САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

Прикладная математики и информатика

Исследования операций и принятия решений в задачах оптимизации,
управления и экономики

Васильев Михаил Александрович

Иммунный алгоритм составления расписаний

Бакалаврская работа

Научный руководитель:
к. ф.-м. н., доцент Григорьева Н. С.

Рецензент:
к. ф.-м. н., доцент Агафонова И. В.

Санкт-Петербург
2016

SAINT PETERSBURG STATE UNIVERSITY
Applied Mathematics and Computer Science
Operation Research and Decision Making in Optimisation, Control and Economics Problems

Vasilev Mikhail

Immune algorithms of schedule constructing

Bachelor's Thesis

Scientific supervisor:
docent Grigorieva N. S.

Reviewer:
docent Agafonova I. V.

Saint Petersburg
2016

Содержание

Введение

В течение жизни мы регулярно сталкиваемся с расписаниями: транспорт, телевизионные передачи, кино, занятия в школе и институте и прочее. Для решения таких вопросов мы используем интуитивный подход. Однако в таких отраслях как автоматизация объёмного производства требуется оптимизировать общее время работы. Теория расписаний — как направление в исследовании операций началось с работы Г. Ганта в 1910, представившим диаграммы для управления проектами. Термин «теория расписаний» был предложен Р. Белламном в 1956 году. Большинство задач теории расписаний являются *NP - трудными*, поэтому решение задач больших размерностей требует больших временных затрат. К решению полиномиально неразрешимых задач применяют эвристики — алгоритмы, дающие приближенное решение за приемлемое время. Схемы, объединяющие основные эвристические методы и более эффективно исследующие пространство поиска, называемые *метаэвристиками* были представлены Ф.Гловером в 1986 году. В данной работе будет представлен *метаэвристический алгоритм* решения задачи о расписании, представленной сетевым графиком, его реализация и применение.

1. Постановка задачи

Рассмотрим задачу из класса **MSP** (Multiprocessor scheduling problem): дано n заданий, m процессоров, на множестве заданий задано отношение частичного порядка: $\prec$.
Каждое задание i имеет время выполнения t_i, $i = 1 \dots n$.
На каждом процессоре в любой момент времени может выполняться не более одного задания.

Обозначим: r_i - время начала выполнения задания i.
Тогда длиной расписания назовём величину $C = \max\limits_{i=1\dots n}\{r_i + t_i\}$.
Необходимо построить расписание без прерываний, минимальной длины, то есть: $C \to \min$.

Интерпретация в виде графа:
$G = G\langle V, E\rangle$ — ориентированный граф.

V — множество заданий.
E — множество рёбер, где существование ребра e_{ij} означает что задание i предшествует заданию $j : i \prec j$, то есть j может начать выполняться только после выполнения i.

2. Алгоритмы решения

Задача **MSP** является *NP - трудной в сильном смысле*, то есть для нее не только не существует алгоритмов, решающих задачу за полиномиальное время, но и не существует быстрых алгоритмов, проверяющих что данное решение является оптимальным. Таким образом, для решения используются приближенные алгоритмы, дающие допустимое решение за приемлемое время. Одним из подклассов приближенных методов являются *метаэвристические алгоритмы*, преимуществом которых является исследование большего пространства поиска нахождения близкого к оптимальному решения.

В общем виде *метаэвристические алгоритмы* состоят из следующих компонент:

- Нахождение начального решения
- Поиск окрестностей для решения, определение переходов между ними
- Отбор кандидатов в решение из окрестностей, определение критериев принятия наилучшего решения
- Определение критерия остановки алгоритма

Метаэвристики состоят из двух категорий: методы локального поиска и эволюционные методы. Методы первой категории позволяют найти оптимум, модифицируя текущее решение. Эволюционные алгоритмы состоят из методов, модифицирующих некоторое множество текущих решений и последующем выборе лучшего.

В данной работе будет рассмотрен один из эволюционных методов — *Иммунный алгоритм*, а так же его применение для решения задачи **MSP**.

3. Иммунный алгоритм

3.1. Иммунные системы

Иммунитетом живых существ называется устойчивость и сопротивление организма чужеродным агентам и инфекциям, а так же минимизация воздействия от них. Иммунитет — одна из самых сложных систем организма, её способности и особенности положили в основу новое направления в информационных технологиях — *иммунокомпьютинг*.

Первые искусственные иммунные сети появились в 1980-х годах в публикациях Фармера, Паккарда и Перельсона. Первая книга об искусственных иммунных системах, за авторством Д. Дасгупты вышла в 1998 году.

