машина или истовремено од оба броја. Функција на почетку креира 2 низа попуњене нултим вредностима, један дужине као и број машина, други дужине броја послова. Свака позиција у тим низовима представља једну машину, тј. један посао. Потом се пролази кроз једнику која, како је већ речено представља пермутацију послова, и по том приоритету делове тих послова додељује потребним машинама пратећи збир времена додељених делова по машинама и по пословима. Пошто се делови једног посла морају додавати редом, може се десити да ако на некој машини претходни део посла није завршен, следећа машина мора чекати претходну како би преузела посао. Наравно ситуација се усложњава увођењем и времена транспорта послова са машине на машину.

```
def fitness_function(populacija):

   funkcija_cilja=[]
   for jedinka in populacija:
      max_vreme_masina = np.zeros(br_masina, dtype=int)
      max_vreme_job = np.zeros(br_poslova, dtype=int)
      for i in range(0,len(masine)):
         #print("redni broj operacije: ", i)
         for job in jedinka:
            if len(lista_masina_za_svaki_posao[job]) <= i:
               continue
            #print("job", job)
            trenutna_masina = lista_masina_za_svaki_posao[job][i]
            #print("trenutna masina", lista_masina_za_svaki_posao[job][i])
            trajanje = processing_time[trenutna_masina][job]
            #print("duzina trajanja posla: ",processing_time[trenutna_masina][job])
            slobodno = max(max_vreme_masina[trenutna_masina],max_vreme_job[job])
            #print("Slobodno",slobodno)
            if i<(len(lista_masina_za_svaki_posao[job])-1):
               transport=
Vreme_transporta[lista_masina_za_svaki_posao[job][i]][lista_masina_za_svaki_posao[job][i+1]]
            else:
               transport=0
            #print(transport)
            max_vreme_masina[trenutna_masina] = slobodno + trajanje
            max_vreme_job[job] = slobodno + trajanje + transport
            #print("max_vreme_masina",max_vreme_masina, "max_vreme_job", max_vreme_job)
```

```
    funkcija_cilja.append(max(max_vreme_masina))

  return funkcija_cilja
```

Селекција

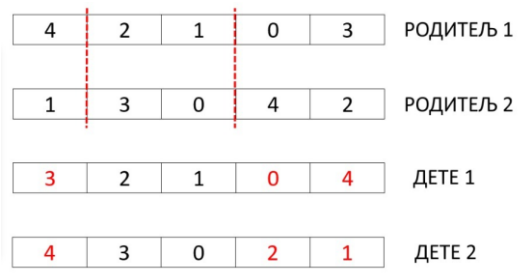
Под појмом селекције подразумева се бирање два родитеља тј. хромозома из текуће популације од којих ће настати нови хромозоми тј. деца. Бирање се врши на основу турнир селекције, где је вероватноћа селекције у зависности од израчунате функције циља (fitness_function) из текуће популације. Селекција се у алгоритму извршава онолико пута колико има елемената у популацији када се одузме број елитних јединки и одабране јединке селекцијом заједно са елитним јединкама чине нову популацију.

```
#turnirska selekcija

def selection(funkcija_cilja):

   #random.seed(5)
   clanovi_turnira= 1 #koliko ce jedinki uci na turnir

   k = -1
   winner= 1000000
   for i in range(clanovi_turnira):
      j = random.randrange(len(populacija))
      #print("j: ", j)
      if funkcija_cilja[j] < winner:
         winner = funkcija_cilja[j]
         #print("f-ja cilja: ", funkcija_cilja[j])
         k=j
   return k
```

Укрштање

Укрштање представља оператор који се примењује како би се од родитеља добили потомци (деца).

Укрштање се врши тако што се на случајан начин врши одабир позиција у стрингу првог родитеља и вредности између одабраних позиција се додељују првом потомку на истим позицијама. Непопуњена места првог потомка се попуњавају редом елементима из другог родитеља који недостају. Исти поступак се примењује и за формирање другог потомка. Резултат укрштања на случајно одабраним родитељима са 5 делова је приказан на слици.

