ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 10
ЗАДАЧА О РЮКЗАКЕ

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо решить задачу о двух камнях с timus, воспользовавшись решением задачи о рюкзаке: У вас есть несколько камней известного веса w1, …, wn. Напишите программу, которая распределит камни в две кучи так, что разность весов этих двух куч будет минимальной.

Для решения этой задачи необходимо воспользоваться решением задачи о рюкзаке и свести нашу задачу к решению этой, нам необходимо сложить максимальный по весу набор камней в рюкзак, суммарный вес которого не может превышать S / 2, где S это суммарный вес всей кучи. Ответ на нашу задачу будет равен S - 2M, так как итоговый результат упаковывания наших камней является наилучшим, то лучше разделить камни на 2 кучи мы не сможем, а значит разница между ними, это то что по сути осталось после упаковки 2х идеальных рюкзаков.

После решения задачи требуется проверить правильность решения загрузив свое решение на проверку на сайте источнике задачи.

Описание метода/модели.

Разделяй и Властвуй

Все описанные ниже алгоритмы так или иначе укладываются в идею разделяй и властвуй. Суть этой идеи в том, чтобы разбить одну огромную и сложную проблему, придумывания решения которой само по себе занимает много времени, на некоторое количество более мелки, решение которых как в вопрос его создания и реализации, так и в вопросе исполнения будет значительно быстрее.

Ярким примером может служить расчет алгебраических выражений,

- мы можем раздробить выражение любой сложности на группы меньшей сложности
- повторить этот процесс несколько раз, до тех пор, пока не достигнем приемлемой сложности
- после, вычислив самые простые выражения мы сможем упростить более сложные и вычислить уже их
- Поднимаясь в обратную сторону мы сократим итоговую сложность первого выражения до самой простой операции между двумя числами
- Таким образом мы получили нужное решение

Эвристические алгоритмы

Что же можно понимать под подходами к построению алгоритмов? Основная масса алгоритмов строиться на некоторой математической обоснованности, т.е. имеется какая-то задача, и эта задача решается сначала аналитически, доказывается и показывается ее правильность, и в итоге все это обличается и описывается каким-либо программным кодом, который затем постоянно решает эту задачу на основании уже придуманного теоретического решения.

Все ли задачи можно решить таким способом?

Что же можно понимать под подходами к построению алгоритмов? Основная масса алгоритмов строиться на некоторой математической обоснованности, т.е. имеется какая-то задача, и эта задача решается сначала аналитически, доказывается и показывается ее правильность, и в итоге все это обличается и описывается каким-либо программным кодом, который затем постоянно решает эту задачу на основании уже придуманного теоретического решения.

Все ли задачи можно решить таким способом?

Что важно учитывать, когда работаешь с эвристическими алгоритмами?

Проще говоря, эвристика — это не полностью математически обоснованный (или даже «не совсем корректный»), но при этом практически полезный алгоритм.

Важно понимать, что эвристика, в отличие от корректного алгоритма решения задачи, обладает следующими особенностями.

Она не гарантирует нахождение лучшего решения.

Она не гарантирует нахождение решения, даже если оно заведомо существует (возможен «пропуск цели»).

Она может дать неверное решение в некоторых случаях.

Примеры эвристических алгоритмов:

- Метод Монте-Карло
- Локальный поиск
- Имитация отжига

Жадные алгоритмы

Жадные алгоритмы сами по себе частично относятся к эвристическим, однако суммарная идея их применения немного отличается.

Жадным алгоритмом называется алгоритм, который в каждой локальной группе решений принимает наиболее оптимальное решение, переходи в него и повторяет процедура далее. Все это строится на идеи о том, что если принимать множество оптимальных решений, то конечное решение так же будет оптимальным.

Такие методы, в случае если имеется доказанный факт оптимальности глобального решения, полученного на основании оптимальных локальных (что отличает такие методы от чисто

эвристических) часто является более применимым чем чисто эвристические методы или динамическое программирование.

Говорят, что к оптимизационной задаче применим принцип жадного выбора, если последовательность локально оптимальных выборов даёт глобально оптимальное решение. В типичном случае доказательство оптимальности следует такой схеме:

- Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
- Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
- Рассуждение завершается по индукции.

Примеры жадных алгоритмов

- Алгоритм Хаффмана (адаптивный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью).
- Алгоритм Крускала (поиск островного дерева минимального веса в графе).
- Алгоритм Прима (поиск островного дерева минимального веса в связном графе).