В иммунологии чужеродные организмы называются *антигенами*. Для его уничтожения, иммунитет вырабатывает клетки — *антитела*, которые связываются с чужеродным агентом и уничтожают его. Основным способом обнаружения антител является сравнение определённых свойств (лимфоцитов) у различных агентов. Для сравнения используется принцип *негативной селекции*, заключающийся в том, что свойства, задаваемые иммунитетом, отсутствуют в организме, то есть если у агента обнаружены эти свойства, то он — чужой. Затем происходит *клональная селекция* — организм вырабатывает *антитела*, максимально похожие на *антигенов*, для последующего очищения от чужеродных тел.

3.2. Иммунный алгоритм в общем виде

В общем виде иммунный алгоритм для задачи оптимизации выглядит следующим образом:

Шаг 1: Создаётся *популяция* π — набор начально сгенерированных решений, фиксируется лучший экземпляр.

Шаг 2: Каждое решение из π клонируется на множество — cl и для каждого клона из cl применяется *мутации* — малое изменение решения.

Шаг 3: Среди мутированных клонов ищется лучший экземпляр и, в случае если он оказывается лучше чем исходное решение в популяции, то текущее решение в популяции, заменяется на лучший клон. Если не выполнен критерий остановки алгоритма, то возвращаемся на **Шаг 2**.

Критерием остановки алгоритма является достижение лимита по времени, достижение лимита по итерациям, отсутствие улучшения решения на протяжении многих итераций.

4. Алгоритм IASC

Для решения задачи **MSP** автором был разработан алгоритм **IASC (Immune algorithm for schedule constructing)** , основной идеей которого является локальное улучшение расписания на множестве «задание — предшественники».

Обозначим:
π - *популяция* — набор расписаний
$MT(i) = \{i, j_1, \dots j_k \mid j_l \prec i\}$ — *мутационное множество* решения i — есть набор из задания i и *множества его предшественников*.
Отметим, что $MT(i)$ — дерево с корнем в i.
Поставим в соответствие каждому допустимому решению из $\pi(i)$ — мутационное множество $MT(i)$. Именно на множестве $MT(i)$ каждое расписание $\pi(i)$ и будет осуществлять мутации.

Алгоритм:

Шаг 1: Генерируется начальное расписание π_0.

Шаг 2: Создаётся популяция π из n решений π_0.

Шаг 3: Для каждого решения $\pi(i)$ из π создаётся клон — $\pi(i)_r$.

Шаг 4: На $\pi(i)_r$ осуществляется мутация корня дерева $MT(i)$ — вершины i, высчитывается целевая функция мутированного клона $\pi(i)^*_r$.

Шаг 5: В случае, если решение $\pi(i)^*_r$ — допустимое, на нем осуществляется мутация потомков дерева $MT(i)$ — вершин $j_1 \dots j_k$. В противном случае, мутации производятся на исходном решении $\pi(i)$.

Шаг 6: Высчитывается целевая функция мутированного клона $\pi(i)^*_c$. Затем в качестве расписания в популяции $\pi(i)$ берётся решение с лучшим временем из $\pi(i)$, $\pi(i)^*_r$, $\pi(i)^*_c$.
После обхода всей популяции, в случае если не выполнен критерий остановки алгоритма, переход на **Шаг 2**.

4.1. Мутации

4.1.1. Мутации для корня MT

1) Пусть i — корень дерева MT, а $pr(i) = \{j_1 \dots j_k\}$ — предшественники задания i.

2) Выбирается базовое задание j_1.

3) Осуществляется перестановка задания i в расписании так, что бы оно стояло после базового, либо параллельно с ним. Соответственно задание x, стоящее на выбранном месте, ставится на то место, на котором было i.

3*) В случае, если множество предшественников — пусто, то есть задание i может начать выполняться с момента времени 0, перестановка осуществляется с любым заданием на первом процессоре

4) На следующей итерации, в качестве базового, берется задание j_2, и так далее, и вновь с j_1.

4.1.2. Мутации для потомков MT

1) Выбирается базовая вершина j_1, из множества потомков дерева MT. Осуществляется перестановка вершины на другой процессор, либо на

этот же процессор, но раньше в расписании. Соответственно задание x, стоящее на выбранном месте ставится на то место, на котором было j_1.

2) На следующей итерации, в качестве базового, берется задание j_2, и так далее, и вновь с j_1.

Результатом мутаций решения $\pi(i)$ — будет последовательное изменение позиции в расписании корневой вершины i и её предшественников.