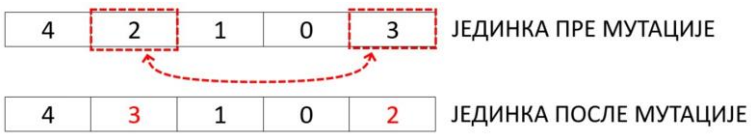


Слика 2: Резултат укрштања на случајно одабраним родитељима са 5 послова

```
#CROSSOVER
def crossover (roditelj1, roditelj2):

    #np.random.seed(5)
    #print("roditelji: ",roditelj1, roditelj2)
    srecan_broj= 0.81 #random.random()
    if srecan_broj<crossover_rate: #PROVERITI OVO, sta vraca i sta treba da pise < ili >!!!
        return (list(roditelj1),list(roditelj2))
    cutpoint=list(np.random.choice(len(roditelj1), 2, replace=False)) #biram dva mesta za crossover
    cutpoint.sort()
    #print(cutpoint)


    dete1 = np.zeros(len(roditelj1), dtype=int)
    dete2 = np.zeros(len(roditelj2), dtype=int)

    dete1[:][:]=-1
    dete2[:][:]=-1

    dete1[cutpoint[0]:cutpoint[1]] = roditelj1[cutpoint[0]:cutpoint[1]]
    dete2[cutpoint[0]:cutpoint[1]] = roditelj2[cutpoint[0]:cutpoint[1]] # ne ukljucuje desnu poziciju, do nje
preslikava

    list_difference1 = [item for item in roditelj2 if item not in dete1]
    list_difference2 = [item for item in roditelj1 if item not in dete2]
    #print("razlike: ",list_difference1,list_difference2)
    poz_zamene_d1=0
    poz_zamene_d2=0

    for i in range(len(dete1)):
        if dete1[i] == -1:
            dete1[i]=list_difference1[poz_zamene_d1]
            poz_zamene_d1=poz_zamene_d1+1
        if dete2[i] == -1:
            dete2[i]=list_difference2[poz_zamene_d2]
            poz_zamene_d2=poz_zamene_d2+1
    #print("deca: ", dete1, dete2)
    return (list(dete1),list(dete2))
```

Мутација

Оператор мутације се врши на основу дефинисане вероватноће мутације. Оператор мутације је двопозициона замена (енгл. *swapping*). Мутација се врши тако што се одабере један родитељ, а затим се случајно одаберу два гена у том родитељу којима се замене места као што је приказано на слици испод. Потомак представља нови хромозом који је добијен заменом места случајно одабраних гена.



Слика 3: Пример мутације

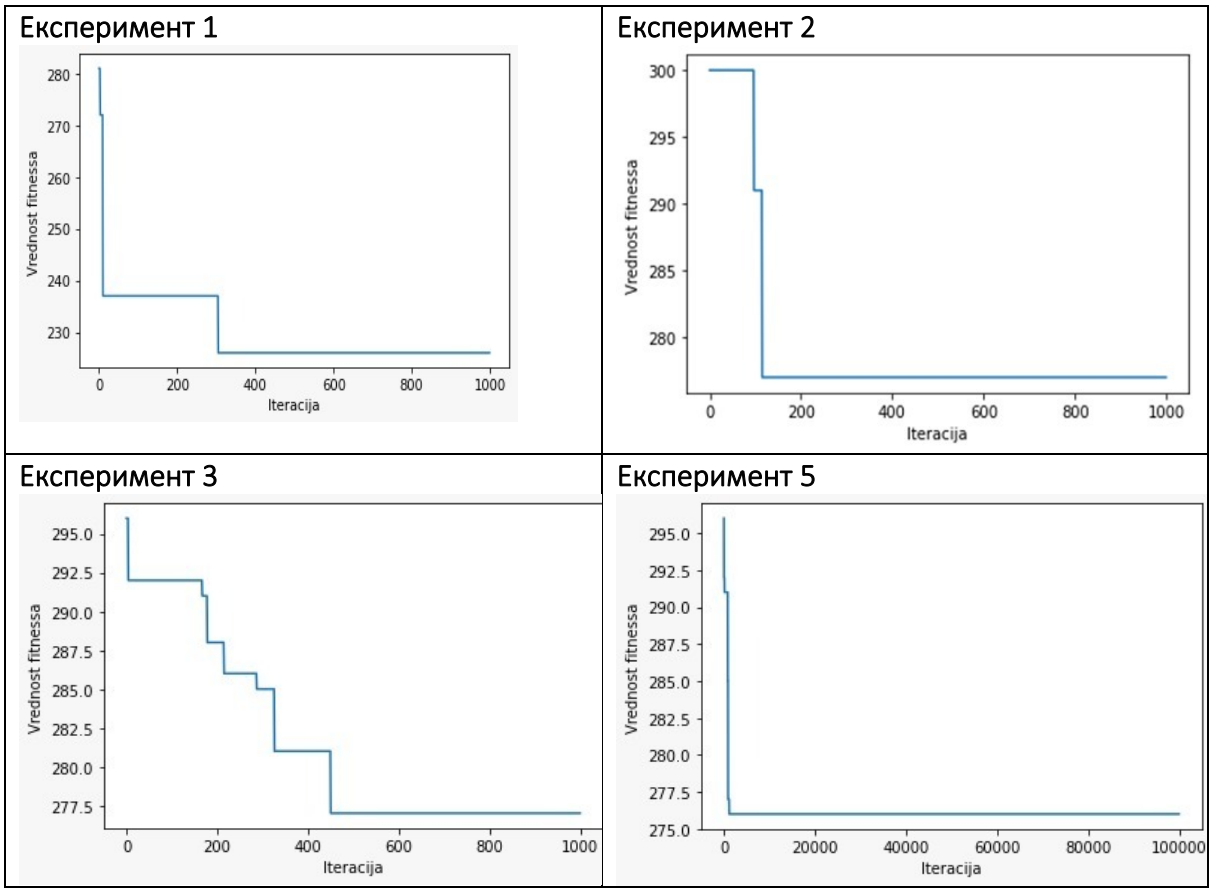
```
#MUTACIJA
def mutacija (dete):
    #print("pre mutacije: ",dete)
    srecan_broj = random.random()
    if srecan_broj<= mutation_rate:
        indexi= list(np.random.choice(len(dete), 2, replace=False))
        vrednost_na_mestu1 = dete[indexi[0]]
        vrednost_na_mestu2 = dete[indexi[1]]

        #print(indexi)
        #print ( vrednost_na_mestu1, vrednost_na_mestu2)
        dete[indexi[0]], dete[indexi[1]] = vrednost_na_mestu2, vrednost_na_mestu1

        #print("posle mutacije: ",dete)
    return(dete)
```

Експериментални резултати

Табела 1. Резултати тестирања генетског алгоритма и алгоритма грубе силе

		Генетски алгоритам			Алгоритам Грубе силе		
Број машина	Број послова	Број јединки , Број генерација	Функција циља	Време извршења	Број пермутација	Функција циља	Време извршења
4	8	5, 1000	205	0.0684	40320	205	1.55
3	10	5, 1000	226	0.6197	3 628 800	226	131.5797
4	12	5, 1000	277	0.773	479 001 600	Грешка меморије	/
4	11	5, 1000	277	0.7578	39 916 800	276	1900.113
4	11	5, 100000	276	71.5393	39 916 800	276	1900.113

Табела 2. Графички приказ промене најмање вредности ф-је циља кроз генерације у експериментима

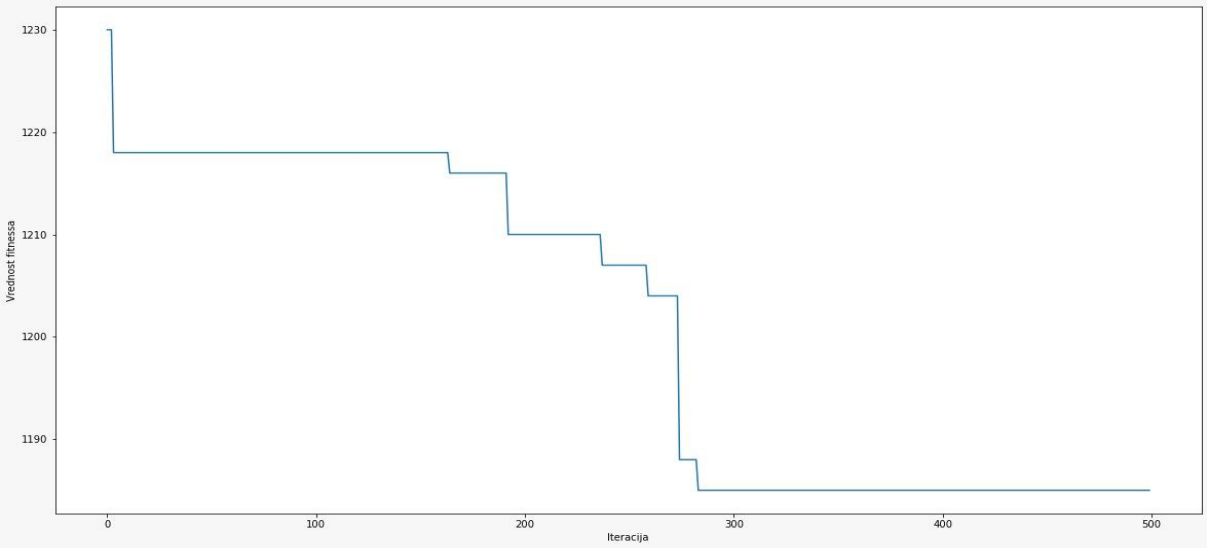


Експерименти су показали да ГА алгоритам ради са одличном тачности и проналази оптимално решење, осим у случајевима када има мали број генерација, али пошто је време извршавања изузетно мало, велики број генерација не представља проблем.

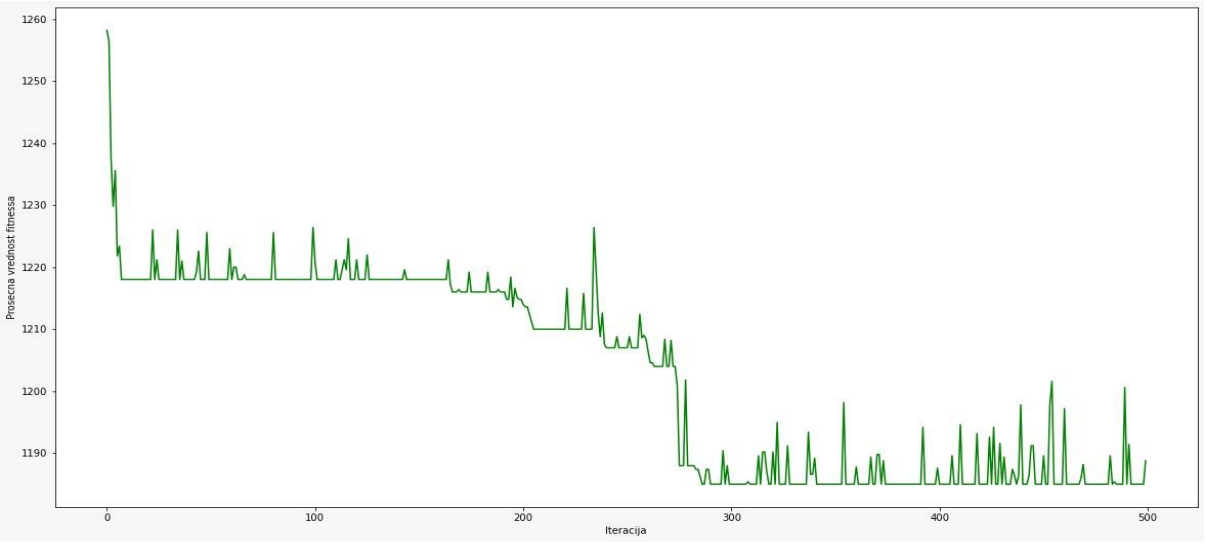
Показни пример

Подаци:

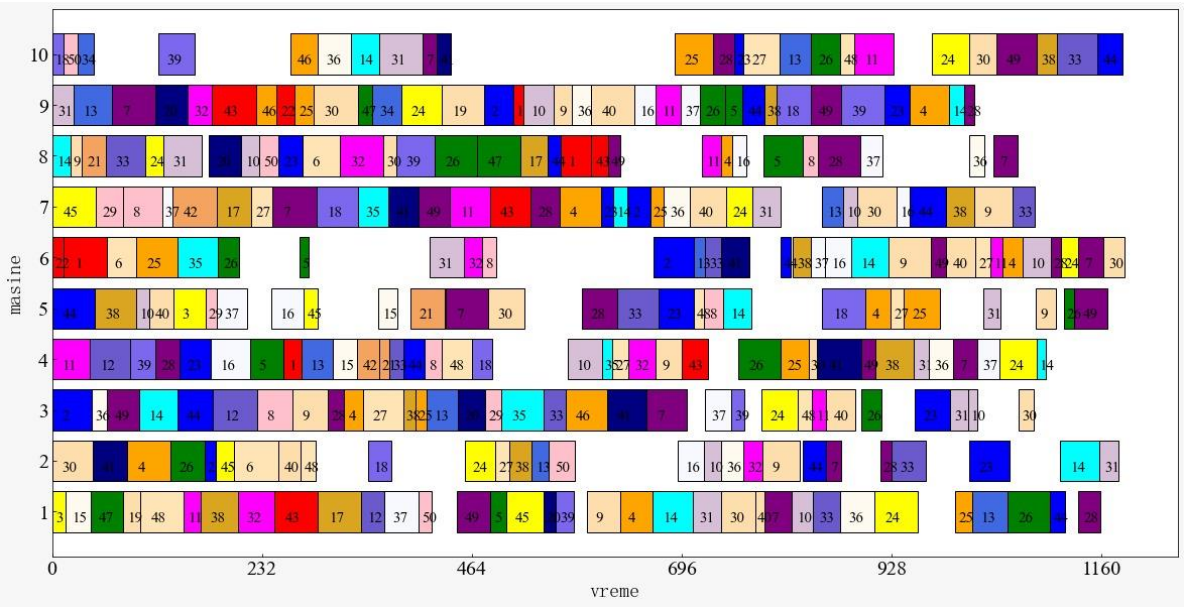
- Број послова: 50
- Број машина: 10
- Величина популације: 5 јединки
- Број генерација: 500
- Број елитних јединки: 1 јединка
- Величина турнира при селекцији: 2 јединке
- Шанса за мутацију: 10%



Слика 4. Промена минималне функције циља кроз итарације



Слика 5. Промена просечних вредности функција циља свих јединки у популацији кроз итерације



Слика 6. Најбоља јединка (решење) представљена гантограмом

Као резултат оптимизације терминирања технолошких система добијен је Гантов дијаграм, који приказује оптималан распоред свих активности потребних за израду претходно одабраних репрезентативних делова чије се терминирање врши. Као критеријум оптимизације у обзир је узето производно време и време транспорта између машина.

Време извршавања овог теста генетским алгоритмом је 41 секунд, док би за израчунавање вредности функције циља свих пермутација којих има 3 041 409 320 x 10^{64} вероватно потрајало више милијарди година (рачунајући да време има линеарни раст). Наравно, огромни проблем би представљао и заузети меморијски простор.

Рачунар на којем су се изводили експерименти поседује оперативни систем *Windows 10 Pro 64-bita*, *Intel* процесор *I7-2670QM*, 8 *GB RAM* меморије 516 *GB SSD* меморије.

Закључак

Експерименти су показали да ГА алгоритам ради са задовољавајућом тачности и увек проналази решење које не мора бити оптимално, али је довољно добро, осим у случајевима када има мали број генерација, али пошто је време извршавања мало, велики број генерација не представља проблем. У поређењу са алгоритмом грубе силе који успешно проналази оптимално решење, ГА је неупоредиво бржи и ефикаснији од алгоритма грубе силе код кога се, такође, сусрећемо са проблемом повећаног коришћења меморије и повећањем броја послова долази до грешака у меморији компајловања.

У овом раду, разматрани проблем терминирања поједностављен је и сведен на флексибилност машина и редоследа операција. Сложенији проблеми оптимизације обухватали би и флексибилност процеса, алата, као и оријентације алата.

Овим експериментом је утврђено да је ГА боља опција од алгоритма грубе силе за решавање проблема терминирања проиводње, али пожељно је упоредити га са још неким алгоритмом оптимизације како бисмо имали ширу слику перформанси ГА на овом пољу решавања проблема.

Литература

1. Изводи са предавања *Рачунарска интелигенција*, Машински факултет, година 2021/2022.
2. *Programmer group*, интернет страница, https://programmer.group/613db68ec2dec.html, датум приступа, јун 2022.