Санкт-Петербургский
Государственный Университет

Математико-механический факультет
Кафедра Системного Программирования

Методы комбинаторной оптимизации в задачах расположения сервисов в дата-центрах

Дипломная работа студента 545 группы
Шалупова Леонида Борисовича

Научный руководитель: .... ст.преп. Симуни М. Л.

____
/подпись/

Рецензент: .... к.т.н. Королёв Ю. В.

____
/подпись/

„Допустить к защите“
Заведующий кафедрой: .... д.ф.м.н., профессор Терехов А. Н.

____
/подпись/

Санкт-Петербург
2008 г.

Аннотация

В настоящей работе описана задача оптимального расположения связанных сетевых компонент (сервисов) на физических компьютерах (серверах), приведена её математическая модель, исследованы существующие методы комбинаторной оптимизации в приложении к данной задаче

Содержание

1 Введение **3**
- 1.1 Предметная область и цели настоящей работы 3
- 1.2 Текущее состояние изучаемой области 4

2 Математическая модель **4**
- 2.1 Основные понятия 4
- 2.2 Задача квадратичного программирования 6
- 2.3 Расширения модели 7

3 Связь с известными задачами комбинаторной оптимизации **7**
- 3.1 Mixed-Integer Quadratic Problem (MIQP) 7
- 3.2 Квадратичная транспортная задача 8

4 Алгоритмы решения **8**
- 4.1 Метод ветвей и границ 8
- 4.2 Жадный поиск 9
- 4.3 Жадный поиск с историей 10
- 4.4 Табу-поиск 11
- 4.5 Генетические алгоритмы 12
- 4.6 Гибридный генетический алгоритм 13
- 4.7 Ant colony optimization 15

5 Реализация **18**

6 Методика тестирования и сравнение производительности **18**
- 6.1 Генерация тестовых задач 18
- 6.2 Методика тестирования 19
  - 6.2.1 Выбор задач 19
  - 6.2.2 Прогонка тестов 19
- 6.3 Результаты 20
- 6.4 Сравнение результатов: выводы 21

7 Заключение **23**

1 Введение

1.1 Предметная область и цели настоящей работы

В компании, занимающейся разработкой системы для обслуживания call-центров, возникла следующая задача: автоматизировать расположение связанных сетевых сервисов на физических серверах, расположенных как в одном месте, так и раскиданных по всему миру.

Опишем подробнее эту задачу:

- Call-центр состоит из логически связанных и сильно разнесённых площадок (локаций)
- Каждая такая локация состоит из группы серверов
- На каждом сервере работает один или несколько сервисов
- Требуется придумать алгоритм, размещающий эти сервисы на серверах по некоторым критериям оптимальным образом

Перечислим некоторые возможные типы сервисов:

- Связь с телефонным миром (интерфейсы к специализированному оборудованию);
- Интерфейсы к базе данных;
- Решения по обработке поступающих событий и принятия решений;
- Статистика;
- Мониторинг и журнал событий.

В реальной жизни характерны такие размеры задачи: несколько локаций, десятки серверов на каждой, сотни работающих сервисов.

Сейчас развертывание всей конфигурации происходит в полуавтоматическом режиме: поддерживается только автоматическая установка указанного сервиса на сервер. Автоматическая настройка связей между сервисами и раскладка сервисов по серверам не подерживается.

Цель дипломной работы – автоматизировать такую раскладку, предложив более-менее «разумный» вариант, учитывая качество связи между серверами и площадками, ресурсы серверов, нагрузки сервисов, связи между ними.

Теперь кратко опишем задачу: есть набор сервисов (указана нагрузка на сервер), связи между ними (указано требование к качеству такой связи), набор серверов, соединенных между собой (дано качество канала).

Разработать алгоритм, раскладывающий оптимальным образом сервисы по серверам учитывая каналы, ресурсы серверов, связи между сервисами.

1.2 Текущее состояние изучаемой области

Описанная задача относится к задачам комбинаторной оптимизации, которые исследуются примерно с 18 века (хороший обзор истории до 1960 года можно найти в [1], а методы наших дней изложены в [2]).

Примеры задач комбинаторной оптимизации: задача коммивояжера, раскраски графа, траспортная. Большинство таких задач – NP-трудные, что вынуждает придумывать сложные алгоритмы для получения точного или хотя бы приблизительного решения.

Все методы решения задач комбинаторной оптимизации делятся на две группы: точные (exact) и эвристические.

Выбор методов зависит от конкретной постановки задачи, где определяющим критерием является время работы. Эвристики могут очень бысто дать решение, близкое к оптимальному, однако про его качество ничего сказать нельзя. Точные методы всегда находят глобальный экстремум, но время работы может быть гораздо дольше.

