Министерство науки и высшего образования Российской Федерации
Национальный исследовательский университет «МИЭТ»

В.Д. Колдаев

Лабораторный практикум по курсу «Алгоритмы и структуры данных»

Часть 2

Утверждено редакционно-издательским советом университета

Москва 2019

УДК 004.42 (075.8)

Рецензент д. физ.-мат. наук, проф. *В.В. Уздовский*

Колдаев В.Д.

Лабораторный практикум по курсу «Алгоритмы и структуры данных»: учебно-метод. пособие. Часть 2. - М.: МИЭТ, 2019. - с.: ил.

Рассмотрен широкий круг алгоритмов обработки линейных и нелинейных структур данных. Приведены основные понятия и определения, технология работы и фрагменты программ. Содержит пять лабораторных работ, каждая из которых включает в себя варианты заданий и задачи для самостоятельного решения.

Для студентов всех направлений подготовки НИУ МИЭТ.

Предисловие

В последние годы программирование для персональных компьютеров вылилось в дисциплину, владение которой стало ключевым моментом, определяющим успех многих инженерных проектов, а сама она превратилась в объект научного исследования. Специалисты в области программирования утверждают, что программы поддаются точному анализу, если они основаны на строгих математических рассуждениях. При этом основное внимание программисты обычно уделяют построению и анализу программы, а выбор представления данных, как правило, удостаивается меньшего, явно второстепенного внимания.

Современная методология программирования предполагает, что оба аспекта программирования - запись алгоритма на языке программирования и выбор структур представления данных - заслуживают абсолютно одинакового внимания, т.е. решение о том, как представлять данные, невозможно принять без понимания того, какие алгоритмы будут к ним применяться, и наоборот, выбор алгоритма часто зависит от строения данных, к которым он применяется. Без знания структур данных и алгоритмов невозможно создать сколько-нибудь серьезный программный продукт.

Ценность настоящего лабораторного практикума состоит в том, что оно предназначено не столько для обучения технике программирования, сколько для обучения, если это возможно, «искусству» программирования.

Методы и методики теории алгоритмов (в основном разделов асимптотического и практического анализа) позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения данной задачи с учетом особенностей их применения;
- получение временны́х оценок решения сложных задач;
- анализ доступных алгоритмов и структур данных, а также выбор среди них наиболее эффективных;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время;
- разработка и совершенствование алгоритмов решения задач на основе практического анализа.

В качестве инструментального языка программирования, на котором приводятся примеры, выбран язык С++. Выбор этого языка обусловлен несколькими причинами:

1) изучается в рамках дистанционного образования и широко используется для создания программных модулей;

2) обеспечивает поддержку объектно-ориентированному подходу (ОПП), который в настоящее время является основным для создания различных проектов.

Часть 2 настоящего учебно-методического пособия содержит пять лабораторных работ.

Теоретическая часть лабораторных работ изложена кратко и носит справочный характер, ее цель – дать основу для решения практических задач. В каждом из ее подразделов приведены типовые задачи с подробными решениям. Задания для самостоятельной работы во многом покрывают контрольные и зачетные задачи. Предполагается, что каждая тема осваивается за несколько этапов. В конце каждой лабораторной работы содержатся контрольные вопросы и задачи для самостоятельного решения.

Отчет по лабораторной работе должен содержать:

1) конспект лабораторной работы;

2) демонстрационные примеры анализирующие наиболее типичные приемы алгоритмического решения поставленной задачи;

2) результаты вычислений для каждого шага алгоритма;

3) программы на языке С++;

4) результаты выполнения работы (таблицы, графики, диаграммы);

5) выводы по работе.

Контроль усвоения дисциплины проводится с применением различных форм текущего контроля: опросы, тестирование, контрольные и самостоятельные работы.

ЛАБОРАТОРНАЯ РАБОТА № 1

Эвристические алгоритмы для решения комбинаторных задач

***Цель работы*:** изучение эвристических алгоритмов и способов их разработки; сравнительная оценка использования эвристических и переборных алгоритмов для решения комбинаторных задач.

***Продолжительность работы*:** 2 часа.

Теоретические сведения

Методы перебора являются *точными* методами решения комбинаторных задач, т.е. дают гарантию отыскания оптимального плана задачи, если речь идет о задачах оптимизации. В то же время методы перебора и все их модификации имеют серьезный недостаток: время работы экспоненциально растет при увеличении размерности задачи, что обычно неприемлемо для решения практических задач.

Эффективными можно считать алгоритмы, имеющие не более чем полиномиальные оценки трудоемкости, так как подобные алгоритмы не могут сводиться к полному перебору планов из-за того, что количество планов растет экспоненциально. Других подходов, пригодных для всех задач комбинаторного характера, учитывающих специфику конкретных задач, не имеется.

К эффективным (т.е. полиномиальным с невысокой степенью полинома) алгоритмам решения относятся, например, отыскание кратчайшего пути между вершинами графа, построение остовного дерева минимального веса, максимизация потока в сети и др. Однако для большей части задач неизвестны эффективные алгоритмы точного решения. При этом приходится жертвовать точностью определения экстремума ради сокращения времени его вычисления. Вместо строгого доказательства того, что полученный план будет давать точный минимум или максимум целевой функции, приходится считать, что полученный план с

большой вероятностью либо оптимален, либо хотя бы близок к оптимальному.

Относительно приемлемой является ситуация, когда точно известна хотя бы степень возможной неоптимальности. Алгоритмы оптимизации, для которых имеются нетривиальные оценки возможного отклонения решения от оптимума, называются ***приближенными*** или субоптимальными.

Примером приближенного алгоритма является задача о составлении расписания работы параллельных процессоров, когда нужно упорядочить исходные задания по убыванию их длительности, а затем каждое следующее задание отправить на наименее загруженный процессор. При этом отношение времени выполнения набора задач, получаемого по этому алгоритму (T), к минимально возможному времени выполнения, которое можно найти в результате перебора (T_0), всегда удовлетворяет неравенству:

$$T/T_0 \le 4/3 - 1/3n.$$

Так при $n = 10$ из уравнения получается оценка: $T_0 \ge 10/13 \cdot T$.

Если, например, время выполнения алгоритма составило 260 с, то нет гарантии, что это наилучшее возможное распределение. Однако можно утверждать, что минимальное возможное время не может быть меньше 200 с.

Не во всех случаях можно подобным образом оценить погрешность: возможна ситуация, когда используемый алгоритм находит достаточно хорошие планы задачи, но нет гарантий, что они близки к оптимальным. Алгоритмы, основанные на нестрогих соображениях «здравого смысла» и не имеющие никаких гарантий близости к оптимальным, называются ***эвристическими алгоритмами*** или эвристиками.

Трудно найти универсальные рецепты построения эвристических алгоритмов для любых задач, однако имеется несколько хорошо зарекомендовавших себя подходов, которые оказываются полезными для решения разных типов задач.

Существуют следующие методы решения комбинаторных задач:

- *метод перебора* (подбираются задачи на развитие мышления);
- *табличный метод* (условия вносятся в таблицу и отыскивается решение);
- *дерево вариантов* (использование древовидных и графовых структур данных).

Этапы обучения решению комбинаторных задач

Этап 1. Осуществляется ***хаотичный перебор***, и отыскиваются все возможные варианты в данной задаче.

Пример. Составьте из трех одинаковых по размеру кубиков красного, желтого и синего цвета несколько отличающихся друг от друга построек.

Этап 2. Решение задач с использованием ***систематического перебора***.

Пример. Расставьте знаки «+» и «–» между данными числами 9…2…4, составьте все возможные выражения.

Варианты решения: 9 + 2 + 4; 9 – 2 – 4; 9 – 2 + 4; 9 + 2 – 4.

Пример. Четыре фигуры нарисованы в ряд: большой и маленький квадраты, большой и маленький круги так, что на первом месте находится круг и одинаковые по форме фигуры не стоят рядом. Отгадайте последовательность рассматриваемых фигур.

Варианты решения:

Большой круг - большой квадрат - маленький круг - маленький квадрат.

Большой круг - маленький квадрат - маленький круг - большой квадрат.

Маленький круг - большой квадрат - большой круг - маленький квадрат.

Маленький круг - маленький квадрат - большой круг - большой квадрат.

Пример. Три компаньона одной фирмы хранят ценные бумаги в сейфе, на котором три замка. Компаньоны хотят распределить между собой ключи от замков так, чтобы сейф мог открываться в присутствии двух компаньонов, но не одного. Как это можно сделать?

Этап 3. Решение задач с использованием ***систематического перебора*** и ***средств организации перебора***. К средствам организации систематического перебора относятся таблицы и графы. Прием графического и предметного моделирования является важным средством решения задач повышенной трудности.

В большинстве комбинаторных задач построение плана можно организовать как пошаговый процесс. Алгоритмы *локальной пошаговой оптимизации*, называемые ***«жадными»*** алгоритмами, основываются на следующих соображениях: на каждом шаге решения может определять-

ся значение очередной переменной. В большинстве случаев трудно прогнозировать, какое решение нужно принять на очередном шаге, чтобы в конце концов прийти к оптимальному плану задачи, однако часто можно выбрать решение, которое выглядит наилучшим для данного шага, без учета отдаленных последствий.

Простейшим видом жадных алгоритмов являются алгоритмы последовательного выбора.

Рассмотрим в качестве примера ***задачу о коммивояжере***. Вначале произвольно выберем начальный город и зададимся вопросом, какой город следует посетить в первую очередь. Очевидный ответ – ближайший. На втором шаге и на последующих шагах по той же логике следует выбирать ближайший из еще не посещенных городов. Очевидно, что такой алгоритм не гарантирует оптимальность построенного плана, так как выигрыш на первых шагах приведет к большим потерям на последующих. Однако можно доказать, что при выборе на каждом шаге ближайшего города ожидаемая (наиболее вероятная) длина всего маршрута будет меньше, чем при любом другом выборе.

Для задачи о рюкзаке можно отсортировать товары по убыванию отношения стоимости единицы товара к ее объему. Затем на каждом шаге выбирать максимальное количество очередного товара, умещающееся в оставшемся объеме.

Улучшить качество работы алгоритмов последовательного выбора можно, осуществив ограниченный перебор, т.е. выбрать решение на данном шаге, учитывая результаты двух, трех или более предыдущих шагов. Например, перебрать все варианты двух первых поездок коммивояжера и выбрать из них тот вариант, который имеет минимальную длину, после чего сделать первый шаг. Затем так же перебрать варианты второго и третьего шагов, сделать второй шаг и т.д. Если число перебираемых шагов ограничено, то алгоритм, хотя будет работать значительно медленнее, сохранит полиномиальную сложность. При этом, в силу ограниченности глубины перебора получение оптимального плана не гарантируется, т.е. не гарантируется, что при увеличении глубины перебора качество полученных планов будет всегда возрастать (на это можно только надеяться).

Другим типом алгоритмов локальной пошаговой оптимизации являются алгоритмы ***последовательного улучшения плана*** (симплексный метод).

Пусть некоторый план решения задачи уже построен тем или иным способом (например алгоритмом последовательного выбора): можно

предложить процедуру несложной модификации плана, при которой он остается допустимым планом задачи. Если для модифицированного плана значение целевой функции лучше, чем для исходного, то новый (улучшенный план) выбирается вместо исходного и к нему следует применить улучшающую модификацию. Процесс завершается, когда все возможные модификации будут только ухудшать план.

Например, рассмотрим некоторый маршрут коммивояжера, проходящий последовательно через города *A, B, C, D*. В качестве модификации проведем анализ маршрута, в котором те же города посещаются в порядке *A, C, B, D*. Очевидно, такая перестановка сохраняет допустимость маршрута, а длина нового маршрута может оказаться меньше, чем у исходного.

Недостатком алгоритмов локальной оптимизации является то, что они не всегда приводят к оптимальному плану решения задачи. В частности, для алгоритмов последовательного улучшения очень важен выбор исходного плана. Для построения исходного плана полезную роль может сыграть использование *случайного выбора*. Случайно выбранный план будет, скорее всего хуже, чем план, построенный по алгоритму последовательного выбора, но при многократном повторении комбинации ***«случайный выбор + последовательное улучшение»*** возрастает вероятность хотя бы раз прийти к глобальному оптимуму.

Интересной разновидностью эвристических алгоритмов являются ***генетические алгоритмы***, основанные на применении биологических аналогий, использующих такие понятия, как популяция, ген, мутация, кроссинговер, отбор. *Генетический алгоритм* – это эвристический алгоритм поиска, основанный на решении задач оптимизации и моделирования путем случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию (является разновидностью эволюционных вычислений).

Отличительной особенностью генетического алгоритма является акцент на операторе «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе. При разработке генетического алгоритма важную роль играет выбор способа кодирования допустимых планов задачи.

Каждый план представляется в виде линейной цепочки (хромосомы), состоящей из некоторых символов или чисел (генов). Для хромосом определяются операции мутации (изменения одного или сразу не-

скольких генов) и кроссинговера (получения новой хромосомы как комбинации двух имеющихся).

При этом:

- случайным образом выбирается начальная популяция хромосом;
- новые хромосомы образуются с помощью случайных мутаций и кроссинговера;
- по правилам отбора формируется новое поколение популяции и т.д.

Таким образом, генетические алгоритмы являются своеобразной разновидностью алгоритмов случайного выбора с последовательным улучшением.

В некоторых случаях эвристический алгоритм производит впечатление настолько естественного (самоочевидного) для данной задачи, что кажется, будто этот алгоритм должен всегда давать оптимальный план решения задачи. Однако если нет строгого доказательства оптимальности, алгоритм приходится считать эвристическим, так как интуиция в подобных случаях часто обманывает.

Пример. Рассмотрим решение задачи о коммивояжере с помощью алгоритма последовательного улучшения плана. Поскольку данную задачу можно решить методом перебора, то имеется возможность сравнить эти два подхода по качеству получаемых планов и по времени решения. Для решения задачи о коммивояжере с симметричной матрицей используется следующий алгоритм улучшения плана.

Пусть имеется некоторый допустимый маршрут, в котором присутствуют путь из города A в город B ($A \to B$) и путь из города C в город D ($C \to D$). Рассмотрим другой маршрут, в котором пути $A \to B$ и $C \to D$ заменены на пути $A \to C$ и $D \to B$ (рис.1).

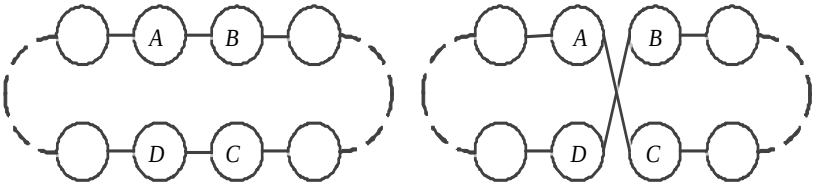

Рис.1. Допустимые маршруты движения

Часть маршрута между городами *C* и *B* будет теперь проходиться в обратном направлении, но это не играет роли, если матрица расстояний симметрична.

Простейший алгоритм последовательного улучшения плана, основанный на рассмотренном преобразовании маршрута, заключается в следующем:

- исходя из некоторого начального плана, следует рассмотреть все пары различных соседних вершин (*A, B*) и (*C, D*) и, как только будут найдены две пары, для которых сумма длин (*AC* + *BD*) меньше, чем сумма (*AB* + *CD*), выполнить описанное преобразование;
- поиск пар повторяется циклически, пока не окажется, что таких пар больше нет;
- зацикливание алгоритма невозможно, поскольку на каждой итерации длина маршрута уменьшается.

При желании можно модифицировать алгоритм, выбирая для выполнения преобразования не первые подходящие пары вершин, а наиболее подходящие пары, для которых преобразование дает максимальный выигрыш. Вероятно, количество итераций при этом сократится, но увеличится трудоемкость выбора пар.

Данный алгоритм можно распространить и на задачи с несимметричной матрицей, но при этом придется сравнивать не суммы длин двух пар, а длины двух подмаршрутов от *A* до *D* через *B* и *C*, с учетом изменения направления обхода вершин от *B* до *C*.

Для того чтобы увеличить вероятность отыскания оптимального (или близкого к оптимальному) маршрута, следует повторить весь алгоритм несколько раз, случайным образом выбирая начальный план.

В ходе решения задачи необходимо проанализировать следующие статистические данные.

1. Сравнить качество планов, получаемых по эвристическому алгоритму, с тем, что дает алгоритм перебора. Это можно сделать для небольших значений числа городов, для которых перебор требует не слишком много времени. Если для некоторых индивидуальных задач эвристический алгоритм не может найти оптимальный маршрут, то следует зафиксировать хотя бы один пример такой задачи.
2. Рассчитать процент задач, для которых совпадают планы, полученные по двум разным алгоритмам.
3. Рассчитать коэффициент качества эвристического алгоритма, выражающий усредненное отношение длины оптимального маршрута к длине маршрута, полученного эвристически. Если во всех проверенных

случаях удается достичь оптимума, то эвристический алгоритм неплох или было проанализировано слишком мало примеров, или же размерность задачи слишком мала. В некоторых случаях хорошо подобранный эвристический алгоритм находит оптимальные решения для подавляющего большинства случайно выбранных индивидуальных задач.

4. Оценить зависимость трудоемкости эвристического алгоритма от числа городов. Показать, что количество пар вершин, среди которых ищутся кандидаты для выполнения операции, квадратично зависит от числа городов. Провести анализ изменений числа итераций преобразования (см. Приложение).

5. Оценить зависимость качества решения (длины найденного маршрута) от количества случайно выбранных начальных планов.

С помощью разработанной программы была решена задача о коммивояжере для значений числа городов n от 6 до 13 (по 100 задач для каждого n), причем каждая задача решалась как с помощью эвристического алгоритма, так и с помощью перебора. Для каждой задачи делалось пять попыток поиска кратчайшего маршрута путем улучшения случайно выбранного допустимого плана.

Первый пример задачи, для которой алгоритмы дали разные результаты, был обнаружен при $n = 6$:

$$(1 \to 3 \to 4 \to 2 \to 6 \to 5 \to 1).$$

Для матрицы (табл.1) эвристический алгоритм находит следующий маршрут коммивояжера.

Таблица 1

Матрица смежности расстояний

	1	2	3	4	5	6
1	∞	53	3	76	34	22
2	53	∞	43	23	23	2
3	3	43	∞	85	47	16
4	76	23	85	∞	88	46
5	34	23	47	88	∞	1
6	22	2	16	46	2	∞

Длина этого маршрута равна 149, и алгоритм улучшения плана не может найти две пары соседних городов, которые позволили бы выполнить улучшающее преобразование.

В то же время переборный алгоритм находит кратчайший маршрут:

$$(1 \to 3 \to 6 \to 5 \to 2 \to 4 \to 1).$$

Длина этого маршрута равна 143. В табл.2 приведены сравнительные характеристики решения задач с помощью эвристического алгоритма и алгоритма перебора. Симметричные матрицы расстояний заполнялись случайными целыми значениями, равномерно распределенными на отрезке 0..99.

Время решения одной задачи эвристическим алгоритмом для данных размерностей достаточно мало, в то время как для алгоритма перебора оно увеличивается примерно в четыре раза при увеличении n на единицу и превышает 38 с (для n = 13). Однако даже для небольших размерностей эвристический алгоритм не всегда находит оптимальный маршрут.

Таблица 2

Сравнительные характеристики решения задач

n	Эвристика		Перебор		Оценка эвристики	
	Среднее время	Кратчайший путь (средний)	Среднее время	Кратчайший путь (средний)	% совпадений	Качество планов
5	0,00	174,55	0,00	174,55	100,00	1,000
6	0,00	177,69	0,00	177,53	99,00	1,000
7	0,00	179,19	0,01	178,48	97,00	0,997
8	0,00	178,45	0,03	178,61	95,00	0,999
9	0,00	187,37	0,18	187,39	89,00	0,993
10	0,00	193,24	0,59	193,49	79,00	0,990
11	0,00	190,75	2.09	187,22	75,00	0,987
12	0,01	192,34	8,21	188,38	68,00	0,983
13	0,01	195,24	38,45	191,88	48,00	0,974

Увеличение числа случайных исходных планов с 5 до 10 позволило добиться совпадения результатов работы алгоритмов эвристического и перебора. Разумеется, это не дает никаких гарантий, что такое совпадение будет всегда иметь место даже для $n \le 13$, и тем более для больших значений.

Для того чтобы оценить трудоемкость использованного эвристического алгоритма, эксперимент был продолжен для значений $n \le 150$ (но уже без перебора). Результаты представлены в табл.3.

Таблица 3

Трудоемкость эвристического алгоритма

n	Среднее время	*n*	Среднее время	*n*	Среднее время
5	0,00	55	0,33	105	1,74
10	0,01	60	0,41	110	2,18
15	0,01	65	0,53	115	2,30
20	0,02	70	0,62	120	2,64
25	0,04	75	0,73	125	2,96
30	0,06	80	0,84	130	3,51
35	0,09	85	1,11	135	3,61
40	0,13	90	1,21	140	3,95
45	0,19	95	1,31	145	4,40
50	0,24	100	1,68	150	4,56

В ходе эксперимента было установлено, что скорость роста времени решения несколько меньше, чем $O(n^3)$ (см. Приложение).

Лабораторное задание

Во всех вариантах заданий требуется разработать эвристический алгоритм для решения поставленной задачи, а также написать и отладить программу на произвольном языке программирования, позволяющую сопоставить решение задачи разработанным алгоритмом и методом перебора.

Программа должна быть протестирована на достаточном количестве случайных задач, чтобы можно было сделать выводы о среднем времени решения и его зависимости от размерности задачи, а также сравнить трудоемкость решения одних и тех же задач с помощью эвристики и с помощью перебора. Поскольку сравнение разнохарактерных алгоритмов по числу типовых операций обычно бывает затруднительно, достаточно фиксировать время решения. Обычно метод перебора требу-

ет значительного времени, поэтому для задач большой размерности лучше использовать эвристический алгоритм.

В настоящей лабораторной работе необходимо:

1) найти конкретный пример задачи, для которой эвристический алгоритм не находит оптимального плана;

2) подсчитать процент задач, для которых переборный и эвристический алгоритмы дают одинаковые результаты, а также усредненную оценку качества планов, полученных с помощью эвристического алгоритма.

Варианты заданий

Вариант	Составить программу
1	***Задача о максимальном цикле***. Пусть дан граф $G = (V, E)$, где $V = \{v_1, v_2,..., v_N\}$ – вершины графа, $E = \{e_1, e_2,..., e_M\}$ – его ребра. Заданы длины ребер L_j ($j = 1..M$). Требуется найти самый длинный простой цикл в графе. Для того чтобы избежать проблем, связанных с тупиковыми маршрутами или с отсутствием циклов в графе, можно считать, что каждая пара вершин соединена ребром
2	***Задача о минимальном доминирующем множестве***. Дан граф $G = (V, E)$, где $V=\{v_1, v_2,..., v_N\}$ – вершины графа, $E = \{e_1, e_2,..., e_M\}$ – его ребра. Требуется найти минимальное (по мощности) множество вершин U такое, что любая из остальных вершин смежна с какой-либо вершиной из U
3	***Задача о максимальном независимом множестве***. Дан граф $G = (V, E)$, где $V = \{v_1, v_2,..., v_N\}$ – вершины графа, $E = \{e_1, e_2,..., e_M\}$ – его ребра. Требуется найти максимальное (по мощности) множество вершин U такое, что никакие две его вершины не смежны
4	***Задача о покрытии***. Дано конечное множество U мощности M и множество его подмножеств U_i ($i = 1...N$), а также цена каждого подмножества C_i. Требуется выбрать набор подмножеств U_i так, чтобы их объединение было равно U, а сумма C_i была при этом минимальна. (Вариант формулировки: дано M языков и N переводчиков, каждый из которых знает некоторое подмножество языков и требует оплату C_i. Требуется найти оптимальное покрытие)

Вариант	Составить программу
5	***Задача о куче камней***. Пусть имеется N камней, известны их веса P_i (i = 1...N) и задано количество куч M. Требуется разложить камни на M куч таким образом, чтобы минимизировать вес самой тяжелой кучи. (Вариант формулировки: дано N программ с длительностями выполнения P_i и M процессоров. Требуется распределить программы так, чтобы раньше закончить их выполнение)
6	***Задача о трех станках***. Дано N деталей, каждая из которых должна быть обработана на станке A, затем на станке B, затем на станке C. Каждый станок может в данный момент времени обрабатывать только одну деталь. Если нужный станок занят, то другие детали могут ожидать его освобождения. Для каждой детали известно длительность ее обработки на каждом станке: T_{Ai}, T_{Bi}, T_{Ci}. Требуется найти такой порядок запуска деталей на обработку (т.е. найти такую перестановку номеров деталей), при котором длительность обработки всего комплекта деталей минимальна

Примечание. В вариантах заданий предложены задачи, принадлежащие классу *NP*-трудных задач. В связи с этим возможность того, что построенный эвристический алгоритм окажется точным, следует считать исключенной.

Контрольные вопросы

1. Как свести задачу оптимизации к задаче поиска?
2. Какой характер имеет зависимость времени перебора от числа переменных?
3. Как зависит глубина рекурсии при переборе от числа переменных?
4. Как зависит общее количество рекурсивных вызовов от числа переменных?
5. Что называется рекордом при решении задач методом перебора?
6. Как реализовать досрочное прекращение метода перебора при получении допустимого плана?

7. Какие структуры данных можно использовать для хранения множества допустимых планов?

8. В какой задаче допустимый план может быть получен раньше, чем определены все переменные.

ЛАБОРАТОРНАЯ РАБОТА № 2

Метод ветвей и границ для решения задачи коммивояжера

***Цель работы*:** ознакомление с методом ветвей и границ и использованием его для решения различных задач.

***Продолжительность работы*:** 2 часа.

Теоретические сведения

Пусть $M = \{m_1,..., m_r\}$ - конечное множество и $f : M \to \Re$ - вещественно-значная функция на нем. Требуется найти минимум этой функции и элемент множества, на котором этот минимум достигается.

Когда имеется та или иная дополнительная информация о множестве, решение этой задачи иногда удается осуществить без полного перебора элементов множества *M*. Но чаще приходится производить полный перебор, при котором возникает задача, как лучше этот перебор организовать.

Метод ветвей и границ применим в случае, когда выполняются специфические дополнительные условия на множество *M* и минимизируемую на нем функцию. Предположим, что имеется вещественно-значная функция φ на множестве подмножеств множества *M* со следующими двумя свойствами:

1) для $\forall\, i$ $\varphi(\{m_i\}) = f(m_i)$,

где $\{m_i\}$ - множество, состоящее из единственного элемента m_i;

2) если $U, V \subseteq M$ и $U \subseteq V$, то $\varphi(U) \geq \varphi(V)$.

В этих условиях можно организовать перебор элементов множества *M* с целью минимизации функции на данном множестве так: разобьем множество *M* на части (любым способом) и выберем ту из его частей Ω_1, на которой функция φ минимальна; затем разобьем на несколько частей множество Ω_1 и выберем ту из его частей Ω_2, на которой функция φ минимальна; затем разобьем Ω_2 на несколько частей и

выберем ту из них, где минимальна φ, и т.д., пока не придем к какому-либо одноэлементному множеству $\{m_i\}$.

Данное одноэлементное множество $\{m_i\}$ называется ***рекордом.*** Функция φ называется ***оценочной***. Очевидно, что рекорд не обязан доставлять минимум функции f ; однако при благоприятных обстоятельствах возникает возможность сократить перебор. Описанный выше процесс построения рекорда состоял из последовательных этапов, на каждом из которых фиксировалось несколько множеств и затем выбиралось одно из них.

Рассмотрим пример использования метода ветвей и границ для решения **задачи о коммивояжере.**

Пусть имеется несколько городов, соединенных некоторым образом дорогами с известной длиной. Орграф называется полным, если в нем имеются все возможные ребра. Требуется установить, имеется ли путь, двигаясь по которому можно побывать в каждом городе только один раз и при этом вернуться в город, откуда путь был начат («обход коммивояжера»), и, если таковой путь имеется, установить кратчайший из таких путей.

Формализуем условие в терминах теории графов. Города будут вершинами графа, а дороги между городами - ориентированными (направленными) ребрами графа, на каждом из которых задана весовая функция: вес ребра - это длина соответствующей дороги. Путь, который требуется найти, - это ориентированный *остовный* простой цикл минимального веса в орграфе, такие циклы называются также *гамильтоновыми* (цикл называется *остовным*, если он проходит по всем вершинам графа; цикл называется *простым*, если он проходит по каждой вершине только один раз; цикл называется *ориентированным*, если начало каждого последующего ребра совпадает с концом предыдущего; вес цикла – это сумма весов его ребер).

Заметим, что вопрос о наличии в орграфе гамильтонова цикла достаточно рассмотреть как частный случай задачи о коммивояжере для полных орграфов.

Если исходный орграф не является полным, то его можно дополнить недостающими ребрами и каждому из добавленных ребер приписать вес ∞, считая, что ∞ - это «компьютерная бесконечность», т.е. максимальное значение из всех возможных в рассмотрениях чисел.

Если во вновь построенном полном орграфе найти гамильтоновый цикл, то при наличии у него ребер с весом ∞ можно будет говорить, что в данном, исходном графе «цикла коммивояжера» нет.

Если же в полном орграфе легчайший гамильтонов цикл окажется конечным по весу, то он и будет искомым циклом в исходном графе.

Отсюда следует, что задачу о коммивояжере достаточно решить для полных орграфов с весовой функцией.

Сформулируем это в окончательном виде:

Пусть $G=[A,B]$ - полный ориентированный граф и $v: B \to \Re$ - весовая функция; найти простой остовный ориентированный цикл («цикл коммивояжера») минимального веса. Пусть $A = \{a_1, \ldots, a_p\}$ - конкретный состав множества вершин и $M = (m_{ij})$, $i, j = 1,\ldots, p$ - весовая матрица данного орграфа, т.е. $m_{ij} = v(a_i, a_j)$, причем для любого $i \to m_{ii} = \infty$.

Основные определения

Пусть имеется некоторая числовая матрица. *Привести строку* этой матрицы означает выделить в строке минимальный элемент (его называют *константой приведения*) и вычесть его из всех элементов этой строки. Очевидно, в результате в этой строке на месте минимального элемента окажется нуль, а все остальные элементы будут неотрицательными. Аналогичный смысл имеет определение - *привести столбец* матрицы.

Привести матрицу по строкам означают, что все строки матрицы приводятся. Аналогичный смысл имеет фраза – *привести матрицу по столбцам*.

Привести матрицу означают, что матрица сначала приводится по строкам, а потом – по столбцам.

