МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Гжельский государственный университет»** (ГГУ)

Колледж ГГУ

Специальность 09.02.07 Информационные системы и программирование

**Практическая работа**

по предмету «МАПО»

на тему «Обратное проектирование алгоритма»

ВЫПОЛНИЛ:

Студент группы ИСП-О-17

Белкин И.В.

ПРОВЕРИЛА:

Прокуронова А. Ю.

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

п. Электроизолятор

1. год

**Методы разработки алгоритмов**

Составление алгоритмов решения задач - это работа творческая. Нет универсального способа, позволяющего без особого труда составлять любые алгоритмы. К сожалению, такого способа не существует, ведь жизненные ситуации и задачи так разнообразны и непредсказуемы! Если бы дело обстояло иначе, появилась бы реальная возможность автоматизировать сам процесс алгоритмизации, поручив его некоторому исполнителю - вероятно, очень высокоинтеллектуальному компьютеру.

Тем не менее, некоторые рекомендации, касающиеся методики разработки алгоритмов, можно дать.

При решении простых задач можно воспользоваться определенной схемой. Есть раздел математики, называемый вычислительной математикой, в котором накоплен многолетний (а порой и многовековой) опыт решения разных вычислительных задач. Нет необходимости разрабатывать заново те алгоритмы, которые уже созданы - надо только их изучить и практически применять при решении своих задач. Таковы, например, методы отыскания корней нелинейных уравнений, вычисления определенных интегралов, численного интегрирования дифференциальных уравнений, методы сортировки данных и многие другие.

В большинстве случаев та или иная задача может быть решена несколькими численными методами. Выбор конкретного численного метода решения задачи обычно производится по следующим критериям:

обеспечение оптимального времени решения задачи;

обеспечение оптимального использования имеющихся ресурсов (памяти);

обеспечение требуемой точности вычислений;

минимальные стоимостные затраты;

возможность использования стандартных подпрограмм.

При дальнейшей постановке задачи на ПК отыскивается наиболее рациональный способ решения задачи.

Однако, алгоритмы становятся все более и более сложными, соответственно растет трудность понимания того, как они работают. А еще труднее обнаружить и исправить в них ошибки или внести какие-то изменения. От 50 до 100% времени программист тратит на исправление и модификацию программ. В связи с этим индустрия программирования предлагает более систематичные подходы к программированию (а тем самым и к алгоритмизации задач в общем), т.е. предлагает методики, использование которых уменьшает вероятность ошибок в программах, упрощает их понимание и облегчает модификацию.

Структурное программирование - одна из популярных методик. Фундаментом структурного программирования является доказанная Бемом и Джекопини теорема о структурировании [15]. Эта теорема устанавливает, что как бы сложна ни была задача, блок-схема соответствующей программы (читай - "соответствующего алгоритма") всегда может быть представлена с использованием весьма ограниченного числа элементарных управляющих структур (последовательность, ветвление, цикл).

Главная идея доказательства этой теоремы состоит в преобразовании каждой части алгоритма в одну из трех основных структур или их комбинацию так, чтобы неструктурированная часть алгоритма уменьшилась. После достаточного числа таких преобразований оставшаяся неструктурированной часть либо исчезнет, либо станет ненужной. Доказывается, что в результате получится алгоритм, эквивалентный исходному и использующий лишь упомянутые управляющие структуры.

Цель структурного программирования - выбор структуры программы путем расчленения исходной задачи на подзадачи. Программы должны иметь простую структуру. Сложные, запутанные программы, как правило, являются неработоспособными, а их тестирование требует больших затрат.

Разработка алгоритма, являясь четким логичным процессом, упрощается на каждом уровне шаг за шагом. Затем в процессе задействуется следующий метод алгоритмизации - метод пошагового уточнения (совершенствования). Сначала задача рассматривается в целом, выделяются наиболее крупные ее части. Алгоритм, указывающий порядок выполнения этих частей, описывается в структурированной форме, не вдаваясь в мелкие детали. Затем от общей структуры переходят к описанию отдельных частей. Таким образом, разработка алгоритма состоит из последовательности шагов в направлении уточнения алгоритма.