В нашей задаче поиск точного решения не является приоритетным, поэтому будем рассматривать только эвристические методы.

2 Математическая модель

2.1 Основные понятия

Рассмотрим математическую модель решаемой задачи.

Построим взвешенный связный граф серверов F. Вершины графа – сервера, веса вершин – доступные ресурсы сервера. В данной постановке в качестве ресурсов берутся неотрицательные числа. Дуги графа будут показывать соединения между серверами. Вес дуги характеризует качество канала (в настоящей работе - *латентность*, меньшие значения соответствуют лучшему качеству канала).

Некоторые обозначения:

- $F.V$ – множество серверов (вершины),
- $F.E$ – связи между ними (дуги),
- $|F.V| = n$,

- f_i – ресурсы i-го сервера,

- ω_{ij} - длина кратчайшего пути между i и j-м сервером. Такой путь всегда существует, так как в нашей модели граф серверов связный.

Построим взвешенный граф сервисов G. Вершины графа – сервисы, вес вершины – необходимые ресурсы для этого сервиса. Дуги графа – зависимости между ними, вес дуги – необходимое качество связи между сервисами. Грубо: чем больше это значение, тем ближе должны находиться сервисы.

Обозначения:

- $G.V$ – множество сервисов (вершины),

- $G.E$ – множество связей между ними (дуги),

- $|G.V| = m$,

- g_i – необходимые ресурсы i-го сервиса,

- d_{ij} – вес дуги между i и j-м сервисом. Для $i = j$ и несоединённых напрямую сервисов d_{ij} равно 0.

Опишем расположение сервисов на серверах. Назовём состоянием вектор $s \in [1..n]^m$, в котором по индексу сервиса находится номер сервера, на котором он расположен, и удовлетворяющий ограничению по ресурсам серверов:

$$\forall i \in 1..n \sum_{\substack{j \in 1..m \\ s_j = i}} g_j \leq f_i$$

Множество всех возможных состояний обозначим S. Можно сформулировать оценочную функцию $q\colon S \to \Re$ нашей задачи:

$$q(s) = \sum_{(i,j) \in G.E} \omega_{s_i s_j} d_{ij}$$

Смысл этой функции в вычислении суммы всех связей между сервисами, которые расположены на разных серверах. Если сервисы расположены на одном сервере ($s_i = s_j$) или не связаны между собой ($d_{ij} = 0$), то их вклад в оценочную функцию будет нулевым.

Тогда исходную задачу можно переформулировать так:

$$\min_{s \in S} q(s)$$

2.2 Задача квадратичного программирования

Сначала дадим определение задачи квадратичного программирования:

$$\min_{x \in X} f(x) = \frac{1}{2} x^T Q x + c^T x$$
$$\begin{cases} A_i x \le b_i \\ E_j x = d_j \end{cases}$$

где $c \in X$; $\deg X = q$; $Q \in M^{q \times q}$; $A_i \in M^{k \times q}, b_i \in M^k$; $E_j \in M^{p \times q}$ соотвественно.

Предметом перебора такой задачи являются вектора. Но, к сожалению, с помощью вектора состояния напрямую не получается записать $f(x)$. Поэтому на основе состояния s построим вектор $x \in \{0,1\}^{n \cdot m}$:

$$x_k = \begin{cases} 1, s_j = i, k = i \cdot m + j; \\ 0, \text{иначе.} \end{cases}$$

Перепишем ограничения на вектор состояния:

$$\forall j \sum_i x_{i \cdot m + j} = 1 \quad \text{(один и только один сервер на каждый сервис } j\text{)}$$
$$\forall i \sum_j g_j x_{i \cdot m + j} \le f_i \quad \text{(ресурсов сервера } i \text{ хватает для размещения сервисов)}$$

Тогда оценочную функцию можно переписать:

$$f(x) = \sum_{\substack{i,j,k,l \\ i \ne k \\ j \ne l}} x_{i \cdot m + j} x_{k \cdot m + l} w_{ik} d_{jl}$$

Учитывая, что $\forall i \; w_{ii} = 0$ и $\forall j \; d_{jj} = 0$ функцию $f(x)$ можно упростить:

$$f(x) = \sum_{i,j,k,l} x_{i \cdot m + j} x_{k \cdot m + l} w_{ik} d_{jl}$$

Это напоминает $x^T Q x$. Построим $Q \in M^{nm \times nm}$:

$$Q[i \cdot m + j, k \cdot m + l] = w_{ik} d_{jl} \;\; \forall i, k \in 1..n; j, l \in 1..m$$

Тогда $f(x) \equiv x^T Q x$.