Весом элемента матрицы называют сумму констант приведения матрицы, которая получается из данной матрицы заменой обсуждаемого элемента на ∞. Следовательно, словосочетание – *самый тяжелый нуль* в матрице означает, что в матрице подсчитан вес каждого нуля, а затем фиксирован нуль с максимальным весом.

Задача коммивояжера

Опишем пошагово метод ветвей и границ для решения задачи о коммивояжере.

Шаг 1. Фиксируем множество всех обходов коммивояжера, т.е. всех простых ориентированных остовных циклов. Поскольку граф – полный, это множество заведомо непустое. При этом значение оценочной функции будет равно сумме констант приведения данной матрицы (весов ребер графа). Если множество всех обходов коммивояжера обозначить через Г, то сумму констант приведения матрицы весов обозначим через φ(Г).

Шаг 2. Выберем в матрице M_1 самый тяжелый нуль (функцию штрафа); пусть он стоит в клетке (i , j); фиксируем ребро графа (i, j) и разделим множество Г на две части: состоящую из обходов, которые проходят через ребро (i , j), и состоящую из обходов, которые не проходят через ребро (i, j). Затем в полученной матрице вычеркнем строку с номером i и столбец с номером j, причем у оставшихся строк и столбцов сохраним их исходные номера. Наконец, приведем эту последнюю матрицу и запомним сумму констант приведения. В проведенных рассуждениях использовался в качестве исходного только один фактический объект - приведенная матрица весов данного орграфа. По ней было выделено определенное ребро графа и были построены новые матрицы, к которым, конечно, можно применить все то же самое.

При каждом таком повторном применении будет фиксироваться очередное ребро графа. Условимся о следующем действии: перед тем, как в очередной матрице вычеркнуть строку и столбец, в ней надо заменить на ∞ числа во всех тех клетках, которые соответствуют ребрам, заведомо не принадлежащим тем гамильтоновым циклам, которые проходят через уже отобранные ранее ребра.

К выбранному множеству с сопоставленными ему матрицей и числом φ применим, до тех пор пока это возможно, указанный выше алгоритм. В результате получится множество, состоящее из единственного обхода коммивояжера, вес которого равен очередному значению функции φ; таким образом, оказываются выполненными все условия, обсуждавшиеся при описании метода ветвей и границ. Затем осуществляется улучшение рекорда вплоть до получения окончательного ответа.

Расшифровка криптограмм

Пусть дана криптограмма

$$BEST + MADE = MASER\ ,$$

в которой каждая буква шифрует какую-то одну цифру из множества:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Требуется расшифровать данную криптограмму. Если воспользоваться деревом полного перебора, то можно показать, что восемь букв и десять цифр могут быть сопоставлены друг с другом различными способами, число которых достигает значения 1 814 400. Однако если воспользоваться определенными свойствами десятичной системы счисления, правилами сложения десятичных чисел, а также операциями ветвления и отсечения ветвей, то пространство поиска в данной задаче можно качественным образом сузить.

P_4	P_3	P_2	P_1	
	B	***E***	***S***	***T***
	M	***A***	***D***	***E***
M	***A***	***S***	***E***	***R***
5	4	3	2	1

Переносы

Номера разрядов

Рис.1. Задача расшифровки криптограммы

Для удобства рассуждений введем следующие обозначения: столбцы зашифрованных цифр пронумеруем слева направо от 1 до 5. Символы P_1, P_2, P_3, P_4 будут обозначать соответствующую цифру переноса из одного разряда в другой. Если цифра i-го разряда больше или равна 10, то $P_i = 1$, в противном случае $P_i = 0$ (рис.1). Очевидно, что при сложении двух десятичных цифр величина переноса не может превышать одной единицы следующего разряда.

Начнем анализ с пятого разряда. Можно утверждать, что $M = 1$, так как M не может равняться нулю, поскольку не принято писать нуль в начальной позиции, также очевидно, что M не может быть больше единицы. Так как в пятом разряде M целиком образована значением переноса P_4, то получаем - $M = 1$. Определен корень ограниченного дерева поиска (рис.2,*а*).

Обращаясь к четвертому разряду, нельзя с уверенностью сказать, чему равно P_3 – единице или нулю.

Отметим оба варианта на дереве поиска (рис.2,*б*).

Левая ветвь: $P_3 = 0$, так как $P_4 = 1$ и $P_3 = 0$, то $B + M$ должно быть равно $A + 10$. Но из-за того, что $M = 1$, получается, что $B = A + 9$. Но B не может быть больше 9, поэтому $A = 0$ (рис.2,*в*).

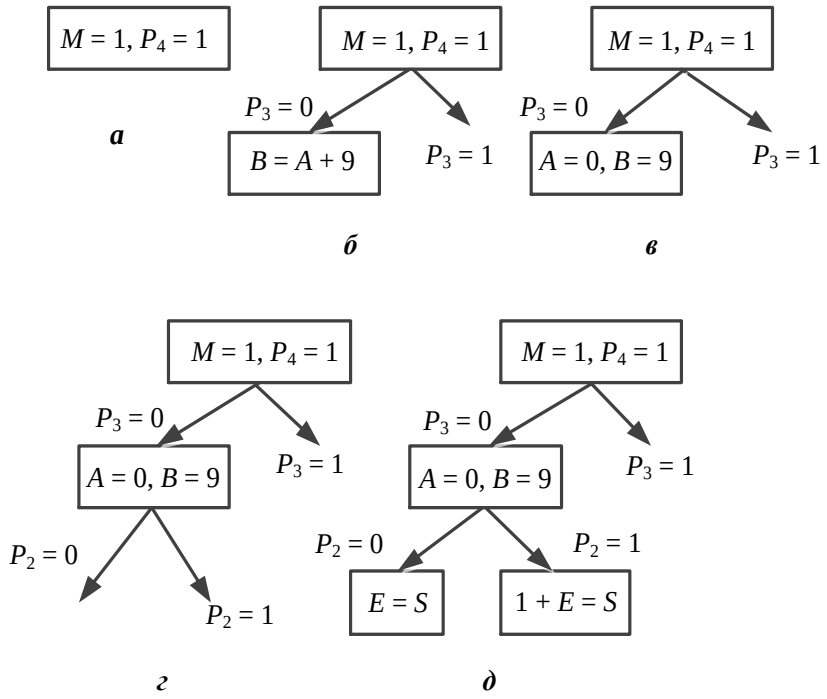

Рис.2. Этапы построения дерева перебора решений для расшифровки криптограммы

Рассматривая ветвь $P_2 = 0$ (рис.2,*г*), нетрудно видеть, что $P_2+E = S$. Поскольку $P_2 = 0$, $A = 0$, то имеем $E = S$, что запрещено правилами построения криптограммы.

Таким образом, путь $P_3 = 0$, $P_2 = 0$ – тупиковый, дальше его продолжать не следует. Если же двигаться по ветви $P_2 = 1$, получаем $1 + E = S$, и этот путь, возможно, следует просматривать далее (рис.2,*д*).

Продолжая аналогичное построение, получим ограниченное дерево поиска для данной криптографической задачи (рис.3). Звездочками обозначены ветви, не имеющие продолжения. Следует отметить, что снача-

ла в качестве принципа ветвления использовался признак наличия или отсутствия переноса, а затем, когда этот признак был исчерпан, пришлось сделать полный перебор по всем возможным значениям буквы *E* (перечислены на рис.3 слева направо).

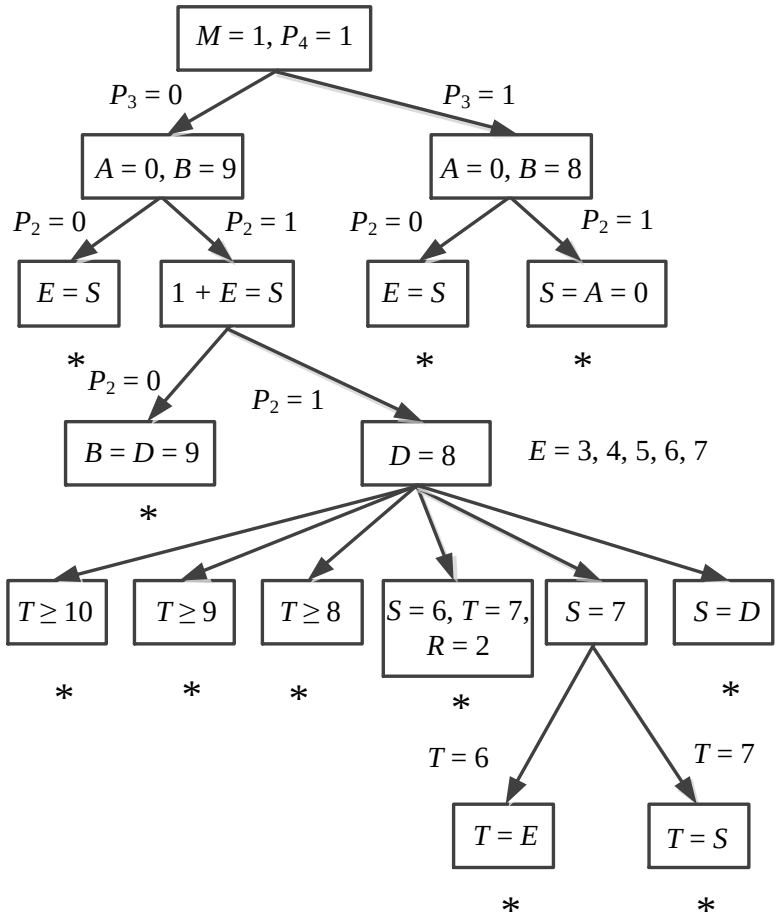

Рис.3. Ограниченное дерево перебора для расшифровки криптограммы

Таким образом, вместо полного дерева перебора удается построить ограниченное дерево, рассмотрение которого позволяет легко найти нужное решение. В результате можно сделать следующий вывод: решение комбинаторных задач существенно упрощается, если вместо полного дерева перебора удается построить ограниченное дерево, обязательно включающее пространство, содержащее решение.

Решение задачи коммивояжера

Методом ветвей и границ решить задачу коммивояжера для графа, содержащего пять вершин.

Пусть исходный ориентированный граф задан матрицей стоимости:

$$C = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \infty & 68 & 73 & 24 & 70 & 9 \\ 2 & 58 & \infty & 16 & 44 & 11 & 92 \\ 3 & 63 & 9 & \infty & 86 & 13 & 18 \\ 4 & 17 & 34 & 76 & \infty & 52 & 70 \\ 5 & 60 & 18 & 3 & 45 & \infty & 58 \\ 6 & 16 & 82 & 11 & 60 & 48 & \infty \end{array}$$

Начальное приведение матрицы стоимости. Матрица стоимости называется *приведенной*, если она имеет в каждой строке и каждом столбце хотя бы один нуль. Операция приведения заключается в вычитании из элементов каждой строки (столбца) минимального элемента этой строки (столбца) – *константы приведения*. Сумма констант приведения образует нижнюю граничную оценку стоимости любого возможного тура.

Вычисление функции штрафа. Функция штрафа – это множество чисел, вычисленных для каждого нуля приведенной матрицы посредством суммирования двух минимальных чисел, из той строки и того столбца, в которых расположен нулевой элемент.

Выбор ребра ветвления. Для ветвления необходимо выбирать ребро, которому соответствует максимальная функция штрафа. Если существует несколько одинаковых максимальных значений функции штрафа, то выбор среди них может быть произвольным.

Вычисление граничной оценки для ветви, соответствующей *невключению ребра в тур*. Эта оценка вычисляется как сумма граничной оценки, соответствующей предыдущему узлу дерева перебора, и выбранного значения функции штрафа.

Вычисление граничной оценки для ветви, соответствующей *включению ребра в тур*. Для вычисления граничной оценки необходимо:

- вычеркнуть в матрице стоимости строку и столбец, соответствующие выбранному ребру;
- скорректировать полученную матрицу таким образом, чтобы устранить возможность досрочного завершения тура (устранить циклы);
- осуществить приведение (если необходимо) полученной матрицы, и если константа приведения отлична от нуля, то сложить эту константу с граничной оценкой предыдущего узла.

Проверка на окончание решения. Если скорректированная матрица имеет размер 2 × 2, и если узел дерева, которому соответствует эта матрица, имеет минимальную граничную оценку, то решение задачи заканчивается: два оставшихся нуля этой матрицы соответствуют двум последним ребрам, которые включаются в тур непосредственно, при этом, очевидно, стоимость тура не изменяется.

Решение. Сначала приведем исходную матрицу по строкам (матрица C_1).

$C_1 =$

	1	2	3	4	5	6	h_i
1	∞	59	64	15	61	0	9
2	47	∞	5	33	0	81	11
3	54	0	∞	77	4	9	9
4	0	17	59	∞	35	53	17
5	57	15	0	42	∞	55	3
6	5	71	0	49	37	∞	11

В последнем столбце этой матрицы (h_i) записаны константы приведения по строкам. Полученную матрицу необходимо привести еще по столбцам.

В результате получаем матрицу C_2 , в которой в столбце h_j записаны константы приведения по столбцам.

$C_2 =$

	1	2	3	4	5	6
1	∞	59	64	0	61	0
2	47	∞	5	18	0	81
3	54	0	∞	62	4	9
4	0	17	59	∞	35	53
5	57	15	0	27	∞	55
6	5	71	0	34	37	∞
h_j	0	0	0	15	0	0

Сумма констант приведения R (оптимальное значение тура) дает нижнюю граничную оценку стоимости всех туров:

$$R = \sum_i h_i + \sum_j h_j = 9 + 11 + 9 + 17 + 3 + 11 + 15 = 75.$$

Вычисляем для ребер, помеченных в матрице C_2 нулем, значения функций штрафа, которые обозначим символом D_{ij}:

$$D_{16} = 0 + 9 = 9; \qquad D_{41} = 17 + 5 = 22;$$
$$D_{25} = 5 + 4 = 9; \qquad D_{53} = 15 + 0 = 15;$$
$$D_{32} = 4 + 15 = 19; \qquad D_{63} = 5 + 0 = 5.$$

Максимальное значение функции штрафа равно 22, на основании чего в качестве ребра ветвления выбирается ребро (4, 1). Разбиваем множество всех возможных туров (R) на два подмножества: {4, 1} - подмножество всех туров, в которые входит ребро (4, 1) и подмножество всех туров {4, 1*}, в которое ребро (4, 1) не входит. Строим первый фрагмент ограниченного дерева перебора (рис.4).

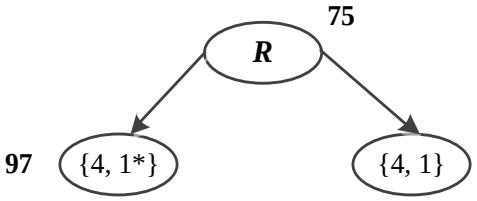

Рис.4. Начальный фрагмент ограниченного дерева перебора

При этом граничная оценка в ветви {4,1*} находится непосредственно, как сумма: 75 + 22 = 97.

Вычеркиваем в матрице C_2 четвертую строку и первый столбец, при этом, чтобы запретить досрочное завершение тура по ребру графа (1, 4), необходимо весу этого ребра присвоить значение ∞ (рис.5).

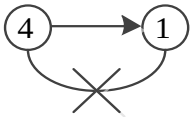

Рис.5. Устранение цикла

В результате подобной корректировки получается матрица C_3.

	2	3	4	5	6
1	59	64	∞	61	0
2	∞	5	18	0	81
$C_3 =$ 3	0	∞	62	4	9
5	15	0	27	∞	55
6	71	0	34	37	∞
h_j	0	0	18	0	0

Для нее сумма коэффициентов по строкам равна нулю. Ее необходимо привести по столбцам (константа приведения $h_4 = 18$), в результате чего получаем приведенную матрицу C_4 и вычисляем граничную оценку в ветви {4, 1}, которая будет равна: 75 + 18 = 93 (рис.6).

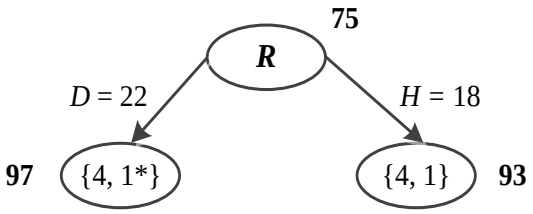

Рис.6. Фрагмент дерева решений с граничными оценками

Сравнивая оценки в листьях дерева приходим к выводу, что ветвление целесообразно продолжать в терминальной вершине с оценкой 93. Так как этому листу соответствует матрица C_4, то с ней необходимо

поступать, как с исходной матрицей, т.е. вычислить для нее значения функций штрафа и выбрать максимальное значение, указывающее на следующее ребро ветвления.

$C_4 =$

	2	3	4	5	6
1	59	64	∞	61	0
2	∞	5	0	0	81
3	0	∞	44	4	9
5	15	0	9	∞	55
6	71	0	16	37	∞

Для матрицы C_4 имеем следующие значения функции штрафа:

$$D_{16} = 59 + 9 = 68, \quad D_{32} = 4 + 15 = 19,$$
$$D_{24} = 0 + 9 = 9, \quad D_{53} = 9 + 0 = 9,$$
$$D_{25} = 0 + 4 = 4, \quad D_{63} = 16 + 0 = 16.$$

Выбираем максимальное значение 68, которому соответствует следующее ребро ветвления (1, 6), после чего можно построить второй фрагмент ограниченного дерева перебора решений (рис.7).

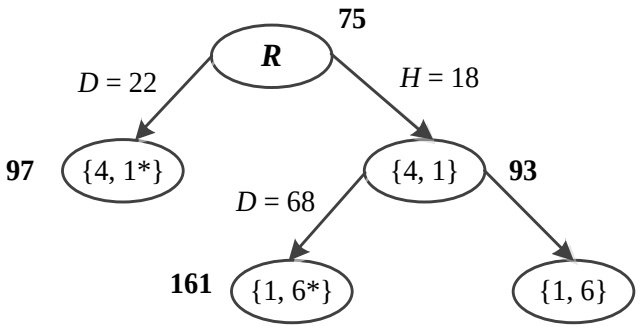

Рис.7. Второй фрагмент дерева перебора решений

Затем приступаем к вычислению граничной оценки в узле дерева {1, 6}. Для этого вычеркиваем в матрице C_4 первую строку и шестой столбец, а также предпринимаем меры для исключения досрочного завершения тура для матрица C_5.

Очевидно что, к данному моменту в тур включено ребро (4, 1) и имеется предложение включить в него ребро (1, 6). Из рис.8 видно, что досрочно завершить тур могли бы ребра (6, 1) и (6, 4).

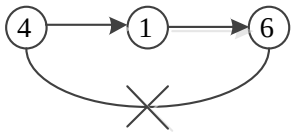

Рис.8. Исключение досрочного завершения тура

Но элемент (6, 1) в матрице C_4 уже отсутствует (эту роль может выполнить лишь элемент (6, 4)), поэтому заменяем его символом ∞. Окончательно матрица C_5 будет иметь вид:

$C_5 =$

	2	3	4	5
2	∞	5	0	0
3	0	∞	44	4
5	15	0	9	∞
6	71	0	∞	37

Матрица C_5 оказалась приведенной ($H = 0$) и, следовательно, оценка в узле {1, 6} остается неизменной (рис.9).

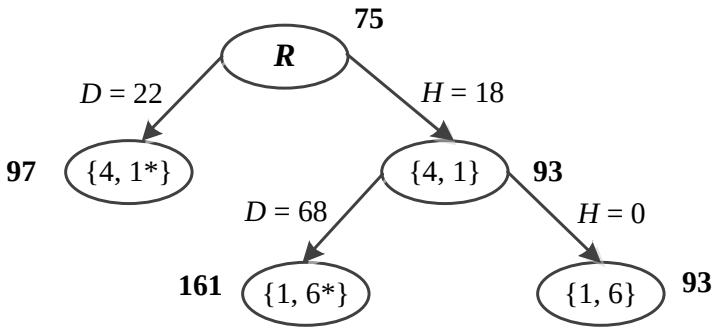

Рис.9. Второй фрагмент дерева решений с граничными оценками

Вычисляем значения функций штрафа для матрицы C_5.

$$D_{24} = 0 + 9 = 9, \quad D_{53} = 9 + 0 = 9,$$
$$D_{25} = 0 + 4 = 4, \quad D_{63} = 37 + 0 = 37.$$
$$D_{32} = 4 + 15 = 19;$$

Определяем следующее ребро ветвления (ребро 6, 3) и вычисляем граничную оценку в том узле дерева перебора, которому соответствуют туры, не содержащие ребро (6, 3) (рис.10). При этом ветвление продолжается от узла с минимальной оценкой, равной 93.

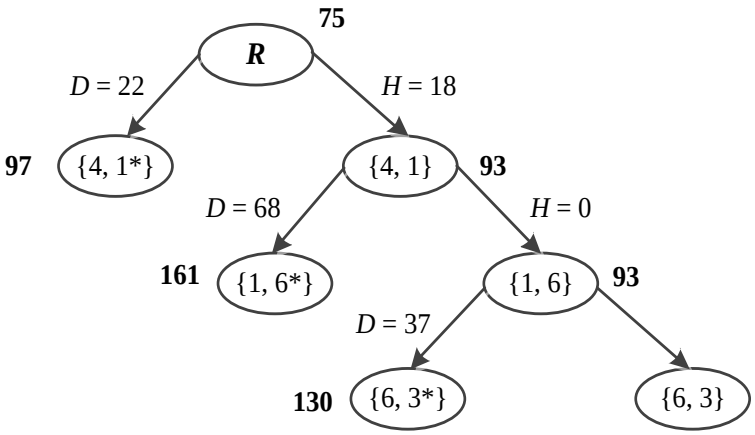

Рис.10. Третий фрагмент дерева решений

Вычеркиваем в матрице C_5 шестую строку и третий столбец и корректируем ее на исключение досрочного завершения тура (рис.11). Это достигается присвоением символа бесконечности элементу (3, 4).

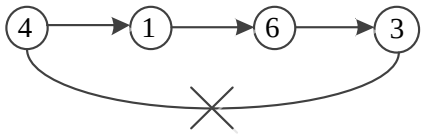

Рис.11. Исключение досрочного завершения тура

В результате получаем матрицу C_6

$C_6 =$

	2	4	5
2	∞	0	0
3	0	∞	4
5	15	9	∞

и после ее приведения – матрицу C_7 .

$C_7 =$

	2	4	5
2	∞	0	0
3	0	∞	4
5	6	0	∞

Так как матрица C_6 не была приведена, то нижняя граничная оценка в узле (6, 3) увеличивается по сравнению с предыдущим узлом ветвления на 9 (константа приведения $h_4 = 9$) и становится равной 102. Тогда третий фрагмент дерева перебора решений с граничными оценками в листьях будет иметь вид, изображенный на рис.12.

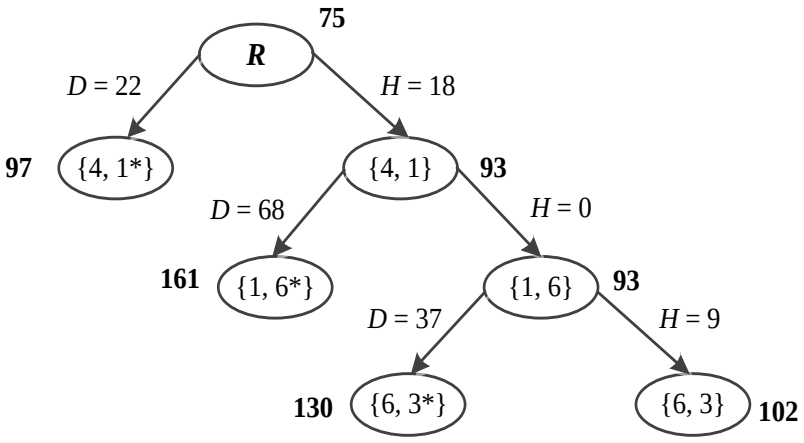

Рис.12. Третий фрагмент дерева решений с граничными оценками

Анализ дерева, изображенного на рис.12 показывает, что ветвление нужно продолжать из узла {4, 1*}, так как граничная оценка там меньше (97), чем в узле, откуда ветвление осуществлялось до сих пор.

Подобная ситуация называется «перескоком», и в этом случае все полученные ранее данные могут быть утеряны. Таким образом, переходим к узлу {4, 1*}, которому соответствует матрица C_8.

$C_8 =$

	1	2	3	4	5	6
1	∞	68	73	24	70	9
2	58	∞	16	44	11	92
3	63	9	∞	86	13	18
4	∞	34	76	∞	52	70
5	60	18	3	45	∞	58
6	16	82	11	60	48	∞

Матрица C_8 не приведена, и после выполнения операции приведения получаем матрицу C_9.

$C_9 =$

	1	2	3	4	5	6
1	∞	59	64	0	61	0
2	42	∞	5	18	0	81
3	49	0	∞	62	4	9
4	∞	0	42	∞	18	36
5	52	15	0	27	∞	55
6	0	71	0	34	37	∞

Полученная в результате приведения матрицы C_8 сумма констант приведения будет равна 97, что соответствует вычисленной ранее через функцию штрафа граничной оценке в узле {4, 1*}. Этот факт является дополнительным признаком правильности выполненных вычислений.

Вычисляем для матрицы C_9 значения функций штрафа:

$$D_{14} = 0 + 18 = 18, \quad D_{42} = 18 + 0 = 18,$$
$$D_{16} = 0 + 9 = 9, \quad D_{33} = 15 + 0 = 15,$$
$$D_{25} = 5 + 4 = 9, \quad D_{61} = 0 + 42 = 42,$$
$$D_{32} = 4 + 0 = 4, \quad D_{63} = 0 + 0 = 0.$$

Очередным ребром ветвления из узла {4, 1*} будет ребро (6, 1), так как ему соответствует максимальное значение функции штрафа. После корректировки матрицы C_9 ребру (1, 6) присваивается значение ∞. Вы-

численные граничные оценки четвертого фрагмента дерева перебора решений будут иметь вид (рис.13).

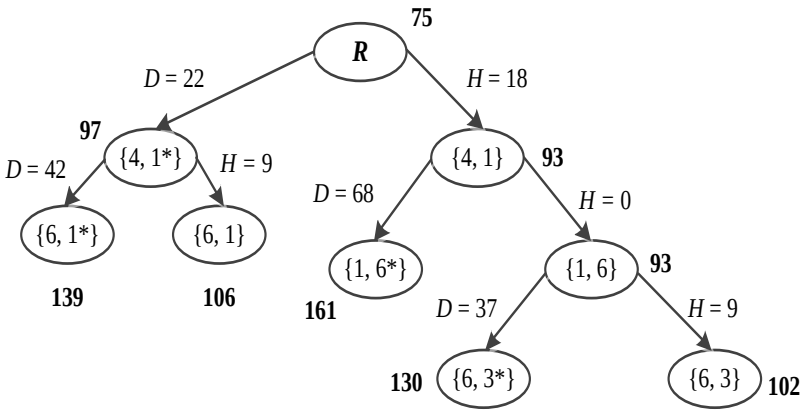

Рис.13. Четвертый фрагмент дерева решений

Из рис.13 видно, что минимальная граничная оценка (102) имеет место в узле {6, 3}, поэтому целесообразно осуществить обратный «перескок» в этот узел и продолжать дальнейшее ветвление из него. Этому узлу соответствует ранее вычисленная матрица C_7 (дадим ей для удобства сквозной нумерации новое обозначение $C_7 = C_{10}$):

$$C_{10} = \begin{array}{c|ccc} & 2 & 4 & 5 \\ \hline 2 & \infty & 0 & 0 \\ 3 & 0 & \infty & 4 \\ 5 & 6 & 0 & \infty \end{array}$$

Вычислим для матрицы C_{10} значения функций штрафа:

$$D_{24} = 0 + 0 = 0, \quad D_{32} = 4 + 6 = 10,$$

$$D_{25} = 0 + 4 = 4, \quad D_{54} = 6 + 0 = 6.$$

Максимальное значение функции штрафа, равное 10, связано с ребром (3, 2), поэтому выбираем его в качестве ребра ветвления, продолжая построение дерева из листа с граничной оценкой 102.

После исключения из матрицы C_{10} третьей строки, второго столбца, а также ее корректировки, получаем приведенную матрицу C_{11}:

$$C_{11} = \begin{array}{c|cc} & 4 & 5 \\ \hline 2 & \infty & 0 \\ 5 & 0 & \infty \end{array}$$

Так как матрица C_{11} уже приведена и имеет размер 2 × 2, то ребра (2, 5) и (5, 4) нужно непосредственно включить в тур, при этом граничная оценка 102 не изменяется.

Окончательный вид тура представлен на рис.14 , а дерево перебора решений - на рис.15.

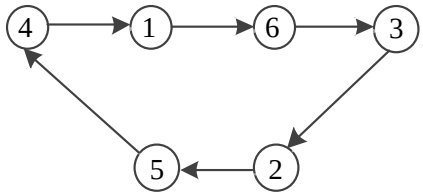

Рис.14. Оптимальный тур

При этом легко проверить, что сумма весов ребер, вошедших в тур, равна 102, что является дополнительным условием правильности полученного решения.

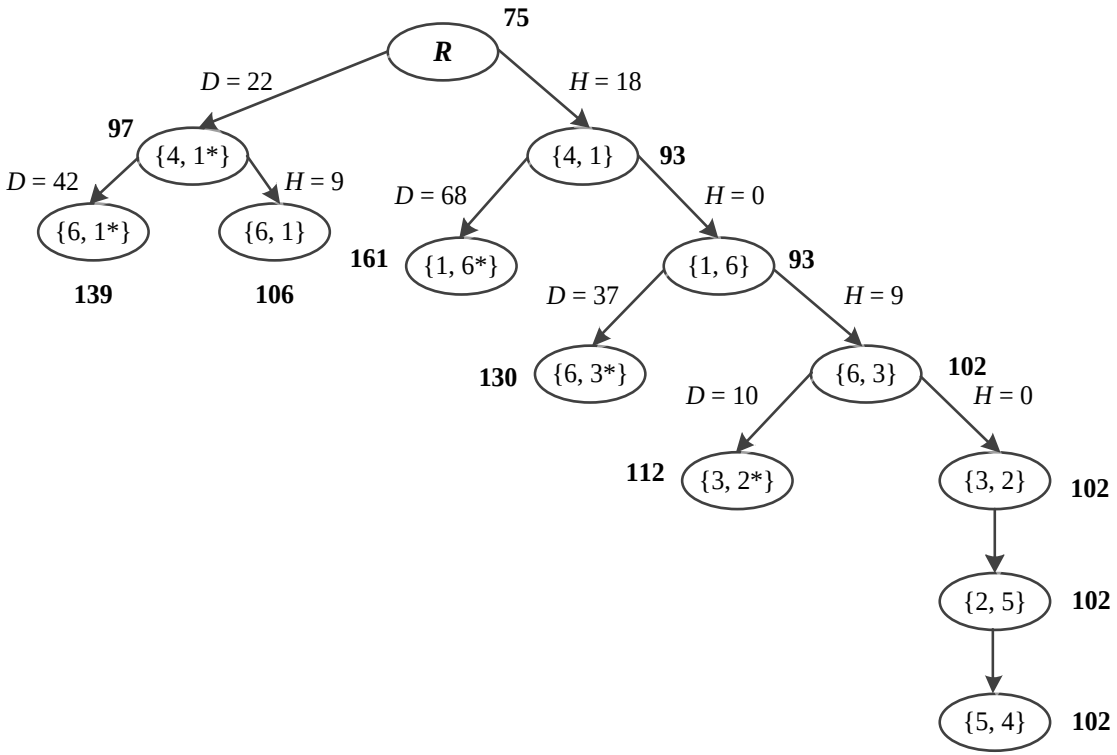

Рис.16. Окончательный вид дерева решений

Лабораторное задание

Алгоритм проведения настоящей лабораторной работы имеет следующий вид.