Дальнейшим развитием, расширением структурного программирования является модульное программирование, идея которого состоит в том, что алгоритм может быть представлен в виде системы, совокупности отдельных модулей. Каждый модуль рассматривается как самостоятельная, относительно независимая программа, которая может содержать набор данных и функций, доступных только из этого модуля.

Модульное программирование позволяет значительно ускорить процесс засчет привлечения к работе нескольких специалистов сразу, доверив каждому разработку отдельного модуля. Кроме того, модульное программирование предполагает возможность использования заранее разработанных стандартных программ (т.н. библиотек стандартных подпрограмм).

На этапе проектирования алгоритма решения сложной задачи, состоящей из нескольких подзадач, используют два подхода: нисходящий и восходящий.

При нисходящем проектировании вначале проектируются функции управляющей программы - драйвера. Затем более подробно представляют каждую подзадачу и разрабатывают другие модули. При нисходящем проектировании на каждом шаге функционирование модуля описывается с помощью ссылок на последующие, более подробные шаги.

При восходящем проектировании вначале проектируют программы низшего уровня, иногда в виде самостоятельных подпрограмм. Затем на каждом шаге разрабатываются модули более высокого уровня.

Существует также несколько общих алгоритмических методов решения очень сложных задач. Более подробно ознакомиться с некоторыми из них можно в [7] (методы частных целей, подъема и отрабатывания назад, метод ветвей и границ и др.).

**Жадные алгоритмы**

Для многих оптимизационных задач есть более простые и быстрые алгоритмы, чем динамическое программирование. В этой главе мы рассматриваем задачи, которые можно решать с помощью жадных алгоритмов (greedyalgorithms). Такой алгоритм делает на каждом шаге локально оптимальный выбор, - в надежде, что итоговое решение также окажется оптимальным. Это не всегда так – но для многих задач такие алгоритмы действительно дают оптимум. Наш первый пример – простая, но не вполне тривиальная задача о выборе заявок . Далее мы обсуждаем, для каких задач годятся жадные алгоритмы.

**Задача о выборе заявок**

Пусть даны n заявок на проведение занятий в одной и той же аудитории. Два разных занятия не могут перекрываться по времени. В каждой заявке указаны начало и коней занятия (si и fi для i-й заявки). Разные заявки могут пересекаться, и тогда можно удовлетворить только одну из них. Мы отождествляем каждую заявку с промежутком [si ,fi ), так что конец одного занятия может совпадать с началом другого, и это не считается пересечением.

Формально говоря, заявки с номерами i и j совместны (compatible), если интервалы [si , fi ) и [sj ,fj ) не пересекаются (иными словами, если

fi £sj или fj £si ). Задача о выборе заявок ( activity-selectionproblem) состоит в том, чтобы набрать максимальное количество совместных друг с другом заявок.

Жадный алгоритм работает следующим образом. Мы предполагаем, что заявки упорядочены в порядке возрастания времени окончания:

f1£f2£ … £fn (1)

Если это не так, то можно отсортировать их за время O(nlogn); заявки с одинаковым временем конца располагаем в произвольном порядке.

Тогда алгоритм выглядит так (f и s – массивы):

Greedy-Activity- Selector (s, f)

1 n ¬ length [s]

2 A ¬ {1}

3 j ¬ 1

4 for i ¬ 2 to *n*

do if si ³ fj

then A ¬ A È{i}

j ¬ i

return A

Работа этого алгоритма показана на рис.1. Множество F состоит из номеров выбранных заявок, j – номер последней из них; при этом

Fj = max {fk : kÎA}, (17.2)

поскольку заявки отсортированы по возрастанию времени окончания. Вначале А содержит заявку номер 1, и j=1 (строки 2-3). Далее ( цикл в строках 4-7) ищется заявка, начинающаяся не раньше окончания заявки номер j. Если таковая найдена, она включается в множество Ф и переменной j присваивается ее номер (строки 6-7).