5. Реализация

Алгоритм **IASC** был реализован на языке *MATLAB*.
В качестве исходных данных, были взяты условия из банка *Optimal Schedules for Prototype Standard Task Graph Set* [3] — где для большинства задач оптимальное решение известно.
Входные данные для программы соответствуют формату **STG** (Standard Task Graph Set):
- 1-я строка: n
- i-я строка: i t_i $h(i)$ $pr(i)$, где:
i - номер задания
t_i - время выполнения
$h(i)$ - количество предшествующих заданий
$pr(i)$ - список предшествующих заданий

5.1. Алгоритм IASC — псевдокод

Algorithm 1 Main function

```
1: function MAIN(Graph : G)                          ▷ Генерируем начальное решение
2:     S:=initialSolution(G)                                  ▷ Создаём популяцию
3:     population := copy(S,n)
4:     iteration = 0
5:     while !stop do
6:         iteration++
7:         for i = 1:n do
8:             currentTime := time(population(i))
9:             rootMutate := mutateRoot(population(i))
10:            rootTime := time(rootMutate)
11:            if rootTime == Inf then                  ▷ Недопустимое решение
12:                childMutate := mutateChild(population(i))
13:            else
14:                childMutate := mutateChild(rootMutate)
15:            end if
16:            childTime := time(childMutate)           ▷ Обновляем популяцию
17:            if childTime <= currentTime then
18:                population(i) := childMutate
19:            else
20:                if rootTime <= currentTime then
21:                    population(i) := rootMutate
22:                end if
23:            end if                                   ▷ Обновляем индексы мутаций
24:            population(i).rootIndex = rootMutate.rootIndex
25:            population(i).childIndex = childMutate.childIndex
26:        end for
27:        stop = (iteration == 10000)                  ▷ Критерий остановки
28:    end while
29: end function
```

Algorithm 2 Schedule time function

```
    function TIME(Schedule : A)
    ▷ Вектор оставшихся времён выполнения заданий
 2:     elapsedTimes := [t_1 ... t_n]
        currentTime := 0
 4:     stop := false
        while !stop do
 6:         readyTasks = []              ▷ Находим задания готовые к выполнению
            readyIndexes = []
 8:         minTime := Inf
            for i = 1:m do
10:             task = A(i,1)
                ready := true
12:             for j = 1:length(G(endEdges,i)) do
                    if ElapsedTimes(endEdges(i,j) != 0) then
14:                     ready := false
                    end if
16:             end for
                if ready == true then
18:                 readyTasks.add(task)
                    readyIndexes.add(i)
20:                 minTime := min(elapsedTimes(task),minTime)
                end if
22:         end for
            for j = 1:length(readyTasks) do
24:             elapsedTimes(j) -= minTime
                if elapsedTimes(j) == 0 then
26:                 A(readyIndexes(j),1).removeFromArray
                end if
28:         end for
            currentTime += minTime
30:         stop == elapsedTimes.equal(zeros(1,n))
            if (readyTasks == [] && !(stop)) then      ▷ Недопустимое решение
32:             return Inf
            end if
34:     end while
        return currentTime
36: end function
```

6. Результаты

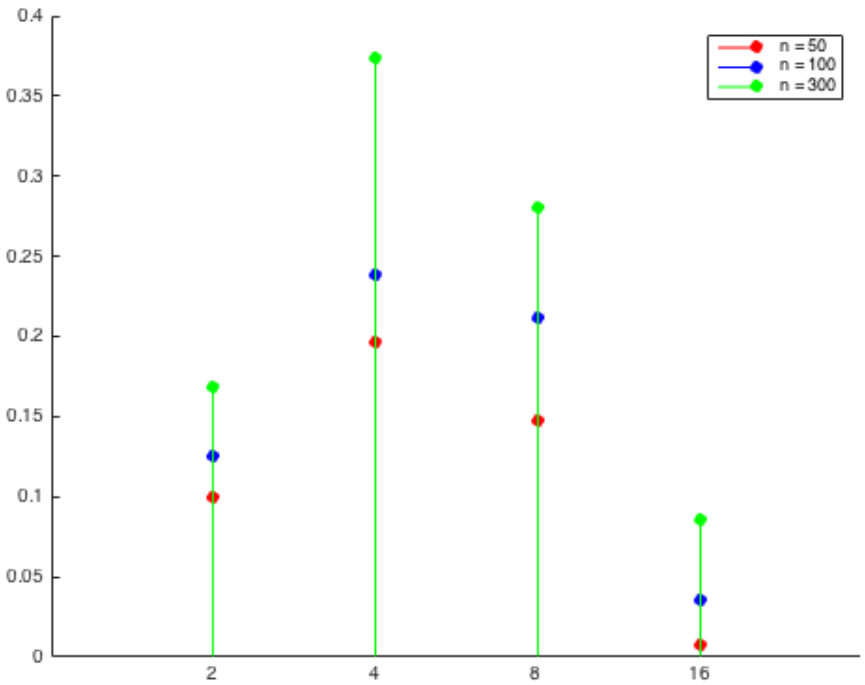
Алгоритм был запущен на пакетах задач из [3] с фиксированным числом заданий $(50, 100, 300)$ и процессоров $(2, 4, 8, 16)$. Общее число задач в каждом пакете — 180. Критерий остановки алгоритма — достижение оптимального результата, или достижение 1000-ой итерации. Ошибка алгоритма вычислялась по следующей формуле: $\frac{(f_a - f_{opt})}{f_{opt}}$, где f_a — решение полученное алгоритмом, f_{opt} — оптимальное решение. Были получены следующие результаты:

N — число заданий
M — число процессоров
MeanItearation — среднее число итераций
MeanTime — среднее время выполнения (в секундах)
MeanError — средняя ошибка
Optimals — процент заданий, в которых оптимальное решение было найдено.

N	**M**	**MeanIteration**	**MeanTime**	**MeanError**	**Optimals**
50	2	804	20	0.1001	10
50	4	910	25	0.197	8
50	8	751	29	0.147	25
50	16	105	8	0.0081	90
100	2	920	44	0.125	5
100	4	970	65	0.239	3
100	8	917	85	0.212	8
100	16	440	95	0.035	56
300	2	981	120	0.169	2
300	4	1000	145	0.374	0
300	8	985	186	0.280	2
300	16	800	260	0.086	19

На представленном ниже графике показана зависимость величины ошибки от количества процессоров, для $n = 50, 100, 300$. Как видно, с ростом

числа заданий, растёт величина ошибки, однако начиная с 4-х процессоров, при фиксированном n, величина ошибки падает.



Для задач больших размерностей из [3] был выбран критерий остановки алгоритма — отсутствие улучшения решения на протяжении 100 итераций. Ошибка алгоритма вычислялась по той же формуле, что и в задачах малых размерностей. В общем случае, результаты работы алгоритма для $n > 500$ удовлетворительные. Ошибка на всех задачах не превышала 0.64. Аналогично задачам малых размерностей, величина ошибки была прямо пропорциональна числу заданий, и обратно пропорциональная числу процессоров ($m > 4$).

Заключение

Алгоритм **IASC** был запущен на более чем 2-х тысячах задач. Лучшие результаты были получены на задачах с большим числом процес-

соров и малым числом заданий: с ростом числа заданий растёт ошибка и число итераций, необходимые для достижения оптимального результата, либо определённой планки погрешности вычислений, а с ростом числа процессоров, несмотря на более трудоёмкие вычисления, алгоритм стабильно выдавал лучшие результаты.

Зачастую, обнаружение локального минимума в решении из популяции закрывало все возможности для улучшения — логика мутаций в приведённом алгоритме может сойтись на случай, когда каждую итерацию будут производиться модификации, не приводящие к улучшению результата, и, как следствие, глобальный минимум не будет достигнут.

Простота реализации, универсальность и гибкость иммунных сетей позволяет использовать их для решения большого количества *NP-трудных* задач. Представленный автором алгоритм **IASC** является иммунной сетью всего с двумя мутациями, однако погрешности алгоритма являются невысокими. Дальнейшее расширение сети позволит добиться ещё более точных результатов.

Список литературы

[1] *В. С. Блюм и В. П. Заболотский* Иммунная система и иммунокомпьютинг // Санкт-Петербургский институт информатики и автоматизации РАН стр 2-10
www.smolensk.ru/user/sgma/MMORPH/N-16-html/blum/blum.pdf

[2] *О. А. Щербина* Метаэвристические алгоритмы для задач комбинаторной оптимизации // Таврический вестник информатики и математики 1(24) 2014 г. стр 3-5
tvim.info/files/56_72_Shcherbina.pdf

[3] *Kasahara Lab Waseda University* Optimal Schedules for Prototype Standard Task Graph Set
www.kasahara.elec.waseda.ac.jp/schedule/index.html

[4] *Лазарев А. А.и Гафаров Е.Р.* Теория расписаний задачи и алгоритмы // МГУ им. Ломоносова 2011 г. стр 13-25
physcontrol.phys.msu.ru/materials/PosobieLazarev/TeorRasp.pdf

[5] *Алексеев Г.А.* Иммунный алгоритм для задач составления расписаний // СПбГУ кафедра Исследования Операций 2013 г. стр 2-10

[6] *Григорьева Н.С.* Теория расписаний методические указания // СПбГУ кафедра Исследования Операций 1995 г. стр 3-5