1. Изучить метод ветвей и границ на примере задачи коммивояжера.

2. Решить задачу коммивояжера для трех графов, содержащих по 6 вершин.

Путь к файлу: D:\ИПОВС\АиСД\ VOYAGER\Коммивояжер1

Путь к файлу: D:\ИПОВС\АиСД\ VOYAGER\Коммивояжер2

Путь к файлу: D:\ИПОВС\АиСД\ VOYAGER\Коммивояжер3

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений.

Исходные матрицы формируются с помощью генератора случайных чисел. Разработанные обучающе-контролирующие системы работают в пошаговом режиме. Результаты вычислений записать в тетрадь.

3. Составить программу решения задачи коммивояжера для графа, заданного матрицей смежности:

	1	**2**	**3**	**4**	**5**
1	∞	7	12	25	10
2	10	∞	9	5	11
3	13	8	∞	6	4
4	6	11	15	∞	15
5	5	9	12	17	∞

4. Решить задачу коммивояжера для задач, представленных в следующем разделе (варианты заданий).

Варианты заданий

Вариант	Решить задачу коммивояжера
1	Пусть граф задан матрицей смежности размером 5×5, в которой помимо диагональных элементов удалены еще два ребра
2	Пусть граф задан матрицей смежности размером 6×6, в которой помимо диагональных элементов удалены еще три ребра
3	Пусть граф задан матрицей смежности размером 5×5, в которой помимо диагональных элементов удалено еще одно ребро
4	Пусть граф задан матрицей смежности размером 6×6, в которой помимо диагональных элементов удалены еще три ребра
5	Пусть граф задан матрицей смежности размером 5×5, в которой помимо диагональных элементов удалены еще два ребра
6	Пусть граф задан матрицей смежности размером 6×6, в которой помимо диагональных элементов удалены еще три ребра
7	Пусть граф задан матрицей смежности размером 5×5, в которой помимо диагональных элементов удалено еще одно ребро
8	Пусть граф задан матрицей смежности размером 6×6, в которой помимо диагональных элементов удалены еще два ребра

Контрольные вопросы

1. Каковы основные принципы метода ветвей и границ?
2. Какое множество называется рекордом?
3. Как сформулировать условие задачи коммивояжера?
4. Что означает привести матрицу по строкам?

5. Как строится дерево перебора для расшифровки криптограмм?
6. Что такое функция штрафа?
7. Как исключается досрочное завершение тура?
8. Что такое нижняя граничная оценка?

ЛАБОРАТОРНАЯ РАБОТА № 3

Имитационное моделирование обслуживания потока заданий на ЭВМ

***Цель работы*:** изучение различных вариантов хранения динамической структуры данных - очереди и разработка методов решения задач с использованием очередей.

Продолжительность работы: 2 часа.

Теоретические сведения

Имитационное моделирование реально существующих объектов и явлений – физических, химических, биологических, социальных процессов, живых и неживых систем представляет собой построение математической модели, описывающей изучаемое явление с достаточной точностью, и последующую реализацию разработанной модели на ЭВМ для проведения вычислительных экспериментов в целях изучения свойств моделируемых явлений. Использование имитационного моделирования позволяет проводить изучение исследуемых объектов и явлений без проведения реальных (натурных) экспериментов.

Очередь – структура данных, в которой каждый вновь поступающий элемент занимает крайнее положение (*конец очереди*). При выдаче информации из очереди выбирается элемент, расположенный в очереди первым (*начало очереди*), и оставшиеся элементы продвигаются к началу. Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях.

В качестве области приложений выбрана эффективная организация выполнения потока заданий на вычислительных системах. В качестве примера использования очереди предлагается задача разработки системы имитации однопроцессорной ЭВМ.

Пусть для вычислительной системы (ВС) с одним процессором и однопрограммным последовательным режимом выполнения поступаю-

щих заданий требуется разработать программную систему имитации процесса обслуживания заданий в ВС. При построении модели функционирования ВС должны учитываться следующие основные моменты обслуживания заданий:

- генерация нового задания;
- постановка задания в очередь для ожидания момента освобождения процессора;
- выборка задания из очереди при освобождении процессора после обслуживания очередного задания.

По результатам вычислительных экспериментов система имитации должна формировать информацию об условиях проведения эксперимента: интенсивность потока заданий, размер очереди заданий, производительность процессора, число тактов имитации. Помимо этого необходимо учитывать полученные в результате имитации показатели функционирования вычислительной системы:

1) количество поступивших в ВС заданий;
2) количество отказов в обслуживании заданий из-за переполнения очереди;
3) среднее количество тактов выполнения заданий;
4) количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания.

Показатели функционирования вычислительной системы, получаемые с помощью систем имитации, могут использоваться для оценки эффективности применения ВС (см. Приложение). Анализ показателей позволяет принять рекомендации о целесообразной модернизации характеристик ВС (например, при длительных простоях процессора и при отсутствии отказов от обслуживания заданий желательно повышение интенсивности потока обслуживаемых заданий и т.д.).

Условия и ограничения:

1) при планировании очередности обслуживания заданий возможность задания приоритетов не учитывается;
2) моменты появления новых заданий и моменты освобождения процессора рассматриваются как случайные события.

Метод решения

Динамическая структура является математической структурой, которой соответствует частично упорядоченное базовое множество *M*,

операции вставки и удаления, элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Таким образом, *очередь* – динамическая структура, операции вставки и удаления переводят ее из одного состояния в другое. При этом добавление новых элементов осуществляется в конец очереди, а извлечение – из начала очереди (обслуживание «первым пришел – первым обслужен»).

Важной задачей при реализации системы обслуживания очереди является выбор структуры хранения, обеспечивающей решение проблемы эффективного использования памяти без переупаковок и без использования связных списков, требующих дополнительных затрат памяти на указатели. Как и в случае со стеком, в качестве *структуры хранения* очереди предлагается использовать одномерный (одноиндексный) массив, размещаемый в динамической области памяти. В связи с характером обработки значений, располагаемых в очереди, для указания хранимых в очереди данных необходимо иметь два указателя – на начало и конец очереди. Эти указатели увеличивают свое значение: один при вставке, другой при извлечении элемента.

Таким образом, в ходе функционирования очереди может возникнуть ситуация, когда оба указателя достигнут своего наибольшего значения и дальнейшее пополнение очереди станет невозможным, несмотря на наличие свободного пространства в очереди. Одним из решений проблемы «движения» очереди является организация на одномерном массиве кольцевого буфера (рис.1).

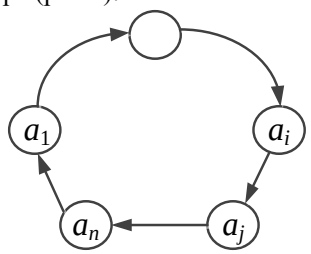

Рис.1. Общая схема структуры хранения вида кольцевой буфер

Кольцевым буфером является структура хранения, получаемая из вектора расширением отношения следования парой $p(a_n, a_1)$. Структура хранения очереди в виде кольцевого буфера может быть определена как одномерный массив, размещаемый в динамической области памяти и

расположение данных в котором определяется с помощью следующего набора параметров (рис.2):

- ***pMem*** – указатель на память, выделенную для кольцевого буфера;
- ***MemSize*** – размер выделенной памяти;
- ***MaxMemSize*** – размер памяти, выделяемый по умолчанию (если при создании кольцевого буфера явно не указано требуемое количество элементов памяти);
- ***DataCount*** – количество сохраненных в очереди значений;
- ***Hi*** – индекс элемента массива, в котором хранится последний элемент очереди;
- ***Li*** – индекс элемента массива, в котором хранится первый элемент очереди.

В связи с тем что структура хранения очереди во многом аналогична структуре хранения стека, предлагается класс для реализации очереди построить наследованием от класса стек. При наследовании достаточно переопределить методы Get и GetNextIndex. В методе Get изменяется индекс для получения элемента (извлечение значений происходит из начала очереди), метод GetNextIndex реализует отношение следования на кольцевом буфере.

Для работы с очередью предлагается реализовать следующие операции:

1) метод Put – добавить элемент (добавление элемента в очередь аналогично добавлению в стек, можно использовать метод класса стек);
2) метод Get – удалить элемент (при удалении элемента из очереди необходимо возвратить значение из динамического массива по индексу начала очереди, переместить указатель начала очереди и уменьшить количество элементов);
3) метод IsEmpty – проверить очередь на пустоту;
4) метод IsFull – проверить очередь на полноту.

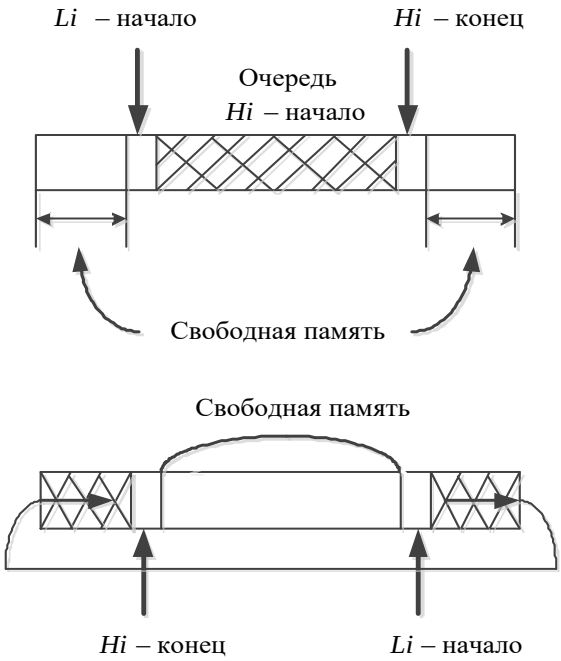

Рис.2. Общая схема реализации кольцевого буфера на одноиндексном массиве

Моделирование потока задач. Для моделирования момента появления нового задания можно использовать значение датчика случайных чисел. Если значение датчика меньше некоторого порогового значения q_1 ($0 \le q_1 \le 1$), то считается, что на данном такте имитации в ВС поступает новое задание. Таким образом, параметр q_1 можно интерпретировать как величину, регулирующую интенсивность потока заданий, – новое задание генерируется в среднем один раз за $(1/q_1)$ тактов.

Моделирование работы процессора. При моделировании момента завершения обработки очередного задания с помощью датчика случайных чисел формируется новое случайное значение, и если это значение меньше порогового q_2 ($0 \le q_2 \le 1$), то делается вывод, что на данном такте имитации процессор завершил обслуживание очередного задания и готов приступить к обработке задания из очереди ожидания: тем самым параметр q_2 можно интерпретировать как величину, характеризу-

ющую производительность процессора вычислительной системы, – каждое задание обслуживается в среднем за $(1/q_2)$ тактов.

Моделирование вычислительной системы. Возможная схема имитации процесса поступления и обслуживания заданий в ВС состоит в следующем.

1) каждое задание в системе представляется некоторым однозначным идентификатором (например, порядковым номером задания);

2) для проведения расчетов фиксируется число тактов работы системы;

3) на каждом такте оценивается состояние потока задач и процессора;

4) регистрация нового задания в ВС может быть сведена к запоминанию идентификатора задания в очередь ожидания процессора. Ситуацию переполнения очереди заданий следует понимать как нехватку ресурсов ВС для ввода нового задания (отказ от обслуживания);

5) при моделировании процесса обработки заданий следует учитывать, что процессор может быть занят обслуживанием очередного задания, либо находиться в состоянии ожидания;

6) в случае освобождения процессора предпринимается попытка его загрузки: для этого извлекается задание из очереди ожидания;

7) ожидание (простой) процессора возникает в случае, когда при завершении обработки очередного задания очередь ожидающих заданий оказывается пустой;

8) после проведения всех тактов имитации осуществляется вывод характеристик ВС:

- количество поступивших в ВС заданий в течение всего процесса имитации;
- количество отказов в обслуживании заданий из-за переполнения очереди (процент отказов определяется как отношение количества не поставленных в очередь заданий к общему количеству сгенерированных заданий, умноженное на 100 %);
- среднее количество тактов выполнения задания;
- количество тактов ожидания (простоя) процессора из-за отсутствия в очереди заданий для обслуживания (процент простоя процессора определяется как отношение количества тактов простоя процессора к общему количеству тактов имитации, умноженное на 100 %).

Разработка программного модуля. С учетом сформулированных выше предложений к реализации обработки заданий процессором, представляется следующая модульная структура программы:

1) TStack.h, TStack.cpp – модуль с классом, реализующим операции над стеком;

2) TQueue.h, TQueue.cpp – модуль с классом, реализующим операции над очередью;

3) TJobStream.h, TJobStream.cpp – модуль с классом, реализующим поток задач;

4) TProc.h, TProc.cpp – модуль с классом, реализующим процессор;

5) QueueTestkit.cpp – модуль программы тестирования.

При этом, с учетом предложенных к реализации алгоритмов, можно сделать следующие объявления классов.

Класс *TStack* (файл TStack.h).

Класс *TQueue* (файл TQueue.h):

```
class TQueue : public TStack { protected:
int Li; // индекс начала очереди
int GetNextIndex (int index); // получить следующий индекс
public:
TQueue (int Size = MaxMemSize):TStack(Size) {Li = 0;};
TElem Get ( void ) ; // извлечь значение
};
```

Тип TElem описывает тип элементов структуры данных (определяется в TStack.h). Для определения методов класса TQueue ниже приведен фрагмент файла TQueue.cpp (модуль реализации).

```
#include ”Queue.h”
int Queue :: GetNextIndex(int index) { // Получение следующего индекса
return ++ index % MemSize ;
}
TElem Queue :: Get ( void ) {// извлечь значение
if ( ~IsEmpty() { // очередь не пуста?
TElem tmp = pMem[Li];
Head = GetNextIndex( Li );
DataCount--;
return tmp;
}
}
```

В конструкторе вызывается конструктор базового класса, а размер области памяти определяется как параметр конструктора (по умолчанию устанавливается MaxMemSize) и явно учитывает тип элемента оче-

реди (TElem). Кроме того, устанавливается значение индекса начала очереди $Li = 0$.

Деструктор – методы IsEmpty, IsFull, Put наследуются из базового класса. Перед обращением к функции Put необходимо проверять очередь на полноту. Метод Get переопределяется для очереди и обеспечивает получение значения из очереди по указателю начала очереди; операция выполняется с удалением возвращаемого значения из очереди. Перед обращением к этой функции необходимо проверять очередь на пустоту.

Метод GetNextIndex переопределяется в классе очереди и обеспечивает представление очереди в виде кольцевого буфера, позволяя устанавливать значения индексов начала и конца очереди при достижении ими граничного значения в нуль.

Класс *TJobStream* (файл TJobStream.h)

```
class TJobStream {
private:
double JobIntens; // интенсивность потока задач
public:
TJobStream (int Intens);
int GetNewJob(void); // генерация нового задания
};
```

Класс *TProc* (файл TProc.h)

```
class TProc {
private:
double ProcRate; // производительность процессора
int JobId; // Id выполняемого задания
public:
TProc (int Rate);
int IsProcBusy(void) ; // процессор занят?
int RunNewJob (int JobId); // приступить к выполнению нового задания
};
```

Этапы разработки

Важной частью настоящей лабораторной работы является организация поэтапной разработки программ. С учетом наличия работоспособного класса TStack, предлагается следующая последовательность разработки.

Этап 1. Реализация программ поддержки очереди.

Этап 2. Реализация системы имитации обслуживания заданий.

Этап 3. Выполнение дополнительных заданий настоящей лабораторной работы.

При выполнении настоящей лабораторной работы продолжайте практическое освоение технологии итеративной разработки с тщательным тестированием работы программ. Для проверки правильности разрабатывается достаточное количество тестов, которые гарантированно проверяют семантические ошибки. Переход к следующему этапу разработки происходит только при успешном выполнении тестов. При проверке разработанных программ очередного этапа должны быть задействованы тесты всех предшествующих этапов.

Учитывая готовность класса TStack, при разработке программ очереди может быть предложена следующая последовательность реализации.

1. Реализация конструктора TQueue, перегрузка метода следования GetNextIndex. Тестирование разработанных программ.

2. Реализация метода извлечения из очереди и тестирование могут быть осуществлены следующим образом:

- разработка спецификации класса TJobStream, реализация конструктора и метода генерации нового задания;
- разработка спецификации класса TProc, реализация конструктора;
- реализация метода опроса состояния процессора и метода запуска нового задания на выполнение и тестирование;
- разработка управляющей программы системы имитации, обеспечивающей циклическое выполнение тактов имитации: генерация нового задания, проверка состояния процессора, запуск нового задания и тестирование;
- разработка программы вывода результатов имитации;
- выполнение вычислительных экспериментов и подготовка отчета.

Входными параметрами вычислительного эксперимента являются: интенсивность потока задач, производительность процессора и количество тактов имитации. Максимальный размер очереди определен константным значением.

Необходимо предусмотреть два режима работы программы – при небольшом числе тактов имитации следует выполнять потактовую распечатку состояния ВС и печать статистики, для длительных периодов имитации следует проводить только печать статистики.

Программы настоящей лабораторной работы следует разрабатывать с обязательной проверкой возможных ошибочных ситуаций (например, выборка значения из пустой очереди). При обнаружении ошибочных ситуаций необходимо завершить программу с выдачей аварийного кода завершения или применить обработку исключений (варианты заданий).

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи.

Варианты заданий

Вариант	Содержание
1	Реализовать возможность задания приоритетов, поступающих в систему задач. В этом случае задачи в очереди должны упорядочиваться в соответствии со своими приоритетами
2	Дополнить постановку настоящей лабораторной работы возможностью моделирования для поступающих заданий с различной длительностью выполнения
3	Расширить разработанную систему для возможности имитации многопроцессорных вычислительных систем, в которых могут быть несколько процессоров. Возможное расширение постановки настоящей лабораторной работы может состоять в использовании нескольких процессоров: от 3 до 6. Важным моментом при анализе поведения таких систем является выбор стратегии выделения имеющихся процессоров для выполнения заданий (могут возникать ситуации длительного откладывания обработки заданий, для которых необходимо большое количество процессоров)
4	Расширить разработанную систему имитации возможностью использования нескольких очередей входных заданий
5	При расширении разработанных программ провести анализ трудоемкости необходимой модификации программного кода: количество методов, в которые требуется внести изменения; количество новых внесенных ошибок разработки и т.п.

Критерии оценивания лабораторной работы

Лабораторная работа считается сданной при выполнении следующих требований: реализованы все классы, при малых значениях количества тактов имитации итоговая статистика соответствует результатам потактового выполнения.

Набор тестов для проверки.

Тест 1. *Количество тактов имитации* 10.
Размер очереди – 3.
Интенсивность потока задач – 0,5.
Производительность процессора – 0,5.

Тест 2. *Количество тактов имитации* 10.
Размер очереди – 3.
Интенсивность потока задач – 0,5.
Производительность процессора – 0,2.

Тест 3. *Количество тактов имитации* 10.
Размер очереди – 3.
Интенсивность потока задач – 0,2.
Производительность процессора – 0,5.

В тесте 1 система является сбалансированной, в тесте 2 малоинтенсивный поток задач при высокопроизводительном процессоре (в результате вычислительного эксперимента должен быть получен большой процент простоя процессора), в тесте 3 интенсивный поток задач при малопроизводительном процессоре (вычислительный эксперимент должен показать большое количество заданий, которым отказано в обслуживании из-за переполнения очереди заданий).

Контрольные вопросы

1. Какие изменения могут потребоваться в структуре хранения при реализации очереди с приоритетами?
2. Какие дополнительные варианты (подходы) могут быть предложены при разработке системы имитации для более адекватного описания функционирования ВС?
3. Насколько точно использованный способ генерации заданий описывает реально существующие потоки заданий в ВС? Какие до-

полнительные схемы генерации входного потока заданий могут быть предложены?

4. Может ли разработанная система имитации быть использована в других областях приложений?

5. Каковы правила работы с очередью, реализация которой выполнена в библиотеке STL?

6. Для решения каких задач может потребоваться использование очереди?

ЛАБОРАТОРНАЯ РАБОТА № 4

Моделирование с использованием генераторов случайных чисел

***Цель работы*:** ознакомление с алгоритмами генерирования случайных чисел и числовыми характеристиками случайных величин.

***Продолжительность работы*:** 2 часа.

Теоретические сведения

Многие явления в природе, технике, экономике и других областях носят случайный характер. В этом случае величина, принимающая свои значения в зависимости от исходов некоторого испытания (опыта), называется случайной величиной.

Пусть X – дискретная случайная величина, возможными значениями которой являются числа $x_1, x_2, \ldots, x_n$.

Обозначим через $p_i = P(X = x_i)$, где $i = 1, 2, \ldots, n$ - вероятности этих значений, т.е. p_i - вероятность события, состоящего в том, что X принимает значение x_i.

Закон распределения полностью задает дискретную случайную величину. Однако часто закон распределения случайной величины неизвестен. В таких ситуациях ее описывают числовыми характеристиками.

Числовые характеристики случайных величин

Случайные величины характеризуются следующими числовыми параметрами:

- *математическое ожидание* ($M(X)$) - это статистическое среднее случайной величины.

$$M(X) = \sum_{i=1}^{n} x_i p_i \ ,$$

где x_i - значение случайной величины; p_i - вероятность появления этой величины.

$$\sum_{i=1}^{n} p_i = 1.$$

Для равновероятных событий: $M(X) = \frac{\sum_{i=1}^{n} x_i}{n}$;

- *дисперсия* - это математическое ожидание квадрата отклонения случайной величины от ее математического ожидания.

$$D(x) = \sum_{i=1}^{n} (x_i - M(x))^2 \cdot p_i \ ;$$

- корень квадратный из дисперсии называется *средним квадратичным отклонением* случайной величины (σ).

$$\sigma(x) = \sqrt{D(x)} \ ;$$

- *коэффициент корреляции* определяется для двух потоков случайных величин:

$$R(x, y) = \frac{M(x \cdot y) - M(x) \cdot M(y)}{\sigma(x) \cdot \sigma(y)}.$$

Потоки случайных величин, для которых $R(x, y) = 0$, называются *некоррелированными* (*независимыми*). При этом $-1 \le R(x, y) \le 1$.

Рассмотрим алгоритмы для детерминированной выборки случайных чисел.

Метод середины квадрата

Один из ранних генераторов случайных чисел, принадлежащих Джону фон Нейману (1946 г.), известен как *метод середины квадрата*.

Метод используется для генерации k-разрядных псевдослучайных чисел. Должно быть задано k-разрядное начальное число x_0 (для удобства предполагаем, что k четно). Это число возводится в квадрат и получается число y . Число y должно иметь $2k$ разрядов. Если число разрядов меньше $2k$, то число y дополняется слева нулями. Затем в y выделяют средние k разрядов, которые и дают новое случайное число.

Алгоритм сводится к выполнению следующих шагов.

Шаг 0. [Инициализация] $x := x_0$.

Шаг 1. [Основной цикл] *FOR* $j := 1$ *TO* m *DO* *шаг 2*; *Stop*.

Шаг 2. [Генерация нового случайного числа x_j]

$$y := x^2; \quad x_j := (\text{средние } k \text{ разрядов числа } y); \quad x := x_j.$$

Число y будет иметь $2k$ разрядов, а следующее случайное число x_j получается, если удалить по $k/2$ разрядов с каждого конца y. Десятичная точка помещается перед первым разрядом числа x_j до поступления его на выход случайного генератора.

Пример. Получить три случайных числа методом середины квадрата

$k = 4$; $x_0 = 3167$; $y = 10\underline{0298}89$;
$x_1 = 0298$; $y = 00\underline{0888}04$;
$x_2 = 0888$; $y = 00\underline{7885}44$;
$x_3 = 7885$.

У переменной y подчеркнуты (выделены) средние k разрядов.

Пример. Для $k = 4$ и $x_0 = 2134$ получить первые восемь псевдослучайных чисел, выработанных алгоритмом:

$x_0^2 = 04\ 5539\ 56$	$x_1 = 0{,}5539$
$x_1^2 = 30\ 6805\ 21$	$x_2 = 0{,}6805$
$x_2^2 = 46\ 3080\ 25$	$x_3 = 0{,}3080$
$x_3^2 = 09\ 4864\ 00$	$x_4 = 0{,}4864$
$x_4^2 = 23\ 6584\ 96$	$x_5 = 0{,}6584$
$x_5^2 = 43\ 3490\ 56$	$x_6 = 0{,}3490$
$x_6^2 = 12\ 1801\ 00$	$x_7 = 0{,}1801$
$x_7^2 = 03\ 2436\ 01$	$x_8 = 0{,}2436$

Несмотря на видимую случайность чисел, генерируемых алгоритмом, ему свойственны недостатки. В самом деле, если в последовательности когда-нибудь появится число 0,0000, то все следующие за ним числа будут также равны 0,0000.

Таким образом, многое зависит от начального выбора значений k и x_0.

Линейный конгруэнтный метод

Метод используется для генерации последовательности $x_1, x_2, \ldots, x_m$ из m псевдослучайных чисел. Должны быть заданы следующие входные значения:

- b - целочисленный множитель $(b \geq 1)$;
- x_0 - начальное случайное целое число $(x_0 \geq 1)$;
- k - шаг $(k \geq 0$ целое);
- m - целочисленный модуль $(m \geq x_0, b, k)$.

Случайные числа генерируются по рекуррентной формуле:

$$x_j = (b \cdot x_{j-1} + k) \bmod m.$$

Алгоритм сводится к выполнению следующих шагов.

Шаг 1. [Основной цикл]

FOR $j := 1$ *TO* m *DO* *шаг* 2, *шаг* 3; *Stop*.

Шаг 2. [Генерация нового необработанного случайного числа]

$$x_j := (b \cdot x_{j-1} + k) \,(\bmod\, m).$$

Число x_j должно лежать в интервале $0 \leq x_j \leq m$. По определению, если $a \bmod m = x$ для целых a и m, то x есть целый остаток от деления a на m. Например, $5 \bmod 3 = 2$.

Шаг 3. [Генерация следующего случайного числа]

$$y_j := x_j / m.$$

(y_j будет лежать в интервале $0 \leq y_j < 1$ и обладать требуемым распределением).

Пример. Получить три случайных числа линейным конгруэнтным методом:

$x_0 = 27$; $b = 5$; $k = 10$; $m = 40$;
$x_1 = (5 \cdot 27 + 10) \bmod 40 = 145 \bmod 40 = 25$;
$x_2 = (5 \cdot 25 + 10) \bmod 40 = 15$;
$x_3 = (5 \cdot 15 + 10) \bmod 40 = 5$.

Пример. Для $k = 0, m = 2^{10}, b = 101, x_0 = 432$ получить первые восемь псевдослучайных чисел, выработанные алгоритмом:

$x_1 = 624$	$y_1 = 624/1024 = 0{,}610$
$x_2 = 560$	$y_2 = 560/1024 = 0{,}546$
$x_3 = 240$	$y_3 = 240/1024 = 0{,}234$
$x_4 = 688$	$y_4 = 688/1023 = 0{,}673$

$x_5 = 880$	$y_5 = 880/1024 = 0{,}859$
$x_6 = 816$	$y_6 = 816/1024 = 0{,}796$
$x_7 = 496$	$y_7 = 496/1024 = 0{,}486$
$x_8 = 944$	$y_8 = 944/1024 = 0{,}923$

Существует такой выбор параметров k, b, m, x_0, при котором алгоритм будет генерировать числа на отрезке [0; 1], представляющиеся непредсказуемыми и удовлетворяющие определенным статистическим критериям. Для всех практических целей эти числа оказываются последовательностью наблюдений равномерно распределенной случайной переменной.

Полярный метод генерации случайных чисел с нормальным распределением

Метод используется для генерации двух независимых случайных чисел с распределением $N(0, 1)$ из двух независимых равномерно распределенных случайных чисел. Распределение $N(0, 1)$ преобразуется в $N(\mu, \sigma^2)$ с использованием центральной предельной теоремы.

Алгоритм сводится к выполнению следующих шагов.

Шаг 1. [Генерация двух равномерно распределенных случайных чисел]. Генерируются два независимых случайных числа u_1 и u_2 с распределением $U(0, 1)$; $v_1 := 2 \cdot u_1 - 1; v_2 := 2 \cdot u_2 - 1.$

(v_1 и v_2 имеют распределение $U(-1; +1)$).

Шаг 2. [Вычисление и проверка s].

$s := v_1^2 + v_2^2$; *IF* $s \geq 1$ *THEN GOTO шаг* 1.

Шаг 3. [Вычисление n_1 и n_2].

$n_1 := v_1 \sqrt{(-2 \ln s)/s}$; $n_2 := v_2 \sqrt{(-2 \ln s)/s}$

(n_1 и n_2 распределены нормально).

Stop.

Алгоритм имеет важное вычислительное преимущество: обычно он генерирует по одному числу с нормальным распределением на каждое число с равномерным распределением. При этом следует убедиться, что компенсируется дополнительное время, затрачиваемое на вычисление натуральных логарифмов и квадратных корней.

Лабораторное задание

Алгоритм проведения лабораторной работы имеет следующий вид.

1. Изучить методы получения случайных чисел.
2. Для каждого из перечисленных выше методов провести анализ числовых параметров случайных величин для списков различной размерности (1000 <= величина выборки <= 10000).

Путь к файлу: D:\ИПОВС\АиСД\RND

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений.

Результаты экспериментов занести в форму табл.1.

Форма таблицы 1

Результаты эксперимента

Название метода			
Числовые характеристики	Число элементов выборки		
	N_1	N_2	N_3
Математическое ожидание			
Дисперсия			
Коэффициент корреляции			

3. Построить гистограммы распределения числовых параметров.
4. Составить программы для решения следующих задач (варианты заданий).

Варианты заданий

Вариант	Составить программу
1, 6, 11, 16	**Задача 1.** Используя генератор случайных чисел, смоделировать процесс заполнения зрительного зала кинотеатра на 300 мест. Зрительный зал имеет вид прямоугольника $N \times M$
2, 7, 12, 17	**Задача 2.** Используя генератор случайных чисел, смоделировать процесс парковки автомобилей коммерческой фирмы у тротуара. Длина тротуара D, длина автомобиля N

Вариант	Номер задачи
3, 8, 13, 18	**Задача 3.** Используя метод середины квадрата, получить 10 случайных чисел
4, 9, 14, 19	**Задача 4.** Используя линейный конгруэнтный метод, получить 10 случайных чисел
5, 10, 15, 20	**Задача 5.** Используя полярный метод, получить 10 случайных чисел

Контрольные вопросы

1. Какие типы числовых характеристик случайных величин существуют?
2. Что показывает коэффициент корреляции?
3. В чем особенность метода середины квадрата?
4. Какие исходные данные используются в линейном конгруэнтном методе?
5. Какие случайные числа получают полярным методом?

ЛАБОРАТОРНАЯ РАБОТА № 5

Машина Тьюринга

***Цель работы*:** ознакомление со способами записи алгоритмов с помощью машины Тьюринга и методикой оценки их эффективности.

***Продолжительность работы*:** 2 часа.

Теоретические сведения

В 1937 г. английский математик Алан Матисон Тьюринг опубликовал работу, в которой уточнил понятие алгоритма, прибегая к воображаемой вычислительной машине.

Машина Тьюринга – один из способов записи алгоритма. Машина Тьюринга – алгоритм, записью которого является функциональная таблица или функциональная диаграмма, а правилом выполнения – описание ее устройства.

Машина Тьюринга так же, как и конечный автомат, является дискретным устройством преобразования информации. Приведем ее точное определение, а затем интерпретацию ее работы.

Машиной Тьюринга называется частичное отображение M:

$$\{Q_0, Q_1,\ldots,Q_{n-1}\} \times \{0, 1\} \longrightarrow \{Q_0, Q_1,\ldots,Q_{n-1}\} \times \{L, R\} \times \{0, 1\},$$

где $\{Q_0, Q_1,\ldots,Q_{n-1}\}$ - множество состояний машины; L, R - обозначает «влево», «вправо».

Тот факт, что отображение частичное, означает, что M может быть определено не для всех наборов аргументов. Машина Тьюринга работает с бесконечной в обе стороны лентой, разбитой на ячейки, в каждой из которых написан один из символов - 0 или 1.

Считывающая (записывающая) головка машины обозревает в каждый момент времени одну из ячеек и за один такт, сменяющий два последовательных момента времени, может перемещаться влево или вправо.

Машина Тьюринга в каждый момент времени находится в одном из состояний Q_0, Q_1, …, Q_{n-1}, и в следующий момент времени переходит в другое состояние или остается в том же. Кроме того, машина мо-

жет изменять символ, стоящий в обозреваемой ячейке. Все эти преобразования - изменения состояния, информации на ленте, направления движения полностью определяются отображением M, а именно, если $M(i, \varepsilon) = (j, L, \eta)$, то в случае, когда машина находится в состоянии Q_i, а на обозреваемой в данный момент ячейке написан символ ε, машина должна записать в эту ячейку вместо символа ε символ η, перейти в состояние Q_j и сдвинуться на одну ячейку влево.

В случае, когда $M(i, \varepsilon) = (j, R, \eta)$, те же действия будут сопровождаться сдвигом вправо.

Например, равенство $M(2, 1) = (1, R, 1)$ означает, что, находясь в состоянии Q_2 и обозревая ячейку, в которой написан символ 1, машина должна сохранить в этой ячейке символ 1, сдвинуться вправо и перейти в состояние Q_1. Если же $M(i, \varepsilon)$ не определено, то машина, находясь в состоянии Q_i и обозревая ячейку с символом ε, прекращает свою работу, не изменяя своего состояния, информации на ленте и никуда не сдвигаясь.

Замечание. Существуют различные модификации машины Тьюринга (машина Поста, машина Минского и т.д.). Некоторые модификации предусматривают на ленте не символы 0 или 1, а буквы некоторого конечного алфавита $A = \{a_1, a_2, \ldots, a_m\}$. В некоторых определениях разрешается не только сдвиг головки машины вправо или влево, но и оставление на прежней позиции. Однако различные модификации машины Тьюринга эквивалентны между собой в том смысле, что классы функций, вычислимых на этих машинах, совпадают.

Структура машины Тьюринга

В качестве исполнителя алгоритмов Тьюрингом был предложен автомат, состоящий:

- из бесконечной ленты, разбитой на ячейки;
- головки (каретки), способной передвигаться над лентой, от ячейки к ячейке, считывать символы, записанные на ленте, записывать символы в ячейки.

В каждой ячейке ленты может быть записан только один из определенного множества символов, называемого алфавитом. За одно срабатывание каретка способна выполнить следующие действия:

- считать символ из ячейки, над которой она находится;
- записать символ в ячейку, над которой она находится;

- переместиться либо влево, либо вправо на следующую ячейку, либо остаться на месте.
- изменять свое внутреннее состояние.

Предполагается, что каретка может находиться в одном из состояний определенного множества состояний. Одним из ее действий, наряду с перечисленными выше, является переход из одного состояния в другое.

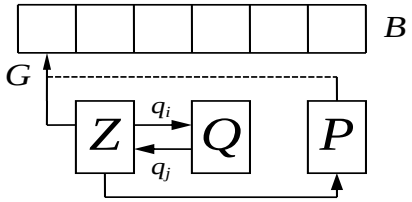

Рис.1. Структурная схема машины Тьюринга

На рис.1 представлена структурная схема машины Тьюринга, где обозначены буквами:

B – внешняя память машины, которую можно интерпретировать, как неограниченную в обе стороны ленту, разделенную на элементарные ячейки;

Q – внутренняя память машины, определяющая состояния, в которых находится машина в любой момент времени;

G – считывающая (записывающая) головка;

Z – логический блок машины. Этот блок формирует символ, который будет записан на ленту, а также управляет переходами машины из одного состояния в другое;

P – устройство, управляющее головкой.

Рассмотрим основные кодовые символы для управления машиной:

- *R* – движение головки вправо;
- *L* – движение головки влево;
- *H* – продолжать обозревать текущую ячейку.

Символы, записанные на ленте *B*, образуют внешний или исходный алфавит. Среди символов исходного алфавита выделяется пустой символ. Чаще всего это символ λ .

Тогда информация на ленте может иметь следующий вид:

$$\{\lambda \quad \lambda \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad \lambda \quad \lambda\}.$$

Алфавит внутренних состояний машины задается множеством:

$$\{q_0, q_1, q_2, \ldots, q_k\},$$

где q_0 - начальное состояние, определяющее начало алгоритма; q_k - конечное состояние, определяющее конец алгоритма.

Говорят, что головка находится в стандартном положении, если она располагается у самого левого, отличного от пустого символа.

Машина Тьюринга работает следующим образом:

1) на ленту записывается исходная информация, и головка устанавливается в исходное положение. Затем через логический блок считывается информация с ленты, и по закону функционирования машины определяется символ, который нужно записать на ленту;

2) на пересечении строки q_i и столбца a_j указывается тройка символов:

$$q_i^1 , a_j^1 , d_k .$$

Здесь q_i^1 - состояние, в которое переходит машина; a_j^1 - символ, который будет записан на ленту; d_k - указатель перемещения головки.

Запись алгоритма определяется составлением функциональной таблицы (табл.1).

Таблица 1

Функциональная таблица для записи алгоритма

Состояния	Исходный алфавит	
	0	1
q_0	$q_1 0R$	$q_0 1R$
q_1	$q_1 0R$	$q_2 1R$
q_2	$q_1 0R$	$q_k 0L$

Исходными данными являются двоичные символы 0 и 1. Работа алгоритма определяется четырьмя состояниями: q_0, q_1, q_2, q_k.

Тройка символов $q_1 0R$, записанная на пересечении строки q_0 и столбца 0, означает: если машина находится в состоянии q_0 и с ленты будет считан символ 0, то машина перейдет в состояние q_1, на ленту будет записан символ 0, и головка сдвинется на один символ вправо. Если информация в ячейке не обновляется или машина остается в прежнем состоянии, то функциональная табл.1 упрощается (табл.2).

Таблица 2

Упрощенная функциональная таблица

Состояния	Исходный алфавит	
	0	1
q_0	q_1R	R
q_1	R	q_2R
q_2	q_1R	q_k0L

От функциональной таблицы можно перейти к функциональной диаграмме. Функциональная диаграмма - способ записи алгоритма в виде графа или графовой структуры (рис.2).

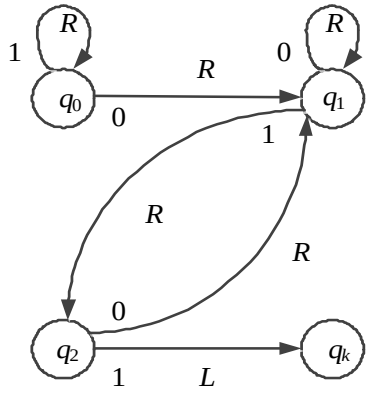

Рис.2. Функциональная диаграмма

Примеры записи алгоритмов

Пример. Пусть задан двоичный код, заключенный в пустые символы λ. Составить машину Тьюринга для получения копии двоичного кода на ленте.

Например, исходные данные – {λλλ1010011λλλ}, результирующее множество – {λλλ1010011*1010011λλλ}.

Анализ алгоритма.

Шаг 1. Движемся вправо, пока не встретим символ λ; ставим на место λ символ *.

Шаг 2. Движемся влево до символа λ и сдвигаемся на один символ вправо; записываем вместо 1 символ α.

Шаг 3. Движемся вправо до символа λ и вместо λ ставим 1.

Шаг 4. Движемся влево до любого символа, отличного от 1 или 0. Сдвигаемся на один символ вправо. Заменяем 0 на β, 1 на α, и т.д., пока не будет получена копия двоичного кода. После этого заменяем α на 1 , β на 0.

Пример. Пусть функциональной табл.3 задана работа машины Тьюринга с исходным алфавитом A ={ 1, λ}.

Таблица 3

Функциональная таблица

Состояние	Исходный алфавит	
	1	λ
q_1	1R	1R

Данная функциональная таблица позволяет бесконечно заполнить всю ленту единицами вправо от выбранной точки.

Ниже приведена табл.4, выполняющая те же самые операции до тех пор, пока на ленте не встретится символ 0.

Таблица 4

Функциональная таблица

Состояние	Исходный алфавит		
	1	λ	0
q_1	1R	1R	q_k1H

Пример. Составить функциональную табл.5 для инвертирования двоичного кода.

Например, исходное множество – {λλλ01101λλλ}, результирующее множество – {λλλ10010λλλ}.

Таблица 5

Функциональная таблица

Состояние	Исходный алфавит		
	0	1	λ
q_0	$1R$	$0R$	q_kH

Пример. Составить алгоритм преобразования числа из множества {1, 2, 3, 4, 5} в унарную запись (табл.6).

Замечание. Унарная запись – представление числа палочками, например 3 ⟶|||

Таблица 6

Функциональная таблица

Состояния	Исходный алфавит							
	0	1	2	3	4	5	λ	*
q_0	R	R	R	R	R	R	$q_1{*}H$	—
q_1	q_kH	q_2R	q_3R	q_4R	q_5R	q_6R	—	L
q_2	—	—	—	—	—	—	q_k1H	R
q_3	—	—	—	—	—	—	q_21R	R
q_4	—	—	—	—	—	—	q_31R	R
q_5	—	—	—	—	—	—	q_41R	R
q_6	—	—	—	—	—	—	q_51R	R

Композиция машин Тьюринга

Написать программу работы машины Тьюринга - значит составить диаграмму или таблицу ее функционирования. Программирование машины Тьюринга состоит в умении разделить процедуру решения задачи на последовательность трех шагов: запись символа, считывание символа, перемещение головки на одну позицию.

Если требуется написать достаточно сложную программу, то для упрощения процесса программирования используются операции композиции машин Тьюринга, т.е. построение сложной машины из более простых. Для этого выполняются две операции: умножения машин Тьюринга и итерации машин Тьюринга.

Умножение машин Тьюринга. Пусть заданы две машины Тьюринга T_1 и T_2. В начальный момент времени t_0 на ленте имеется не-

которая конфигурация, которую начинает обрабатывать машина T_1, отправляясь от некоторого начального состояния $q^1{}_0$, причем головка машины в момент t_0 находится напротив ячейки с номером $l^1{}_0$ (рис.3).

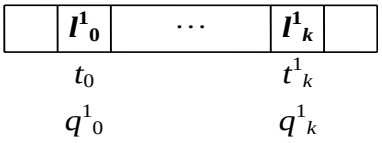

Рис.3. Умножение машин Тьюринга (работа машины T_1)

Тогда к некоторому моменту $t^1{}_k$ машина T_1 перейдет в состояние $q^1{}_k$, а головка машины остановится напротив ячейки $l^1{}_k$. В момент времени $t^1{}_k$ машину T_1 отключаем, и с этого момента начинает работать машина T_2, отправляясь от своего начального состояния $q^2{}_0$. Причем в момент времени t_1 головка будет расположена напротив ячейки $l^1{}_k$ (рис.4).

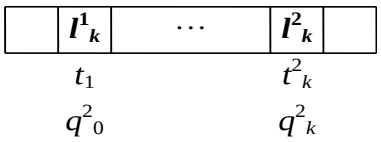

Рис.4. Умножение машин Тьюринга (работа машины T_2)

В момент времени $t^2{}_k$ машина T_2 закончит работу и остановится напротив ячейки $l^2{}_k$. Из рис.3 и рис.4 видно, что результатом последующей работы машин T_1 и T_2 будет некоторая машина T, функциональная таблица которой строится по правилам: верхняя часть таблицы описывает машину T_1, а нижняя часть – работу машины T_2, причем состояние останова машины T_1 – $q^1{}_k$ отождествляется с начальным состоянием машины T_2.

Пример. Пусть заданы две машины T_1 и T_2 (табл.7 и 8).

Таблица 7

Функциональная таблица машины T_1

	Состояния	Исходный алфавит	
		0	1
T_1:	q_0	$q_1 0 L$	$q_0 1 L$
	q_1	$q_1 0 L$	$q_k 1 H$

Таблица 8

Функциональная таблица машины T_2

T_2:	Состояние	Исходный алфавит	
		0	1
	q_0	q_k0H	q_00L

Тогда композиция машин T_1 и T_2 будет выглядеть следующим образом (табл.9).

Таблица 9

Функциональная таблица композиции машин

$T = T_1 \cdot T_2$	Состояния	Исходный алфавит	
		0	1
	q_0	q_10L	q_01L
	q_1	q_10L	q_21H
	q_2	q_k0H	q_20L

В случае, когда одна из перемножаемых машин имеет несколько состояний останова, указывается, какой из остановов предыдущего сомножителя отождествляется с начальным состоянием последующего.

Система команд (таблица или диаграмма) машины T является результатом объединения системы команд машин T_1 и T_2. Нетрудно видеть, что машина $T = T_1 \cdot T_2$ работает так, как если бы после завершения работы машины T_1 начала работать машина T_2. Очевидно, что произведение машины Тьюринга некоммутативно, т.е. $T_1 \cdot T_2 \neq T_2 \ T_1$.

Если машина T_1 (записываемая в произведении первой) имеет не одно, а два заключительных состояния, то появляется неоднозначность (индетерминизм) в том, с каким из этих состояний отождествлять начальное состояние машины T_2.

Пример. Пусть машина T_1 имеет два состояния останова.

$$\text{Тогда } T = T_1 \cdot \begin{cases} (1) & T_2 \\ (2) & \end{cases}.$$

Начальное состояние машины T_2 отождествляется с первым состоянием останова машины T_1. Таким образом, машина T будет иметь два заключительных состояния: одно совпадает с состоянием останова машины T_2, другое - со вторым заключительным состоянием машины T_1.

Итерация машин Тьюринга. Итерация машин Тьюринга заключается в отождествлении *r*-го состояния останова машин Тьюринга с начальным состоянием. Машина *T* может быть задана следующим образом:

$$T = \begin{cases} (1) \\ \cdot \\ \cdot \\ \cdot \\ (S) \end{cases}$$

Значения в скобках указывают на отождествление состояний останова с начальным состоянием. Если машина T_1 имеет лишь одно состояние останова, то в этом случае машина *T* не будет иметь останова и станет бесконечной. Если итерация произведена над машиной, которая, в свою очередь является результатом умножения и итерации других машин, то соответствующее число точек ставится над теми машинами, чьи состояния останова и начальные отождествляются.

Пример. Пусть задана машина Тьюринга.

$$T = \dot{T}_1 \cdot \ddot{T}_2 \cdot \begin{cases} (1) & \dot{T}_3 \\ (2) & T_4 \cdot \ddot{T}_5 \cdot \\ (3) & T_6 \end{cases}$$

Здесь показано умножение и итерация машин Тьюринга. При этом состояние останова машины T_3 отождествляется с начальным состоянием машины T_1, а состояние останова машины T_5 – с начальным состоянием машины T_2.

Машина Поста. Конструктивно к машине Тьюринга близка машина Поста. Принципиально она отличается двоичным алфавитом входных и выходных данных (в машине Тьюринга алфавит не определен, а выбирается).

В машине Тьюринга следует сконструировать управляющий автомат, а в машине Поста - программу решения задачи. Как следствие - универсальность и простота машины Поста.

Система команд машины Поста:

- ***Shr*** - движение вправо к соседней позиции;
- ***Shl*** - движение влево к соседней позиции;

- ***Wr1*** - читать текущую ячейку: если в ячейке символ 0, то записать 1; если в ячейке символ 1, то останов неприменимости;
- ***Wr0*** - читать текущую ячейку: если в ячейке символ 1, то записать 0; если в ячейке символ 0, то останов неприменимости;
- ***Jmp j0, j1*** - читать текущую ячейку: если в ячейке символ 0, то перейти к команде *j*0; если в ячейке символ 1, то перейти к команде *j*1;
- ***Stop*** - останов.

Упорядоченный набор команд (программа) применим к данной проблеме и является алгоритмом ее решения, если применение программы завершится командой *Stop*. Если произойдет останов по неприменимости, то программа не применима к данной проблеме и проблема алгоритмически не разрешима.

В теории вычислительных машин машина Поста представляет собой минимальную универсальную и полную систему команд и элементарную конструкцию.

Лабораторное задание

Алгоритм проведения настоящей лабораторной работы имеет следующий вид.

1. Изучить особенности работы машины Тьюринга.
2. Используя обучающе-контролирующие системы, моделирующие работу машины Тьюринга, решить задачи (табл.10):

Путь к файлу: D:\ИПОВС\АиСД\ TURING\Тьюринг1

Путь к файлу: D:\ИПОВС\АиСД\ TURING\Тьюринг2

Система работает в диалоговом режиме с использованием меню. Вся необходимая информация во время работы системы отображается на экране дисплея и не требует специальных пояснений (варианты заданий).

Варианты заданий

Вариант	Составить функциональную таблицу и функциональную диаграмму
1	Пусть задан произвольный двоичный код. Получить на ленте зеркальное отображение кода. Например, исходное множество - {λλλ01101λλλ}, результирующее множество -{λλλ10010*01001λλλ}

Вариант	Составить функциональную таблицу и функциональную диаграмму
2	Составить алгоритм преобразования десятичных чисел {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} в унарную запись.
3	Составить алгоритм преобразования унарной записи в десятичные числа {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
4	Задан произвольный двоичный код. Разбить его на тройки символов, разделенные символом *. Например, исходное множество - {λλλ011010110l0λλλ}, результирующее множество – {λλλ011*010*110*10λλλ}
5	Задан произвольный двоичный код. Получить копию этого двоичного кода. Например, исходное множество - {λλλ01101λλλ}, результирующее множество - {λλλ10010*10010λλλ}
6	Построить машину Тьюринга, вычисляющую $U((n)_1)=(n\text{-}1)_1$, где $n > 0$ и $(n)_1$ означает запись числа n в унарной форме, т.е. в виде $\underbrace{\| \ldots \ldots \|}_{n}$. Другими словами, машина Тьюринга должна стирать одну палочку в записи числа

Контрольные вопросы

1. Какова структура машины Тьюринга?
2. Что такое внутренняя и внешняя память машины Тьюринга?
3. В чем заключается особенность построения функциональных таблиц?
4. Какова связь функциональной таблицы с функциональной диаграммой?
5. В чем смысл умножения машин Тьюринга?
6. Что такое итерация машин Тьюринга?

Библиографический список

1. ***Ахо А., Хопкрофт Дж., Ульман Дж.*** Структуры данных и алгоритмы. – М.: СПб.: Киев: ИД «Вильямс», 2001.
2. ***Брудно А.Л., Каплан Л.И.*** Московские олимпиады по программированию. – М.: Наука, 1990.
3. ***Вирт Н.*** Алгоритмы и структуры данных / Пер. с англ. – М.: Мир, 2001.
4. ***Гагарина Л.Г., Колдаев В.Д.*** Алгоритмы и структуры данных: учеб. пособие. – М.: Финансы и статистика; Инфра-М, 2009.
5. ***Кирюхин В.М., Лапунов А.В., Окулов С.М.*** Задачи по информатике. Международные олимпиады 1989–1996 гг. – М.: ABF, 1996.
6. ***Кнут Д.*** Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. – М.: Мир, 2000.
7. ***Колдаев В.Д.*** Основы алгоритмизации и программирования: учеб. пособие / *Под ред. Л.Г. Гагариной*. – М.: ИД «ФОРУМ» – Инфра-М, 2009.
8. ***Колдаев В.Д.*** Численные методы и программирование: учеб. пособие. / *Под ред. Л.Г. Гагариной*. – М.: ИД «ФОРУМ» – Инфра-М, 2010.
9. ***Колдаев В.Д., Лупин С.А.*** Архитектура ЭВМ: учеб. пособие. – М.: ИД «ФОРУМ» – Инфра-М, 2009.
10. ***Колдаев В.Д.*** Основы логического проектирования: учеб. пособие. – М.: ИД «ФОРУМ» – Инфра-М, 2011.
11. ***Кормен Т., Лейзерсон Ч., Ривест Р.*** Алгоритмы: построение и анализ. – М.: МЦНМО, 2000.
12. ***Лабораторный практикум*:** учебно-метод. пособие / Сост. И.В. Барышева, И.Б.Мееров, А.В. Сысоев, Н.В. Шестакова / *Под ред.* В.П. *Гергеля*. – Нижний Новгород: Нижегородский госуниверситет, 2017.
13. ***Мейн М., Савитч У.*** Структуры данных и другие объекты в C++ / Пер. с англ. – М.: ИД «Вильямс», 2002.
14. ***Хусаинов Б.С.*** Структуры и алгоритмы обработки данных. Примеры на языке Си: учеб. пособие. – М.: Финансы и статистика, 2004.
15. ***Шень А.*** Программирование. Теоремы и задачи. – М.: МЦНМО, 2004.

Приложение

Функция сложности алгоритма

Эффективность программы является очень важной ее характеристикой. Эффективность программы имеет две составляющие: память (или пространство) и время. Пространственная эффективность измеряется количеством памяти, требуемой для выполнения программы. Компьютеры обладают ограниченным объемом памяти. Если две программы реализуют идентичные функции, то программа, которая использует меньший объем памяти, характеризуется большей пространственной эффективностью. Иногда память становится доминирующим фактором в оценке эффективности программ. Однако в последний годы в связи с быстрым ее удешевлением эта составляющая эффективности постепенно теряет свое значение.

Временна́я эффективность программы определяется временем, необходимым для ее выполнения. Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временно́й, так и пространственной сложности. Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных. Например, некоторый алгоритм может существенно зависеть от размера обрабатываемого массива. Если время обработки удваивается с удвоением размера массива, то порядок временно́й сложности алгоритма определяется как размер массива. Порядок алгоритма - это функция, доминирующая над точным выражением временно́й сложности.

Функция $f(n)$ имеет порядок $O(g(n))$, если имеется константа K и счетчик n_0, такие, что $f(n)(K \cdot g(n)$ для $n > n_0$.

Время обработки массива определяется из уравнения:

Действительное время (Длина массива) =
= Длина массива^2 + 5 × Длина массива + 100.

Виды функции сложности алгоритмов

***O*(1).** В алгоритмах константной сложности большинство операций в программе выполняются один или несколько раз. Любой алго-

ритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность.

***O*(*n*).** Время работы программы линейно обычно, когда каждый элемент входных данных требуется обработать лишь линейное число раз.

***O*(n^2), *O*(n^3), *O*(n^a).** Полиномиальная сложность. $O(n^2)$ - квадратичная сложность, $O(n^3)$ - кубическая сложность.

***O*(log(*n*)).** Когда время работы программы логарифмическое, программа начинает работать намного медленнее с увеличением *N*. Такое время работы встречается обычно в программах, которые делят большую проблему на более мелкие и решают их по отдельности.

***O*(*n*log(*n*)).** Такое время работы имеют те алгоритмы, которые делят большую проблему на мелкие, и решив их, соединяют эти решения.

***O*(2^n).** Экспоненциальная сложность. Такие алгоритмы чаще всего возникают в результате подхода, именуемого методом грубой силы.

Временна́я функция сложности

Программист должен уметь анализировать алгоритмы и определять их сложность. Временна́я сложность алгоритма может быть посчитана исходя из анализа его управляющих структур. Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, то все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью. Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов.

Как правило, около 90 % времени работы программы требует выполнение повторений и только 10 % составляют непосредственно вычисления. Анализ сложности программ показывает, что в большинстве случаев это циклы наибольшей глубины вложенности. Повторения могут быть организованы в виде вложенных циклов или вложенной рекурсии. Эта информация может использоваться программистом для построения более эффективной программы следующим образом.

Прежде всего можно попытаться сократить глубину вложенности повторений. Затем следует рассмотреть возможность сокращения количества операторов в циклах с наибольшей глубиной вложенности. Если 90 % времени выполнения составляет выполнение внутренних циклов,

то 30 %-ное сокращение этих процедур приводит к 27 %-му снижению времени выполнения всей программы.

Теоретическая и практическая функции сложности

В зависимости от конкретной формы этих критериев сложность алгоритма, в свою очередь, подразделяется на *практическую* и *теоретическую*. Практическая сложность обычно оценивается во временны́х единицах (секунды, минуты, количество временны́х тактов процессора, количество выполнения циклов и т.д.). Практическая емкостная сложность выражается, как правило, в битах, байтах, словах и т.п.

Перечислим основные факторы, от которых может зависеть сложность алгоритма:

- быстродействие ЭВМ и ее емкостные ресурсы (в первую очередь объем оперативной памяти). В самом деле, чем ниже тактовая частота процессора, чем меньше объем оперативного запоминающего устройства, тем медленнее выполняются арифметические и логические операции, тем чаще (для больших задач) приходится обращаться к медленно действующей внешней памяти, и, следовательно, большее время уходит на реализацию алгоритма;
- выбранный язык программирования. Задача, запрограммированная, например, на языке Ассемблер, в общем случае решится быстрее, чем по тому же самому алгоритму, но запрограммированному на языке более высокого уровня, например, на Pascal, C++;
- выбранный математический метод формулирования задачи;
- искусство и опыт программиста. В общем случае по одному и тому же алгоритму опытный программист напишет более эффективно работающую программу, чем его начинающий коллега.

Функция $f(n)$ в ряде случаев может иметь достаточно сложную аналитическую форму. Поскольку для временно́й теоретической сложности большее значение имеет не столько вид функции, сколько порядок ее роста, то во многих математических дисциплинах, в том числе и в теории алгоритмов функцию $f(n)$ определяют как $O(g(n))$ и говорят, что она порядка $g(n)$ для больших n, если

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{const} \neq 0,$$

где $f(n)$ и $g(n)$ - экспериментальная и теоретическая функции сложности.

Пример. Определить функцию сложности алгоритма по результатам эксперимента:

n	Количество сравнений
6	54

Решение. Вначале найдем экспериментальную функцию сложности (*О*э).

Экспериментальная функция сложности алгоритма принимает следующий ряд значений:

$$an \qquad an\log_2 n \qquad an^2 \qquad an^3 \qquad ae^n \qquad an!$$

Экспериментальная функция

Коэффициент a определяет истинность экспериментальной функции

(1<= a <=2).

При a = 1 значение экспериментальной функции совпадает со значением теоретической функции сложности:

а) допустим *О*э = an, тогда

an=54;

6a=54;

$a = \frac{54}{6}$ (не удовлетворяет условию 1<= a <=2);

б) допустим *О*э= $an\log_2 n$, тогда:

$an\log_2 n$ =54;

$a6\log_2 6$ =54;

$a = \frac{54}{6\log_2 6} = \frac{9}{\log_2 6}$ (не удовлетворяет условию 1<= a <=2);

в) допустим *О*э = an^2, тогда:

an^2 =54;

a36 = 54;

$a = \frac{54}{36} = 1{,}5$ (удовлетворяет условию 1<= a <=2).

Таким образом экспериментальная функция сложности имеет вид

$$O\text{э}(1{,}5\ n^2).$$

Найдем теоретическую функцию сложности

$$\lim_{n\to\infty}\frac{O\text{э}(n)}{Om(n)}=\text{const}\neq 0$$

$$\lim_{n\to\infty}\frac{1{,}5n^2}{X}=\text{const}\neq 0$$

$$\lim_{n\to\infty}\frac{1{,}5n^2}{n^2}=\text{const}\neq 0$$

Отсюда, теоретическая функция сложности алгоритма - $O(n^2)$.

Содержание

Учебное издание
Колдаев *Виктор Дмитриевич*
Лабораторный практикум по курсу «Алгоритмы и структуры данных». Часть 1
Редактор *Н.А. Кузнецова*. Технический редактор *Е.Н. Романова*. Корректор *Л.Г. Лосякова*. Верстка автора.
Подписано в печать с оригинал-макета . . 2019. Формат 60×84 1/16.
Печать офсетная. Бумага офсетная. Гарнитура Times New Roman.
Усл. печ. л. . уч.-изд. л. . Тираж 200 экз.
Заказ .
Отпечатано в типографии ИПК МИЭТ.
124498, г. Москва, Зеленоград, площадь Шокина, дом 1, МИЭТ