Алгоритм Greedy-Activity- Selector требует всего лишь q (n) шагов ( не считая предварительной сортировки). Как и подобает жадному алгоритму, на каждом шаге он делает выбор так, чтобы остающееся свободным время было максимально.

**Правильность алгоритма**

Не для всех задач жадный алгоритм дает оптимальное решение, но для нашей дает. Убедимся в этом.

Теорема 1. *Алгоритм* Greedy-Activity- Selector дает*набор из наибольшего возможного количества совместных заявок.*

*Доказательство.* Напомним, что заявки отсортированы по возрастанию времени окончания. Прежде всего докажем, что существует оптимальное решение задачи о выборе заявок, содержащее заявку номер1 (с самым ранним временем окончания). В самом деле, если в каком-то оптимальном множестве заявок заявка номер 1 не содержится, то можно заменить в нем заявку с самым ранним временем окончания не заявку номер 1, что не повредит совместности заявок ( ибо заявка номер 1 кончается еще раньше, чем прежняя, и не с чем пересечься не может) и не изменит их общего количества. Стало быть, можно искать оптимальное решение, начинающееся с жадного выбора.

После того, как мы договорились рассматривать только наборы, содержащие заявку номер 1, все несовместные с ней заявки можно выкинуть, и задача сводится к выбору оптимального набора заявок из множества оставшихся заявок (совместных с заявкой номер 1). Другими словами, мы свели задачу к аналогичной задаче с меньшим числом заявок. Рассуждая по индукции, получаем, что, делая на каждом шаге жадный выбор, мы придем к оптимальному решению.

**Когда применим жадный алгоритм?**

Как узнать, даст ли жадный алгоритм оптимум применительно к данной задаче? Общих рецептов тут нет, но существует две особенности, характерные для задач, решаемых жадными алгоритмами. Это принцип жадного выбора и свойство оптимальности для подзадач.

**Принцип жадного выбора**

Говорят, что к оптимизационной задаче применим принцип жадного выбора (greedy-choiceproperty), если последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение. Различие между жадными алгоритмами и динамическим программированием можно пояснить так: на каждом шаге жадный алгоритм берет "самый жирный кусок", а потом уже пытается сделать наилучший выбор среди оставшихся, каковы бы они ни были; алгоритм динамического программирования принимает решение, просчитав заранее последствия для всех вариантов.

Как доказать, что жадный алгоритм дает оптимальное решение? Это не всегда тривиально, но в типичном случае такое доказательство следует схеме, использованной в доказательстве теоремы 1. Сначала мы доказываем, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не худшее первого. Затем показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной, и рассуждение завершается по индукции.

**Оптимальность для подзадач**

Говоря иными словами, решаемые с помощью жадных алгоритмов задачи обладают свойством оптимальности для подзадач (haveoptimalsubstructure): оптимальное решение всей задачи содержит в себе оптимальные решения подзадач. (С этим свойством мы уже встречались, говоря о динамическом программировании). Например, при доказательстве теоремы 1 мы видели, что если А – оптимальный набор заявок, содержащий заявку номер 1, то А’=A \{1} – оптимальный набор заявок для меньшего множества заявок S’, состоящего из тех заявок, для которых si³f1 .

**Жадный алгоритм или динамическое программирование?**

И жадные алгоритмы, и динамическое программирование основываются на свойстве оптимальности для подзадач, поэтому может возникнуть искушение применить динамическое программирование в ситуации, где хватило бы жадного алгоритма, или, напротив, применить жадный алгоритм к задаче, в которой он не даст оптимума. Мы проиллюстрируем возможные ловушки на примере двух вариантов классической оптимизационной задачи.

Дискретная задача о рюкзаке (0-1 knapsackproblem) состоит в следующем. Пусть вор пробрался на склад, на котором хранится *n* вещей. Вещь номер *i* стоит *vi*долларов и весит *wi*килограммов (*vi*и *wi*- целые числа). Вор хочет украсть товара на максимальную сумму, причем максимальный вес, который он может унести в рюкзаке, равен W ( число W тоже целое). Что он должен положить в рюкзак?