Задачи, в которых жадные алгоритмы не дают оптимального решения

Для ряда задач, относящихся к классу NP, жадные алгоритмы не дают оптимального решения. К ним относятся:

- задача коммивояжера;
- задача минимальной раскраски графа;
- задача разбиения графа на подграфы;
- задача выделения максимальной клики;
- задачи, связанные с составлением расписаний.

Тем не менее, в ряде задач жадные алгоритмы дают неплохие приближённые решения.

Динамическое программирование

Динамическое программирование чем-то похоже на жадные алгоритмы, в частности тем, что, как и в случае с жадным алгоритмом здесь оперируют подзадачами и их решением.

Суммарно идея динамического программирования заключается в том, что мы разбиваем задачу на малые подзадачи, спускаясь вниз, далее, находим самую минимальную подзадачу, решаем, ее, затем, имея ее решение, мы создаем подзадачи, которые включают в свое решение решенную более меньшую подзадачу, снова получаем их решения, сохраняем их, и движемся далее вверх. Основной вопрос в том, дает ли рекурсивный алгоритм одинаковый ответ при одинаковой подзадаче?

Рис.1.Схема динамического программирования

По итогу динамическое программирование требует следующего для решения какой либо задачи:

- перекрывающиеся подзадачи;
- оптимальная подструктура;
- возможность запоминания решения часто встречающихся подзадач.

Словами общую идею построения алгоритма можно описать следующим образом:

- Разбиение задачи на подзадачи меньшего размера.
- Нахождение оптимального решения подзадач рекурсивно, проделывая такой же трехшаговый алгоритм.
- Использование полученного решения подзадач для конструирования решения исходной задачи.

Есть 2 варианта динамического программирования:

- нисходящее динамическое программирование: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений уже решенных подзадач.
- восходящее динамическое программирование: все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи. Этот способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Пример

Очень удобно проиллюстрировать динамический алгоритм на поиске чисел Фибоначчи. Часто во многих книгах по программированию числа Фибоначчи используются для иллюстрирования рекурсии и рекурсивных алгоритмов. Однако, затем многие программисты узнают, что это определенно неоптимальное использование рекурсии.

Как можно заметить алгоритм поиска чисел Фибоначчи однотипен, для каждого следующего числа нужно знать 2 предыдущих, и так что называется до победного. При этом строго известны первые 2 числа. Если решать эту задачу простой рекурсией, то результат будет плачевным, чем больше будет число, тем больше будет однотипных операций. Однако, замечая, что подзадачи поиска предыдущего числа всегда одинаковы, и по сути единожды найденное число можно затем использовать всякий раз когда нужно вычислить следующее, мы можем сохранять все числа в некоторый массив и использовать их, если нам нужно вычислить новое число, что позволит превратить наш алгоритм из рекурсивного в итеративный.

По сути, решаю эту задачу, мы сначала обратили внимание на закономерность и однотипность действий, поняли, что для каждого следующего числа эти действия буду давать такой же результат, и сама задача сводиться к нахождению предыдущих значений.

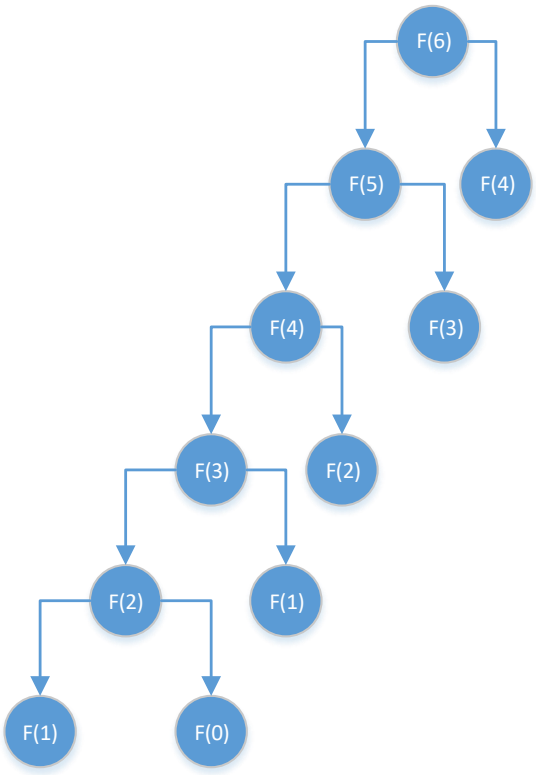

Рис.2.Итеративное решение поиска чисел Фибоначчи

Задача о рюкзаке

Задача о рюкзаке (англ. Knapsack problem) — дано N предметов, Ni предмет имеет массу Wi>0 стоимость Pi>0. Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W (вместимость рюкзака), а суммарная стоимость была максимальна.

Метод динамического программирования

Пусть A(k,s) есть максимальная стоимость предметов, которые можно уложить в рюкзак вместимости s, если можно использовать только первые k предметов, то есть {n1,n2,…,nk}, назовем этот набор допустимых предметов для A(k,s).

A(k,0)=0

A(0,s)=0

Найдем A(k,s). Возможны 2 варианта:

Если предмет k не попал в рюкзак. Тогда A(k,s) равно максимальной стоимости рюкзака с такой же вместимостью и набором допустимых предметов {n1,n2,…,nk−1}, то есть A(k,s)=A(k−1,s)

Если k попал в рюкзак. Тогда A(k,s) равно максимальной стоимости рюкзака, где вес суменьшаем на вес k-ого предмета и набор допустимых предметов {n1,n2,…,nk−1} плюс стоимость k, то есть A(k−1,s−wk)+pk

A(k,s)={A(k−1,s) , bk=0

{A(k−1,s−wk)+pkbk=1

То есть: A(k,s)=max(A(k−1,s),A(k−1,s−wk)+pk) =max

Стоимость искомого набора равна A(N,W) так как нужно найти максимальную стоимость рюкзака, где все предметы допустимы и вместимость рюкзака W.

Восстановим набор предметов, входящих в рюкзак

Будем определять, входит ли Ni предмет в искомый набор. Начинаем с элемента A(i,w), где i=N, w=W. Для этого сравниваем A(i,w) со следующими значениями:

Максимальная стоимость рюкзака с такой же вместимостью и набором допустимых предметов {n1,n2,…,ni−1}, то есть A(i−1,w)

Максимальная стоимость рюкзака с вместимостью на wi меньше и набором допустимых предметов {n1,n2,…,ni−1} плюс стоимость Pi, то есть A(i−1,w−wi)+Pi.

Заметим, что при построении A мы выбирали максимум из этих значений и записывали в A(i,w). Тогда будем сравнивать A(i,w) с A(i−1,w), если равны, тогда Ni не входит в искомый набор, иначе входит.

Метод динамического программирование всё равно не позволяет решать задачу за полиномиальное время, потому что его сложность зависит от максимального веса. Задача о ранце (или задача о рюкзаке) — одна из NP-полных задач комбинаторной оптимизации.

Реализация

```
for i = 0 to w
A[0][i] = 0
for i = 0 to n
A[i][0] = 0                                  //Первые элементы приравниваем к 0
for k = 1 to n
        for s = 1 to w                              //Перебираем для каждого k все вместимости
                if s >= w[k]                              //Если текущий предмет вмещается в рюкзак
                        A[k][s] = max(A[k - 1][s], A[k - 1][s - w[k]] + p[k]) //Выбираем класть его или нет
                 else
                        A[k][s] = A[k - 1][s]
```

Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

```
using System;
using System.Diagnostics;
using System.Globalization;
public class Bunc_Rocks
{
    /// <summary>
    /// Функция, реализующая решение задачи о рюкзаке
    /// </summary>
    /// <param name="W">Максимальный вес</param>
    /// <param name="weight">массив со значениями по весу</param>
    /// <param name="n">колиество предметов</param>
    /// <returns>Идеальное заполнение рюкзака</returns>
    static int KnapSack(int W, int[] weight, int n)
    {
        int[] dp = new int[W + 1];
        for (int i = 1; i < n + 1; i++)
        {
            for (int w = W; w >= 0; w--)
            {
                if (weight[i - 1] <= w)
                    dp[w]= Math.Max(dp[w], dp[w - weight[i - 1]] + weight[i - 1]);
            }
        }
        return dp[W];
    }
    /// <summary>
    /// Функция, решающая задачу о двух кучах камней
    /// </summary>
    private static void Main()
    {
            while (true)
            {
                Console.Write("Для Тимуса - 1; Для Лабы - 2: ");
                int chosen_number = Convert.ToInt32(Console.ReadLine());
                if (chosen_number == 1) TestTimus();
                if (chosen_number == 2) TestLaba();
                else Console.WriteLine("Выберите имеющийся вариант");
            }
    }
    /// <summary>
    /// Функция для решения задачи из тимуса
    /// </summary>
    public static void TestTimus()
    {
        double W_temp = 0;
        int N = Convert.ToInt32(Console.ReadLine());
        string str2 = Console.ReadLine();
        string[] w_2 = str2.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
        int[] weights = new int[N];
        for (int i = 0; i < N; i++) { weights[i] = Convert.ToInt32(w_2[i]); W_temp += weights[i]; }
        int W = (int)Math.Floor(W_temp / 2);

        Console.WriteLine(W_temp - 2 * KnapSack(W, weights, N));
    }
    /// <summary>
    /// Функция с рандомной генерацией значений
    /// </summary>
    public static void TestLaba()
    {
        Random rnd = new Random();
        double W_temp = 0;
        int N = rnd.Next(1,21);

        int[] weights = new int[N];
```

```
    GetRandomArr(ref weights, 1, 10000);
    for (int i = 0; i < N; i++) { W_temp += weights[i]; }
    int W = (int)Math.Floor(W_temp / 2);

    Console.WriteLine(W_temp - 2 * KnapSack(W, weights, N));
}
/// <summary>
/// Заполнение массива рандомными неповторяющими числами в диапазоне
/// </summary>
/// <param name="arr">массив, который требуется заполнить</param>
/// <param name="min">От</param>
/// <param name="max">До</param>
public static void GetRandomArr(ref int[] arr, int min, int max)
{
    Random random = new Random();
    for (int i = 0; i < arr.Length; i++)
    {
        var num = random.Next(min, max+1);

        if (arr.Contains(num))
        {
            i--;
        }
        else
        {
            arr[i] = num;
        }
    }
}
 }
```

Реализация:

1) Задача была решена с применением решения задачи о рюкзаках, суть которой была в том, чтобы зная максимальный вес, вес каждого отдельного предмета и их цену, найти такую комбинацию предметов, которые бы, по итогу, стоили бы больше всего. Это было реализовано в методе KnapSack
2) Метод KnapSack принимает общий вес W, вес каждого отдельного предмета weight и количество этих предметов N, для стандартной задачи о рюкзаке мне бы еще пришлось подавать цену каждого отдельного предмета, но здесь её будет занимать вес. Обычная реализация задачи о рюкзаке предполагала бы инициализацию двумерного массива N+1, W+1 по k и s соответственно, где по горизонтали бы располагались значения по весу, а по вертикале индекс предмета добавляемого. А далее бы проходились по всем предметам и если бы их можно было бы вставить по весу, то мы бы сравнивали, что ценнее, оставить всё как было на прошлом шаге или добавить на это место новый элемент, и так до последнего. Если элемент по весу не подходит, оставляем всё как было (if (weight[i - 1] <= w)). После прохождения по всему массиву, берем в качестве ответа его точку [N,W], она будет результатом сложения всех предметов в рюкзаке. Этот способ хоть и быстрый по времени, но неэффективный по памяти, так как массиву требуется держать множество значений, которые на дальнейших шагах уже будут и не нужны, что в случае больших W и N может много стоить.
3) Поэтому у меня инициализируется одномерный массив, длинною с общий вес. По сути, мне всего-то нужно в памяти держать значения, найденные на предыдущем ходу на [w - weight[i - 1]]. Это можно реализовать, идя по s в обратном порядке, определяя сначала последний элемент, а потом остальные, преобразуя постепенно массив.
4) Задача о куче камней преобразуется в эту таким образом: считаем, что передаваемый в KnapSack W будет не превышать вес всех камней деленный на две, вес и цена каждого камня будут равными друг другу значениями, так как нам дан только вес и для решения ищется разница по нему. После получения идеальной кучи камней, вычитаем её до множенное на два значение из общего веса кучи, т.е из двух идеального упакованных рюкзаков, результат чего и является нашей минимальной разностью.
5) Метод GetRandArr возвращает сгенерированный массив из рандомных чисел для тестирования.
6) Методы TestLaba и TestTimus предназначены для задания из лабы и тимуса соответственно

Результаты:

1) Код был загружен и проверен на timus

Последние попытки

Автор: Антон • Задача: Куча камней

ID	Дата	Автор	Задача	Язык	Результат проверки	№ теста	Время работы	Выделено памяти
10254143	17:28:56 23 апр 2023	Антон	1005. Куча камней	Visual C# 2019	Accepted		0.125	5 052 КБ

Показывать по 10 | 30 | 100 строк на странице • Обновлять каждые 15 | 60 | 240 секунд | не обновлять

Рис.3.Тимус

Заключение.

Динамическое программирование и в частности задача о рюкзаке в данном случае оказались эффективным методом для решения задачи. Но классическое итеративное решение неэффективно по времени, так как на больших значениях весов или количестве предметов выделяется много памяти на хранение всех позиций в массиве, многие из которых нам на дальнейших шагах не будут нужны, поэтому данный алгоритм можно упростить, сведя к одномерному массиву. Или решить рекурсивно, что будет легче для памяти, но хуже по времени.