Задача переписывается:

$$\begin{gathered}\min_{x \in \{0,1\}^{nm}} x^T Q x \\ \forall j \sum_i x_{i \cdot m+j} = 1 \\ \forall i \sum_j g_j x_{i \cdot m+j} \leq f_i\end{gathered} \tag{1}$$

Ограничения же переписываются в $n+m$ линейных условий.

Таким образом, задача приведена к виду бинарной квадратичной задачи (*binary quadratic programming*). Для этих задач существуют методы оценки нижней границы [3].

2.3 Расширения модели

В реальной жизни ресурсы не выражаются одним числом, а представляют собой вектора. Отдельные компоненты указывают на требования к процессорам, памяти, дисковой активности и т.д. Наша модель обобщается для этого случая за счёт превращения векторов f и g в матрицы и тривиального переписывания ограничений на ресурсы сервера.

Кроме этого оптимизировать можно по нескольким целевым функциям (multi-objective optimization), например, по $f(x)$ и средней загрузке сервера. Рассмотрение таких задач выходит за рамки настоящей работы, однако алгоритмы, используещиеся здесь легко переносятся на несколько целевых функций. Более подробно это описано в [2].

3 Связь с известными задачами комбинаторной оптимизации

3.1 Mixed-Integer Quadratic Problem (MIQP)

Mixed-Integer Quadratic Problem (MIQP) называется такая задача:

$$\begin{aligned}&\min_x f(x) = \frac{1}{2} x^T G x + g^T x \\ &A^T x \geq b \\ &x \in X, x_i - \text{integer } \forall i \in I\end{aligned}$$

В предыдущей главе исходная задача была сведена к бинарной квадратичной, которая является частным случаем MIQP. На рынке существует множество математических пакетов, точно решающих MIQP. Однако

на текущий момент все они требуют положительной определённости G, чего мы не можем гарантировать.

Такое требование связано с тем, что эти пакеты используют метод ветвей и границ [4], и, как следствие, используют непрерывную задачу квадратичного программирования, которая быстро решается только при положительной определённости Q.

3.2 Квадратичная транспортная задача

Аналогом нашей задачи является квадратичная транспортная задача (quadratic assignment problem – **QAP**).

Её постановка: дано n заводов и n мест их расположения. Для каждой пары мест известно расстояние, для каждой пары заводов известен поток (объём ресурсов, траспортируемое между заводами). Выбрать такое расположение заводов, при котором сумма произведений потоков и соотвествующих расстояний будет минимальной. [5]

QAP используется для планирования университетов, заводов, больниц, разводке печатных плат и т.п.

QAP – частный случай нашей задачи при $n = m$, $\forall i\; f_i = 1$, $\forall i\; v_i = 1$. Известно, что QAP – NP-трудная, таким образом, мы рассматриваем тоже NP-трудную задачу.

4 Алгоритмы решения

Разумной идеей кажется взять алгоритмы, которые хорошо себя показали на QAP, реализовать и сравнить их для нашей задачи. Здесь мы рассмотрим лишь подмножество возможных вариантов. Более-менее полный список алгоритмов можно найти в [2].

Также невозможно описать все варианты конкретного алгоритма. Поэтому мы остановимся на конкретной реализации алгоритмов для нашей задачи и дадим ссылки на более подробное описание.

4.1 Метод ветвей и границ

> Метод ветвей и границ — общий алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. По существу, метод является комбинаторным (алгоритм перебора) с отсевом подмножеств множества допустимых решений, не содержащих оптимальных решений. Метод был впер-

> вые предложен Ленд и Дойг в 1960 г. для решения задач линейного программирования. [4]

Для отсева решений метод использует какой-нибудь способ оценки границ. В частности, для квадратичной задачи обычно используется соответствующая непрерывная задача. В нашем случае мы не можем гарантировать положительной определённости Q, следовательно, не можем эффективно посчитать непрерывную задачу.

Существуют и другие методы оценки границ, но в задачах оптимизации из-за скорости свой работы они скорее используются для примерного нахождения значения оптимального решения, нежели для метода ветвей и границ. Список алгоритмов для оценки QAP можно найти в описании на QAPLIB [6]. Для наших примеров оценка нижней границы не актуальна из-за очень большой размерности задач. Поэтому для нашей задачи использовать метод ветвей и границ неэффективно.

4.2 Жадный поиск

> Жадный алгоритм – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская что конечное решение также окажется оптимальным. [7]

Принятие локально оптимальных решений на каждом этапе приводит к нахождению только локального экстремума, поэтому мы не можем прямо использовать жадный поиск.

Однако различные модификации жадного поиска способны выходить из локального экстремума и искать следующий. Также жадный поиск активно используется для улучшения каких-либо решений, полученных, например, случайным образом.

Прежде чем говорить о локальном поиске, нужно определить из чего мы будем выбирать локально оптимальные решения, т.е. определить *соседей* состояния.

О соседних состояниях

Здесь и далее будут использоваться обозначения: $state$ – текущий рассматриваемый вектор, состояние поиска, $N(state)$ – соседи текущего вектора.

Для определения понятия «соседи» введём сначала понятие расстояния между состояниями:

$$D(\pi^1, \pi^2) = |\{i \in 1..n | \pi^1_i \neq \pi^2_i\}|$$

Соответственно,

$$N_2(state) = \{x \in S | 0 < D(x, state) \leq 2\}$$

т.е. соседними будут состояния, которые отличаются одним или двумя элементами. Это согласуется с задачей QAP, в которой соседи отличаются на одну перестановку пары элементов.

Однако в этом случае размер такого множества будет довольно большим. Поэтому было решено использовать его подмножество:

$$N(state) = \{swap(state, i, j) | \forall i, j \; i \neq j\} \cup \{x \in S | D(x, state) = 1\} \quad (2)$$

т.е. перестановки пар элементов и одиночные замены.

Опишем алгоритм жадного поиска (Algorithm 1).

Algorithm 1 greedySearch(*state*)

Require: Вычислить локальный минимум около состояния *state*

repeat
  $best := state$
  $state := selectBestState(N(best))$
until $f(state) >= f(best)$
return *best*

4.3 Жадный поиск с историей

Простейший способ выходить из локального минимума:

- разрешить заходить в состояния с большим значением, чем лучшее
- запоминать уже просмотренные состояния и не заходить в них ещё раз

Тогда поиск будет исследовать окрестности минимума и выходить за его пределы.

Как и следовало ожидать, такой алгоритм плохо работает в тех задачах, в которых есть плотная концентрации локальных минимумов или множество одинаковых значений вокруг минимума велико. В этих случаях он обходит все соседние элементы с одинаковым значением. Тем не менее, из-за простоты реализации жадный поиск с историей будет участвовать в сравнении алгоритмов.

Algorithm 2 greedyWithHistory()

```
state = generateFeasibleState()
best = state
seen = {best}
repeat
  state := selectBestState(N(state)\seen)
  seen := seen ∪ {state}
  if f(state) < best then
    best := state
until |N(state)\seen| = 0
return best
```

4.4 Табу-поиск

Табу-поиск [8] – другая разновидность жадного поиска, которая умеет выходить из локальных минимумов. Ключевая идея – запретить назначение некоторых серверов на конкретные сервера. Список таких запретов называется «табу-список» и содержит номер итерации, до которой нельзя назначать конкретный сервис на конкретный сервер.

Алгоритм имеет множество модификаций (RoTS [9], ReTS [10], ITS [11]), полный список доступен на QAPLIB [6].

Robust Tabu Search

RoTS отличается от стандартного табу-поиска «переменной длиной списка» запретов, что выражается в переменном количестве итераций, на которое запрещается данная позиция. Т.е. табу-поиск отличается изменённым по сравнению со стандартным (2) множеством соседей состояния:

$$N^{TS}_{iteration}(state) = N(state) \setminus \{s \in S | \forall i \in 1..m \ L_{is_i} < iteration\}$$

где $iteration$ – текущая итерация, L_{ij} – номер итерации, до которой запретить ставить сервис i на сервер j.

Проще всего работу этого поиска показать на псевдокоде (Algorithm 3).

Iterated Tabu Search

Как следует из названия, ITS [11] – итеративный алгоритм, где каждый шаг состоит из трех операций:

Algorithm 3 robustTabuSearch($state$, S_{min}, S_{max})

$best := state$
$iteration := 0$
repeat
  {Найти лучшее решение среди соседей $state$}
  $state, move := selectBestState(N^{TS}_{iteration}(state))$
  if $cost(best) > cost(state)$ **then**
    $best := state$
  {Запретить делать ход $move$ на какое-то количество итераций}
  forbid move $move$ until $iteration + random(S_{min}, S_{max})$
  $iteration := iteration + 1$
until terminating
return $best$

selection Выбирается, с каким состоянием мы будем работать на данном шаге. Обычно это результат предыдущего шага или наилучший результат на текущий момент.

diversification(*distance*) Состояние случайным образом меняется (мутирует) так, чтобы расстояние до исходного составило *distance*.

intensification(*iterations*) на полученный результат мутации запускается табу-поиск на *iterations* итераций

Для реализации алгоритма нужно задать стратегию выбора, механизм изменения параметра *distance* у мутации, количество итераций *iterations*. Различные варианты описаны в оригинальной работе [11]. Данная реализация придерживается следующих решений:

- На каждом шаге обрабатывается результат предыдущего
- *distance* меняется в заданном интервале. На каждом шаге *distance* уменьшается, при выходе за допустимые пределы или стагнации значений, *distance* увеличивается.
- *iterations* постоянно и является параметром алгоритма

4.5 Генетические алгоритмы

Генетический алгоритм – это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путем последовательного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию. [12]

В Г.А. используются четыре основные операции:

generate_population(N): Создать какой-нибудь набор состояний определённого размера. Для Г.А. качество этих состояний не важно, он сам будет отбирать из них лучшие.

recombine(s1, s2): Имея два состояния, создать третье, «включающее свойства» первых двух. Например, брать поочерёдно элементы из первого и второго состояний.

mutate(s): Случайным образом чуть-чуть поменять состояние, *мутировать* его.

select_population(P, N): Выбрать из популяции P N лучших состояний.

Algorithm 4 customGenetic(Np, Nr, Nm)

{Получить какой-нибудь набор состояний размера Np}
$P := generatePopulation(Np)$
repeat
  {Скрестить случайные состояния Nr раз}
  for $k = 1$ to Nr **do**
    $s1 := randomChoice(P)$
    $s2 := randomChoice(P)$
    $P := P \cup \{recombine(s1, s2)\}$
  {Мутировать Nm случайных состояний}
  for $k := 1$ to Nm **do**
    $s := randomChoice(P)$
    $P := P \cup \{mutate(s)\}$
  {Отобрать Np лучших состояний}
  $P := selectPopulation(P, Np)$
until *terminate*
return $findBest(P)$

Мы не будем исследовать генетический алгоритм прямо в таком виде, а рассмотрим его модификацию, которая показала лучшее результаты для решения QAP.

4.6 Гибридный генетический алгоритм

В этом разделе мы рассмотрим одну из модификаций генетического алгоритма – *memetic algorithm.*

Memetic algorithm – модификация генетического алгоритма, которая вместо обычных состояний держит в популяции локальные минимумы, проделывая скрещивания и мутации именно с ними.

Описание MA для QAP и сравнение с другими эвристическими методами приводится в [13]. Полное описание MA и приложение его для других задач описано в [14].

Algorithm 5 memetic(Np, Nr)

```
{Получить какой-нибудь набор состояний размера Np}
P := generatePopulation(Np)
P := {localSearch(x)|x ∈ P}
repeat
  {Скрестить случайные состояния Nr раз}
  for k := 1 to Nr do
    s1 := randomChoice(P)
    s2 := randomChoice(P)
    P := P ∪ {narrowLocalSearch(recombine(s1, s2), s1, s2)}
  {Отобрать Np лучших состояний}
  P := selectPopulation(P, Np)
  {Если элементы популяции слишком похожи – мутировать их}
  if converged(P) then
    best := find_best(P)
    P = {best} ∪ {localSearch(mutate(x))|x ∈ P}
until terminate
return find_best(P)
```

Опыт использования алгоритма показал, что качество его работы сильно зависит от входных параметров (размера популяции, количества рекомбинаций, величины мутаций).

Рассмотрим подробнее вспомогательные функции:

Рекомбинация

Рекомбинация – функция, генерирующая из двух состояний одно, представляющее собой нечто среднее между ними. В memetic algorithm рекомбинация позволяет искать локальные минимумы, расположенные между двумя данными.

Главная идея рекомбинации – заимствовать части обеих состояний, но не допустить при этом случайных мутаций.

Операция работает следующим образом:

- Если какой-то сервис в исходных состояних находится на одном и том же сервере, то и в потомке этот сервис будет находиться там же.
- Иначе расположение сервиса случайно выбирается из серверов исходных состояний (не нарушая условий (1), конечно).

Мутация

Мутация – функция, заменяющая некоторое количество элементов случайными значениями. На практике расстояние между сервисами до и после мутации постоянно и является параметром алгоритма.

Узкий локальный поиск

После процесса рекомбинации мы не проводим локальный поиск по всему множеству S, а сужаем количество вариантов так, чтобы в результате оставались на своих местах те сервисы, которые попали на один и тот же сервер в аргументах рекомбинации.

4.7 Ant colony optimization

Муравьиная оптимизация была разработана по результатам изучения муравьиной колонии. Рассмотрим механизм общения муравьёв. Сначала они случайным образом исследуют пространство вокруг муравейника в поисках пищи. После её нахождения – возвращаются в муравейник, помечая свой путь феромонами [15]. Следующий муравей умеет обнаружить такой след и последовать ему. Чем больше феромонов скопилось на каком-то пути, тем больше вероятность следования муравьёв по нему. Однако пища когда-то заканчивается, следовательно, муравьи должны перестать ходить по следу. Это достигается тем, что феромоны со временем испаряются и след исчезает, что заставляет муравьёв искать другие пути.

В 1992 году Марко Дориго (Marco Dorigo) [16] предложил использовать такой подход для решения задач комбинаторной оптимизации. С тех пор было предложено множество модификаций метода [17]. В текущей работе подробно будет рассмотрен только один из них – MAX-MIN Ant System (MMAS) [18], который на данный момент показывает наилучший результат для QAP.

Описание алгоритма

Как оказалось, идея природы довольно хорошо перекладывается на математический язык. Итак, разделим решение из пространства решений на составные части, которые назовём компонентами (solution components). Каждой компоненте соответствует какой-то уровень феромонов (просто число). «Муравьи» будут конструировать конкретное решение, выбирая между конкретными компонентами, учитывая уровень феромонов. Если решение окажется удачным, то уровень соответствующей компоненты будет увеличен. Чтобы поиск в какой-то момент не остановился, перед обновлением уровня феромонов все они будут уменьшены в какое-то количество раз.

Модификация MAX-MIN имеет три особенности:

- В обновлении феромонов учавствует только один муравей – лучший за текую итерацию

- Значение уровня феромонов ограничено ($[R_{min}, R_{max}]$), чтобы поиск не останавливался. Положительность R_{min} и ограничение сверху (R_{max}) гарантирует, что муравьи будут исследовать всё пространство решений, а не только лучшие варианты на текущий момент

- При старте алгоритма уровень установлен в R_{max}, что обеспечивает лучшее исследование пространства поиска на начальном этапе

Теперь опишем сам процесс. Алгоритм будет состоять из итераций, на каждой из которых несколько муравьёв будет генерировать набор решений согласно уровням феромонов, а потом лучшее решение будет обновлять эти уровни. Рассмотрим подробнее алгоритм в приложении к нашей задаче.

Компонентой решения будет являться выбор конкретными сервисом конкретного сервера (т.е. всего $m * n$ компонент). Обозначим соответствующий уровень феромонов через $\tau_{ij}(t)$, где i – сервис, j – хост, а t – текущая итерация («время»).

На каждой итерации t каждый муравей поочерёдно выбирает хосты для сервисов. Для сервиса i вероятность выбора сервера j из возможного множества серверов N_i, на которых достаточно ресурсов для сервиса i на данном шаге, составляет:

$$p_{ij}(t) = \frac{\tau_{ij}}{\sum_{k \in N_i} \tau_{ik}} \tag{3}$$

После выбора пути каждым муравьём, из них выбирается тот, чьё решение лучше, и уровень феромонов обновляется:

$$\tau_{ij}(t+1) := \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}^{best}(t)$$

где $\rho \in (0,1)$ моделирует испарение феромонов, а $\Delta\tau_{ij}^{best}$ – вклад лучшего за итерацию муравья:

$$\Delta\tau_{ij}^{best} = \begin{cases} 1, & \text{если } s_i = j \\ 0, & \text{иначе} \end{cases}$$

где вектор s – полученное муравьём решение.

Формализуем сказанное в псевдокоде (Algorithm 6).

Algorithm 6 MMAS(p, $NumAnts$, R_{min}, R_{max})

```
{Инициализировать уровень феромонов}
resetPheromoneLevelToRmax();
repeat
   {Сгенерировать набор решений}
   for k := 1 to NumAnts do
      for i := 1 to m do
         {Выбрать сервер по формуле 3}
         states_i(k) := selectHost()
   {Выбрать из них лучшее}
   best = selectBest(states)
   {Возможно, это решение лучше, чем все предыдущие}
   if cost(best) < cost(globalBest) then
      globalBest := best
   {Испарить в p раз}
   for i := 1 to m do
      for j := 1 to n do
         τ_ij = p · τ_ij
         if τ_ij < R_min then
            τ_ij = R_min
   {Прибавить уровни для лучшего решения}
   for i := 1 to m do
      {j – сервер для сервиса i}
      j := best_i
      τ_ij = τ_ij + 1
      if τ_ij > R_max then
         τ_ij = R_max
until terminate
return globalBest
```

Особенности реализации

В отличие от MMAS, R_{min} является параметром, R_{max} вычисляется, как функция от p. MMAS же использует адаптивный алгоритм для вычисления и корректирования границ [18].

5 Реализация

Перечисленные выше алгоритмы были реализованы на языке C++ с оберткой на Python.

Опишем ход работы. Сначала с помощью генератора задач (Python) получается описание обеих графов в текстовом виде. Потом описание обрабатывается скриптом на Python, который рисует графическое представление графов и генерирует вход для решающей программы, которая принимает графы в виде матриц смежности. Запускается решатель. Результатом его работы является вектор состояния, который после этого может быть отображён графически с пощью скрипта на Python.

6 Методика тестирования и сравнение производительности

6.1 Генерация тестовых задач

Для тестирования алгоритмов был придуман сравнительно простой генератор. Граф серверов эмулирует сервера, подключённые к одному коммутатору. Граф сервисов строится следующим образом:

- Генерируется граф – цепочка из вершин
- Дуги графа случайным образом перемещаются, сохраняя связность графа
- В граф добавляются случайным образом дуги
- Дуги снова перемещаются

Таким нехитрым образом при должном подборе количества каждой из операций получаются графы вида изображенного на рисунке 6.1.

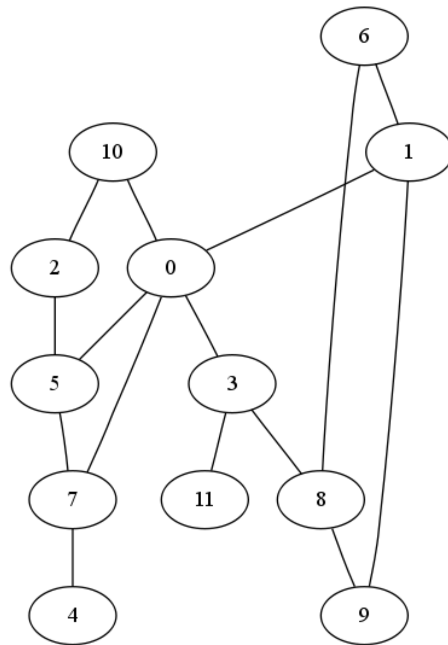

Рис. 1: Пример сгенерированного графа сервисов

6.2 Методика тестирования

6.2.1 Выбор задач

Подборку задач можно разделить на две части: QAP и сгенерированные тестовые задачи. Так как QAP является частным случаем нашей задачи, то можно использовать существующие задачи из QAPLIB [6].

Для QAP выбирались задачи больших размерностей (>30), которые отражали задачи реального мира или были сгенерированы подобным образом. Под размерностью QAP имеется ввиду число расположений.

Тестовые задачи генерировались исходя из размерностей промышленных задач. Подробнее об этом было написано во введении.

6.2.2 Прогонка тестов

Для каждого алгоритма и каждой задачи решающая программа запускалась несколько раз.

Тесты проводились на многоядерном процессоре, и приоритет процесса был выставлен на максимум. В такой конфигурации процесс занимал одно ядро полностью.

Каждый запуск был ограничен по времени. Временной интервал выбирался из предположения, что начальные подсчёты хочется делать ин-

терактивно, следовательно, время дожно быть небольшим. После ряда экспериментов было установлено, что в большинстве случаем запуск дольше 200 секунд не приводит к улучшению результата. Таким образом, измерения проводились запуская каждый алгоритм по 200 секунд

Результатом для последующего сравнения являлись две величины:

Лучший результат за время работы. Это лучшее значение стоимости состояний за все запуски решающей программы для определенной задачи.

Среднее значение текущего лучшего результата. При работе решающей программы через равные промежутки времени запоминался текущий лучший результат. Потом по этим данным бралось среднее, которое тоже усреднялось для нескольких запусков. Полученная величина оценивает «скорость» работы алгоритма.

6.3 Результаты

QAP

Результаты исполнения алгоритмов на задачах QAP приведены в таблицах 1 и 2 (**выделены** лучшие результаты).

Таблица 1: Среднее значение стоимости лучшего текущего решения для Quadratic Assignment Problems

Задача	ITS	Memetic	MMAS	Forbidden states	RoTS
sko42	15815.06	15821.00	15853.16	15829.35	**15813.20**
sko90	**115883.59**	116275.69	116527.06	116669.85	115919.13
ste36a	9534.16	**9529.92**	9558.60	9728.65	9580.37
ste36b	**15856.40**	15917.96	15954.12	16856.55	16406.37

Таблица 2: Лучшая стоимость найденного решения для Quadratic Assignment Problems

Задача	ITS	Memetic	MMAS	Forbidden states	RoTS
sko42	**15812**	**15812**	15816	**15812**	**15812**
sko90	**115602**	116062	116380	116336	115726
ste36a	**9526**	**9526**	**9526**	**9526**	**9526**
ste36b	**15852**	**15852**	**15852**	16480	**15852**

Сгенерированные задачи

Результаты исполнения алгоритмов на сгенерированных задачах приведены в таблицах 6.3 и 4. Число в названии задачи показывает число вершин в графе сервисов.

Memetic algorithm за 200 секунд не успел построить начальную популяцию для задачи ls322.

Таблица 3: Среднее значение стоимости лучшего текущего решения для сгенерированных задач

Задача	ITS	Memetic	MMAS	Forbidden states	RoTS
ls48a	**236.01**	**236.00**	**236.00**	**236.00**	260.00
ls48b	196.09	**196.03**	196.04	215.63	222.67
ls99	**388.48**	390.20	391.42	412.68	408.69
ls148a	500.63	**497.70**	503.05	567.87	553.47
ls148b	645.89	**574.61**	619.91	781.45	765.13
ls219	938.63	915.54	**913.62**	1003.42	969.78
ls322	1432.15	N/A	**1241.33**	1497.11	1448.86

Таблица 4: Лучшее значение стоимости найденного решения для сгенерированных задач

Задача	ITS	Memetic	MMAS	Forbidden states	RoTS
ls48a	**236**	**236**	**236**	**236**	252
ls48b	**196**	**196**	**196**	208	216
ls99	**384**	**384**	388	388	**384**
ls148a	484	**480**	**480**	548	528
ls148b	564	**542**	562	748	720
ls219	**880**	900	888	952	900
ls322	1188	N/A	1204	1304	**1160**

6.4 Сравнение результатов: выводы

Лучше всего себя показали ITS и Memetic – итеративные алгоритмы с мутацией решений.

Memetic легче распараллелить по операциям локального поиска. Его недостатки состоят в том, что с увеличением размерности время инициализации возрастает, соответственно, его не всегда можно будет применять в интерактивном интерфейсе. Кроме этого memetic принимает на

вход большое число параметров, которые требуют точной настройки для получения хорошего результата.

ITS – отличается простотой реализации и настройки. Он не требует времени на инициализацию и какой-то результат выдаёт сразу, что позволяет его использовать для быстрых начальных подсчётов. Однако прямых методов его распараллелить, по всей видимости, не существует.

7 Заключение

Результатом данной работы явилось создание математической модели исходной задачи, адаптация и эффективная реализация алгоритмов комбинаторной оптимизации, сравнение их на автоматически сгенерированных тестах, похожих на задачи реального мира.

По итогам сравнения предложены два алгоритма, ITS и Memetic, для дальнейшего изучения и внедрения в промышленную систему Genesys [19].

Список литературы

[1] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Handbook of Discrete Optimization*, pages 1–68. Elsevier, Amsterdam, July 24 2005.

[2] Thomas Weise. Global optimization algorithms: Theory and application, 2008.

[3] R. Fletcher and S. Leyffer. Numerical experience with lower bounds for MIQP branch-and-bound. *SIAM Journal on Optimization*, 8:604–616, 1998.

[4] Branch and bound. http://en.wikipedia.org/wiki/branch_and_bound.

[5] P. PARDALOS, F. RENDL, and H. WOLKOWICZ. The quadratic assignment problem: a survey and recent developments. In P. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems (New Brunswick, NJ, 1993)*, pages 1–42. Amer. Math. Soc., Providence, RI, 1994.

[6] S.E. KARISCH R.E. BURKARD, E. ÇELA and F. RENDL. A quadratic assignment problem library — www.seas.upenn.edu/qaplib.

[7] Greedy algorithm. http://en.wikipedia.org/wiki/greedy_algorithm.

[8] D. de Werra A. Hertz, E. Taillard. A tutorial on tabu search. In *Proc. of Giornate di Lavoro AIRO'95 (Enterprise Systems: Management of Technological and Organizational Changes)*, pages 13–24, Italy, 1995.

[9] D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[10] R. Battiti and G. Tecchiolli. The reactive tabu search, 1994.

[11] Dalius Rubliauskas Alfonsas Misevicius, Antanas Lenkevicius. Iterated tabu search: an imporvement to standard tabu search. *INFORMATION TECHNOLOGY AND CONTROL*, 35, 2006.

[12] Genetic algorithm. http://en.wikipedia.org/wiki/genetic_algorithm.

[13] Peter Merz and Bernd Freisleben. A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao,

and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 2063–2070, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press.

[14] P. Merz. Memetic algorithms for combinatorial optimization problems: Fitness landscapes and effective search strategies, 2000.

[15] Pheromone. http://en.wikipedia.org/wiki/pheromone.

[16] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[17] M. Dorigo, G. Di Caro, and L. Gambardella. Ant algorithms for discrete optimization, 1998.

[18] T. Stutzle and H.H. Hoos. Max min ant system, 2000.

[19] Genesys telecommunications laboratories. http://www.genesyslab.com.