Непрерывная задача о рюкзаке ( fractionalknapsackproblem)) отличается от дискретной тем, что вор может дробить краденые товары на части и укладывать в рюкзак эти части, а не обязательно вещи целиком (если в дискретной задаче вор имеет дело с золотыми слитками, то в непрерывной – с золотым песком).

Обе задачи о рюкзаке обладают свойством оптимальности для подзадач. В самом деле, рассмотрим дискретную задачу. Вынув вещь номер *j* из оптимально загруженного рюкзака, получим решение задачи о рюкзаке с максимальным весом W – *wj*и набором из *n* -1 вещи ( все вещи, кроме *j* -й). Аналогичное рассуждениеподходит и для непрерывной задачи: вынув из оптимально загруженного рюкзака, в котором лежит *w* килограммов товара номер *j* , весь этот товар, получим оптимальное решение непрерывной задачи, в которой максимальный вес равен W-*w* (вместо W), а количество*j* -го товара равно *wj -* *w* ( вместо *wj*).

Хотя две задачи о рюкзаке и похожи, жадный алгоритм дает оптимум в непрерывной задаче о рюкзаке и не дает в дискретной. В самом деле, решение непрерывной задачи о рюкзаке с помощью жадного алгоритма выглядит так. Вычислим цены (в расчете на килограмм) всех товаров (цена товара номер *i* равна *vi /* *wi*). Сначала вор берет по максимуму самого дорогого товара; если весь этот товар кончился, а рюкзак не заполнен, вор берет следующий по цене товар, затес следующий, и так далее, пока не наберет вес W. Поскольку товары надо предварительно отсортировать по ценам, на что уйдет время О(*n* *logn* ), время работы описанного алгоритма будет О(*n* *logn* ).

Чтобы убедиться в том, что аналогичный жадный алгоритм не обязан давать оптимум в дискретной задаче о рюкзаке, взгляните на рис. 2(а). Грузоподъемность рюкзака 50кг, на складе имеются три вещи, весящие 10, 20 и 30кг и стоящие 60, 100 и 120 долларов соответственно. Цена их в расчете на единицу веса равна 6,5 и 4. Жадный алгоритм для начала положит в рюкзак вещь номер 1; однако оптимальное решение включает предметы номер 2 и 3.

Для непрерывной задачи с теми же исходными данными жадный алгоритм, предписывающий начать с товара номер 1, дает оптимальное решение (рис. 2(в)). В дискретной задаче такая стратегия не срабатывает: положив в рюкзак предмет номер 1, вор лишается возможности заполнить рюкзак «под завязку», а пустое место в рюкзаке снижает цену наворованного в расчете на единицу веса. Здесь, чтобы решить, класть ли данную вещь в рюкзак, надо сравнить решения двух подзадач: когда данная вещь заведомо лежит в рюкзаке и когда этой вещи в рюкзаке заведомо нет. Тем самым дискретная задача о рюкзаке порождает множество перекрывающихся подзадач – типичный признак того, что может пригодиться динамическое программирование. И действительно, к дискретной задаче о рюкзаке оно применимо .

Рис. 2. В дискретной задаче о рюкзаке жадная стратегия может не сработать. (а) Вор должен выбрать две вещи из трех с тем, чтобы их суммарный вес не превысил 50кг. (б) Оптимальный выбор – вторая и третья вещи; если положить в рюкзак первую, то выбор оптимальным не будет, хотя именно она дороже всех в расчете на единицу веса. (в) Для непрерывной задачи о рюкзаке с теми же исходными данными выбор товаров в порядке убывания цены на единицу веса будет оптимален.

**Заключение.**

В результате исследования литературы по вопросу оптимизации задач, мы отдали предпочтение использованию «жадных алгоритмов», которые дают нам следующие возможности: