

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО**

**ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«Донской государственный технический университет»**

**(ДГТУ)**

Кафедра «Программное обеспечение вычислительной техники и

автоматизированных систем»

**ЛАБОРАТОРНАЯ РАБОТА № 6**

по дисциплине «Эвристические методы и алгоритмы»

тема: «Методы получения расписаний для однородных систем обработки информации с использованием модификаций генетического алгоритма»

Выполнил:

ст. гр. ВПР 31 Д. С. Кононов

Проверил:

д.н., профессор В. Г. Кобак

Ростов-на-Дону

2020

1. **Введение**

Предметом области исследования расписаний является круг задач проектирования и организационного управления в различных системах, в которых требуется найти наилучшее (оптимальное) значение выбранных критериев их функционирования с учетом имеющихся ограничений.

Программирование для многопроцессорных машинных систем связано с распараллеливанием и синхронизацией вычислений и организацией выполнения параллельных вычислительных процессов. Это выдвигает целый ряд сложных задач, среди которых весьма важными являются, расчет характеристик времени и количества операций, требующихся для выполнения параллельных программ, и построения расписаний (планов), выполнения параллельных программ на многопроцессорных и многомашинных вычислительных системах.

Модели параллельных программ и операционные характеристики процессов их выполнения служат основой для планирования параллельных вычислительных процессов, т.е. для построения расписаний указанных процессов. Расписания параллельных вычислительных процессов определяют порядок выполнения программы на вычислительной системе, включая распределение частей программы по процессам. С увеличением числа распределяемых частей программ и количества используемых процессоров сложность построения оптимальных расписаний обычно резко возрастает. Поэтому важное значение имеют простые в построении и удобные в реализации приближенные расписания параллельных вычислительных процессов, близкие к оптимальным с точки зрения времени выполнения параллельных программ.

Генетические алгоритмы (ГА) являются одной из парадигм эволюционных вычислений, представляют собой алгоритмы поиска лучшего, а не оптимального решения задачи, построены на принципах, сходных с принципами естественного отбора и генетики.

ГА имеет вероятностную природу и в связи с этим результаты, получаемые с помощью него, отличаются в каждом запуске и определяются случайной последовательностью, переданной в схему алгоритма. Точность алгоритма зависит не только от входной последовательности случайных чисел, но и от условий задачи, таких как размерность задачи и конкретное распределение весов.

# Постановка задачи

Имеется вычислительная система (ВС), состоящая из  несвязанных идентичных устройств (приборов, процессоров и т.п.) .

На обслуживание в ВС поступает набор из  независимых параллельных заданий (работ) , известно время решения  задания  на любом из устройств. При этом каждое задание может выполняться на любом из устройств (процессоре), в каждый момент времени отдельный процессор обслуживает не более одного задания и выполнение задания не прерывается для передачи на другой процессор. Требуется найти такое распределение заданий по процессорам, при котором суммарное время выполнения заданий на каждом из процессоров было бы минимальным. Под расписанием следует понимать отображение , такое, что если , то говорят, что задание  в расписании , назначенного на процессор . При сделанных выше допущениях расписание можно представить разбиением множества заданий на  непересекающихся подмножеств , где .

Критерий, используемый для минимизации времени завершения обслуживания заданий, является минимальным критерием и определяется в следующем виде: , где  - время завершения работы процессора .

# Генетические алгоритмы

# Стандартная схема работы генетического алгоритма

Генетические алгоритмы (ГА) являются одной из парадигм эволюционных вычислений, представляют собой алгоритмы поиска лучшего, а не оптимального решения задачи, построены на принципах, сходных с принципами естественного отбора и генетики.

ГА имеет вероятностную природу и в связи с этим результаты, получаемые с помощью него отличаются в каждом запуске и определяются случайной последовательностью, переданной в схему алгоритма. Точность алгоритма зависит не только от входной последовательности случайных чисел, но и от условий задачи, таких как размерность задачи и конкретное распределение весов.

Рассмотрим общую схему работы генетического алгоритма:

Ш.1 Формируется начальное поколение, состоящее из заданного числа особей.

Ш.2 Отбор особей и применение ГА операторов кроссовера и мутации с заданной вероятностью для создания нового поколения.

Ш.3 Проверка условия останова, которая обычно заключается в неизменности лучшего решения в течение заданного числа поколений. Если проверка прошла не успешно, то переход на Ш.2.

Ш.4 Лучшая особь выбирается как найденное решение.

ГА является общим алгоритмом для решения любой задачи, и при его применении к конкретной необходимо выбрать механизм кодирования параметров задачи (фенотипа) в гены особи (генотипа), определить оптимизационную функцию  (fitness function) и выбрать условия останова.

В данном случае решения задачи теории расписания минимаксный критерий будет являться оптимизационной функцией, а условием останова - неизменность лучшего решения в течение заданного числа поколений.

Введем следующие обозначения:

 - число особей (хромосом);

 - условие останова по количеству одинаковых поколений;

 - вектор особей (хромосом);

 - вектор приспособленности;

 - вероятность оператора кроссовера;

 - оператор кроссовера;

 - вероятность оператора мутации;

 - оператор мутации.

Рассмотрим подробнее Ш.2 алгоритма:

Ш.2.1 На вход шага поступает исходный вектор особей (хромосом) , каждая из особей представляет какое-то расписание с соответствующим ему  и соответственно приспособленностью особи.

Ш.2.2 , где  - порядковый номер особи в векторе .

Ш.2.3 Выполняется оператор кроссовера с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то , где - случайное число в интервале [1..], исключая совпадение с числом , иначе особь переходит в новое поколение без изменений, т.е. .

Ш.2.4 Выполняется оператор мутации с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то .

Ш.2.5 . Если , то перейти к Ш.2.3.

Ш.2.6 Сформировано новое поколение . Осуществляется переход на Ш.3.

* 1. **Модификация оператора кроссовера**

Модификация заключается в изменении оператора кроссовера так, чтобы для очередной особи случайным образом выбиралось две особи вместо одной, и скрещивание проводилось с наилучшей из этих двух особей.

Рассмотрим подробнее Ш.2 алгоритма:

Ш.2.1 На вход шага поступает исходный вектор особей (хромосом) , каждая из особей представляет какое-то расписание с соответствующим ему  и соответственно приспособленностью особи.

Ш.2.2 , где  - порядковый номер особи в векторе .

Ш.2.3.1 Выполняется оператор кроссовера с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1].

Ш.2.3.2 Если , то генерируем два случайных числа  и  в интервале [1..], исключая совпадения этих чисел друг с другом и числом , иначе особь переходит в новое поколение без изменений, т.е. , и переходим на Ш.2.4.

Ш.2.3.3 Если особь  окажется лучше особи , то , иначе операция кроссовера будет .

Ш.2.4 Выполняется оператор мутации с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то .

Ш.2.5 . Если , то перейти к Ш.2.3.1.

Ш.2.6 Сформировано новое поколение . Осуществляется переход на Ш.3.

* 1. **Модификация турнирного отбора**

Модификация заключается в изменении турнирного отбора, т.е. после применения ГА операторов необходимо сравнить очередную особь, для которой применялись эти операторы, с результирующей особью, и та, которая окажется лучше, перейдет в новое поколение.

Рассмотрим подробнее Ш.2 алгоритма:

Ш.2.1 На вход шага поступает исходный вектор особей (хромосом) , каждая из особей представляет какое-то расписание с соответствующим ему  и соответственно приспособленностью особи.

Ш.2.2 , где  - порядковый номер особи в векторе .

Ш.2.3 Выполняется оператор кроссовера с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то , где - случайное число в интервале [1..], исключая совпадение с числом , иначе особь переходит в новое поколение без изменений, т.е. .

Ш.2.4 Выполняется оператор мутации с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то .

Ш.2.5 Если особь  окажется лучше результирующей особи , то .

Ш.2.6 . Если , то перейти к Ш.2.3.

Ш.2.7 Сформировано новое поколение . Осуществляется переход на Ш.3.

**4. Реализации алгоритма половинного деления множества заданий Python**

from random import randint, choice

class HDA:

def krit(self, mass, n):

p = [[] for i in range(n)]

for i in mass:

sums = list(map(sum, p))

p[sums.index(min(sums))].append(i)

return p

def HDA(self, mass, proc):

Pa, Pb = self.krit(mass, 2)

Pa = self.krit(Pa, proc // 2)

Pb = self.krit(Pb, proc // 2)

P = Pa + Pb

sums = list(map(sum, P))

return sums

def main(self, mass, proc):

return max(self.HDA(list(reversed(sorted(mass))), proc))

class GA:

def \_\_init\_\_(self, Osob=10, Gen=10, Pcross=0.8, Pmut=0.8, Lim=3, Left=1, Right=100, Nproc=4):

self.C = [[randint(Left, Right) for i in range(Gen)] for i in range(Osob)]

self.Pcross = Pcross

self.Pmut = Pmut

self.Lim = Lim

self.Left = Left

self.Right = Right

self.Nproc = Nproc

def crossover\_first(self, left, right):

if randint(0, 100) / 100 <= self.Pcross:

new\_osob = []

for i in range(len(left)):

if randint(0, 5) == 0:

new\_osob.append(right[i])

elif right[i] > left[i]:

new\_osob.append(left[i])

else:

new\_osob.append(right[i])

return new\_osob

def crossover\_second(self, left, middle, right):

if randint(0, 100) / 100 <= self.Pcross:

new\_osob = []

if sum(middle) < sum(right):

osob = middle

elif sum(middle) > sum(right):

osob = right

else:

return None

for i in range(len(left)):

if randint(0, 3) == 0:

new\_osob.append(osob[i])

elif osob[i] > left[i]:

new\_osob.append(left[i])

else:

new\_osob.append(osob[i])

return new\_osob

def mut(self, obj):

if randint(0, 100) / 100 <= self.Pmut:

obj[choice(obj)], obj[obj.index(min(obj))] = obj[obj.index(min(obj))], obj[obj.index(min(obj))]

def check\_end(self, T):

for i in range(len(self.C)):

if HDA().main(self.C[i], self.Nproc) != T[i]:

return False

else:

return True

def default(self):

new\_C = self.C.copy()

left = choice(new\_C)

del new\_C[new\_C.index(left)]

right = choice(new\_C)

cross = self.crossover\_first(left, right)

if cross:

self.C[self.C.index(left)] = cross

def modification(self):

new\_C = self.C.copy()

left = choice(new\_C)

del new\_C[new\_C.index(left)]

middle = choice(new\_C)

del new\_C[new\_C.index(middle)]

right = choice(new\_C)

cross = self.crossover\_first(left, right)

if cross:

self.C[self.C.index(left)] = cross

def tournament(self):

for left in self.C:

new\_C = self.C.copy()

del new\_C[new\_C.index(left)]

right = choice(new\_C)

cross = self.crossover\_first(left, right)

if cross:

self.C[self.C.index(left)] = cross

def main\_first(self):

count = 0

repeats = 0

print('Generation quality at the start:', min([HDA().main(i, self.Nproc) for i in self.C]))

print()

while repeats < self.Lim:

count += 1

T = [HDA().main(i, self.Nproc) for i in self.C]

self.default()

if self.check\_end(T):

repeats += 1

else:

repeats = 0

print('Standard GA generation:')

self.output\_mass()

print('Generation quality at the end of standard GA:', min([HDA().main(i, self.Nproc) for i in self.C]))

print('All generations:', count)

print()

def main\_second(self):

count = 0

repeats = 0

while repeats < self.Lim:

count += 1

T = [HDA().main(i, self.Nproc) for i in self.C]

self.modification()

if self.check\_end(T):

repeats += 1

else:

repeats = 0

print('Modification GA generation:')

self.output\_mass()

print('Generation quality at the end modification GA:', min([HDA().main(i, self.Nproc) for i in self.C]))

print('All generations:', count)

print()

def main\_tournament(self):

count = 0

repeats = 0

while repeats < self.Lim:

count += 1

T = [HDA().main(i, self.Nproc) for i in self.C]

self.tournament()

if self.check\_end(T):

repeats += 1

else:

repeats = 0

print('Tournament GA generation:')

self.output\_mass()

print('Generation quality at the end tournament GA:', min([HDA().main(i, self.Nproc) for i in self.C]))

print('All generations:', count)

print()

def output\_mass(self, mass=None):

#print('-'\*25)

if not mass:

mass = self.C

for i in range(len(mass)):

print('Gene vector №{} = '.format(i + 1), mass[i])

print('-'\*25)

if \_\_name\_\_ == '\_\_main\_\_':

a = GA(Lim=3)

print('Primary generation:')

a.output\_mass()

a.main\_first()

a.main\_second()

a.main\_tournament()

1. **Результаты, выводы**

В ходе лабораторной работы была написана программа, реализующая генетические алгоритмы, такие как: стандартная схема работы генетического алгоритма, модификация оператора кроссовера, модификация турнирного отбора.

Также было произведено сравнение работы алгоритмов. Выяснили, что алгоритм позволяет получить точной результат, при использовании варианта модификации турнирного оператора отбора. Остальные результаты являются менее точными и оптимальными.

Результат работы программы:



