Оглавление

Введение

Классическая задача о рюкзаке (о загрузке) известна очень давно, ниже приведена ее формализация. Пусть есть N разных предметов, каждый предмет имеет вес w_i и полезность p_i , так же имеется максимальный вес W, который можно положить в рюкзак. Требуется собрать такой набор предметов P, чтобы полезность их была наибольшей, а суммарный вес не превышал W.[6]. Конечно, никто не собирается писать программу, чтобы наилучшим образом загрузить рюкзак, отправляясь в поход или в путешествие, тут все слишком просто, и никто не задумывается об этом, но существует и более широкое применение.

Задача о загрузке (задача о рюкзаке) и различные её модификации широко применяются на практике в прикладной математике, криптографии, экономике, логистике, для нахождения решения оптимальной загрузки различных транспортных средств: самолетов, кораблей, железнодорожных вагонов и т.д.

Рассматриваемая нами задача является NP – полной, то есть для нее не существует полиномиального алгоритма , решающего её за разумное время, в этом и есть проблема. Либо мы выбираем быстрый алгоритм, но он как известно не всегда решает задачу наилучшим образом, либо выбираем точный, который опять же не является работоспособным для больших значений. Существует несколько модификаций задачи.

1. Каждый предмет можно брать только один раз.
2. Каждый предмет можно брать сколько угодно раз.
3. Каждый предмет можно брать определенное количество раз
4. На размер рюкзака имеется несколько ограничений.
5. Некоторые вещи имею больший приоритет, чем другие

Цель данной работы – выделить основные методы решения задачи о загрузке, классифицировать и сравнить эти методы.

Реализовать алгоритмы решения классической задачи о рюкзаке. Протестировать их и разбить их на две группы: точные и приближенные, сравнить по скорости решения, по точности. Определить в каких случаях следует использовать тот или иной подход к решению задачи.

Алгоритмы решения можно разделить на два типа: точные и приближенные. Точные: применение динамического программирования, полный перебор, метод ветвей и границ (сокращение полного перебора). Приближенные алгоритмы: Жадный алгоритм.

Глава 1 Задача о загрузке, рюкзаке, ранце. Постановка и NP-полнота задачи

1.1 Постановка задачи о рюкзаке

Задача о ранце – одна из задач комбинаторной оптимизации. Классическая задача о ранце известна очень давно. Вот её постановка: Имеется набор из N предметов, каждый предмет имеет массу Wi и полезность Vi, i=(1,2..N), требуется собрать набор с максимальной полезностью таким образом, чтобы он имел вес не больше W, где W – вместимость ранца. Традиционно полагают что W_i , V_i , W, P – целые неотрицательные числа, но встречаются и другие постановки, условия в которых могут отличаться.[6] Возможны следующие вариации задачи:

Каждый предмет можно брать только один раз. Формализуем. Пусть задано конечное множество предметов $Q = \{q_1, q_2, .., q_n\}$, для каждого $q \in Q$, определена стоимость p_i и вес w_i , тогда нужно максимизировать $\sum_{i=1}^{N} p_i * x_i$, при ограничениях $\sum_{i=1}^{N} w_i * x_i \leq W$, где W-вместимость ранца, а x_i=1, если предмет взят для загрузки и x_i=0 если не взят. Если на размер рюкзака имеется только одно ограничение, то задача называется одномерной, в противном случае – многомерной.

Каждый предмет можно брать m раз. Формализация аналогична, разница лишь в том, что x_i принимает значения на интервале (0..m).

Каждый предмет можно брать неограниченное количество раз. Очевидно, что x_i лежит в диапазоне (0..[W/w_i]) квадратные скобочки означают целую часть числа. [6]

Если же значения весов и цен предметов не целые числа, такая задача будет называться непрерывной задачей о рюкзаке, если же числа целые, то соответственно дискретной. Например, если мы имеем дело с золотыми слитками, мы не можем их делить – это дискретная задача, а если с золотым песком, то это непрерывная задача о рюкзаке.

1.2 NP – полнота задачи

Большинство используемых алгоритмов имеют полиномиальное время работы, если размер входных данных – n, то время их работы в худшем случае оценивается как $O(n^k)$ где k это константа. Но встречаются задачи, которые нельзя разрешить за полиномиальное время. Это класс NP - полных задач. Некоторые задачи этого класса на первый взгляд аналогичны задачам разрешимым за полиномиальное время, но это далеко не так. Задача называется NP - полной, если для нее не существует полиномиального алгоритма.[3] Алгоритм называется полиномиальным, если его сложность O(N) в худшем случае ограничена сверху некоторым многочленом (полиномом) от N. Такие задачи возникают очень часто в различных областях: в булевой логике, в теории графов, теории множеств, кодировании информации, в алгебре, в биологии, физике, экономике, теории автоматов и языков. Считается что NP - полные задачи очень трудноразрешимы, а так же, что если хотя бы для одной из них удастся найти полиномиальный алгоритм, то такой алгоритм будет существовать для любой задачи из этого класса. Над поиском полиномиальных алгоритмов к таким задачам трудились многие ученые, и все же и все же при таком разнообразии NP - полных задач, ни для одной из них до сих пор не найдено полиномиального алгоритма.[10]. Из всего вышесказанного следует, что если известна NP - полнота задачи, то лучше потратить время на построение приближенного алгоритма, чем пытаться построить полиномиальный, или же, если это позволяют условия, использовать алгоритмы с экспоненциальной сложностью работы

Глава 2 Методы решения задачи о рюкзаке

2.1 Классификация методов

На практике очень часто возникают NP-полные задачи, задач о рюкзаке – одна из них . Конечно надежд, на то что для них найдется полиномиальный алгоритм практически нет, но из этого не следует что с задачей нельзя ничего сделать. Во первых, очень часто удается построить полиномиальный алгоритм для NP – полной задачи, конечно он даст приближенное, а не точное решение, но зато будет работать за реальное время. Во вторых, данные могут быть таковы, что экспоненциальный алгоритм, например переборный сможет работать на них разумное время. К точным методам относятся: Полный перебор, метод ветвей и границ, ДП – программирование. К приближенным: Жадные алгоритмы. Полный перебор – перебор всех вариантов (всех состояний) –малоэффективный, но точный метод. Метод ветвей и границ – по сути сокращение полного перебора с отсечением заведомо “плохих” решений. ДП – алгоритм, основанный на принципе оптимальности Беллмана. Жадный алгоритм – основан на нахождении относительно хорошего и “дешевого” решения.

2.2 Динамическое программирование

В основе метода динамического программирования лежит принцип оптимальности Беллмана:”Каково бы ни было состояние системы перед очередным шагом, надо выбирать управление на этом шаге так, чтобы выигрыш на этом шаге плюс оптимальный выигрыш на всех последующих шагах был оптимальным”. Проще говоря оптимальное решение на i шаге находится исходя из найденных ранее оптимальных решений на предшествующих шагах. Из этого следует, что для того чтобы найти оптимальное решение на последнем шаге надо сначала найти оптимальное

решения для первого, затем для второго и так далее пока не пройдем все шаги до последнего.

Имеется набор из N предметов. Пусть MaxW - объем рюкзака, P_i – стоимость i-го предмета, W_i – вес i-го предмета. Value [W, i] – максимальная сумма, которую надо найти. Суть метода динамического программирования – на каждом шаге по весу $1<W_i<W$ находим максимальную загрузку Value[W_i, i], для веса W_i. Допустим мы уже нашли Value[1..W, 1..i-1], то есть для веса меньше либо равного W и с предметами, взятыми из 1..N-1. Рассмотрим предмет N, если его вес W_N меньше W проверим стоит ли его брать.

Если его взять то вес станет W-W_i , тогда Value[W, i] = Value[W – W_i , i-1] + P_i (для Value[W – W_i , i-1]) решение уже найдено остается только прибавить P_i.

Если его не брать то вес останется тем же и Value[W , i] = Value[W – Wi , i-1]. =Из двух вариантов выбирается тот, который дает наибольший результат. Рассмотрим алгоритм подробнее.

Вместимость ранца = 5				
номер	i	1	2	3
веса предметов	W	1	3	2
цены предметов	P	6	12	10
1я итерация (Weight = 1)	Value			
		Weight	1	
i=1	I			
заносим p[1] в value[1, 1] для веса 1 и набора	1		6	
предметов 0..1 это оптимум	2			
		Weight	1	
	I			
	1		6	
предмет 2 не помещается в вес=1 значит value[1, 2]=value[1, 1]	2		6	
то же самое для value[1, 3]	3		6	
Следовательно оптимум для Value[1, 3]=6, то есть для данного набора предметов и веса рюкза=1.				

Рис 1.1

2я итерация								
	I	Weight	1	2				
предмет 1 помещается в рюкзак весом 2	1		6	6				
предмет 2 не помещается в рюкзак весом 2 значит там	2		6	6				
предмет 3 помещается в пустой рюкзак весом 2	3		6	10				
Следовательно оптимум для Value[2, 3]=10, то есть для данного набора предметов и веса рюкза=2.								
3я итерация								
	I	Weight	1	2	3			
предмет 1 помещается в ранец весом 2	1		6	6	6			
предмет 2 помещается в ранец весом 2 записываем в value[3, 2] P[2]=12	2		6	6	12			
предмет 3 помещается в ранец весом 3 вместе с предметом 1. value[3-(w[3]=2),(i=3)-1]+P[3]-16 (берем 1й и 3й предметы)	3		6	10	16			
Следовательно оптимум для Value[3, 3]=16. вес рюкза=3.								

Рис 1.2

4я итерация								
	I	Weight	1	2	3	4		
помещаем 1й предмет	1		6	6	6	6		
помещаем 2й предмет складываем его с value[1 , 1] (взяли 2й и первый предметы)	2		6	6	12	18		
если возьмем 3й предмет то его нужно сложить с value[2,2]=6 в сумме будет 16, что меньше чем value[4,	3		6	10	16	18		
Оптимум для value[4, 3]=18. вес ранца=4.								
5я итерация								
	I	Weight	1	2	3	4	5	
помещаем 1й предмет	1		6	6	6	6	6	
помещаем 2й предмет+ value[2, 1] в сумме 18 (взяли 2й и 1й предметы)	2		6	6	12	18	18	
помещаем 3й предмет в сумме с value[3, 2] = 22 (берем 2й и 3й предметы)	3		6	10	16	18	22	
Таким образом мы набрали максимальную стоимость value[5, 3] для данных предметов и веса рюкзака								

Рис 1.3

Динамическое программирование для задачи о рюкзаке дает точное решение, причем одновременно вычисляются решения для всех размеров рюкзака от 1 до MaxW, но какой ценой? Для хранения таблицы стоимости и запоминания того, брался каждый предмет или нет, требуется порядка O(N*MaxW) памяти, временная сложность равна O(N*MaxW) ;

Опишем основную логику решения: {Загружаем рюкзак если его вместимость = Weight} **for** Weight:=1 **to** MaxW **do begin**

for i:=1 **to** N **do** {берем предметы с 1 по N}

{если вес предмета больше Weight}

{или предыдущий набор лучше выбираемого}

if (W[i]>Weight) **or** (Value[Weight, i-1] >=

Value[Weight-W[i], i-1]+P[i]) **then begin**

{Тогда берем предыдущий набор}

Value[Weight, i]:=Value[Weight, i-1];

{говорим что вещь i не взята}

Take [Weight, i]:= false;

End

{иначе добавляем к предыдущему набору текущий

предмет}

Else begin

Value [Weight, i]:=Value [Weight - W[i], i-1]

+P[i];

{говорим что вещь i взята}

Take [Weight, i]:= true;

End;

End;

Как было сказано ранее, алгоритм динамического программирования для ‘рюкзака’ дает точное решение путем использования дополнительной памяти O(N*MaxW), временная сложность алгоритма так же будет порядка O(N*MaxW).

2.3 Полный перебор

Название метода говорит само за себя. Чтобы получить решение нужно перебрать все возможные варианты загрузки. Здесь мы будем рассматривать такую постановку задачи. В рюкзак загружаются предметы N различных типов (количество предметов каждого типа не ограничено), каждый предмет типа I имеет вес W_i и стоимость P_i , i=(1,2..N). Требуется определить максимальную стоимость груза вес, которого не превышает W. Очевидна простая рекурсивная реализация данного подхода Рис 1.4. Временная сложность данного алгоритма равна O(N!). Алгоритм имеет сложность факториал и может работать лишь с небольшими значениями N. С ростом N, число вариантов очень быстро растет, и задача становится практически неразрешимой методом полного перебора. На рис 1.5 показано дерево перебора, дерево имеет 4 уровня. В каждом кружочке показан вес предмета, корень дерева – нулевой вес, то есть когда рюкзак пуст. Первый предмет можно выбрать четырьмя способами, второй – тремя, третий – двумя, а дальше можем взять только один оставшийся предмет.

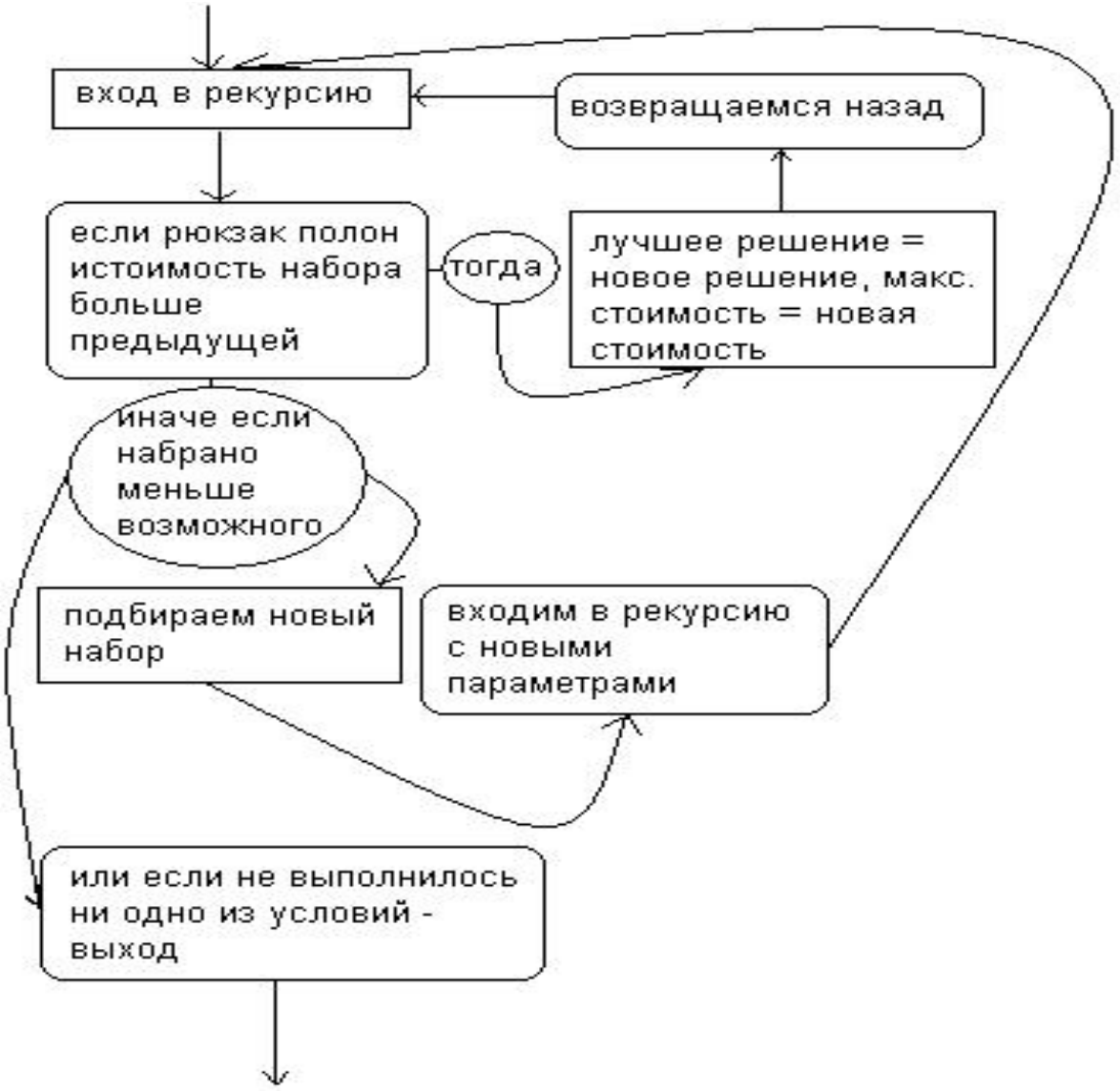

вход в рекурсию
возвращаемся назад
если рюкзак полон
истоимость набора
больше
предыдущей
тогда
лучшее решение =
новое решение, макс.
стоимость = новая
стоимость
иначе если
набрано
меньше
возможного
подбираем новый
набор
входим в рекурсию
с новыми
параметрами
или если не выполнилось
ни одно из условий -
выход

Рис 1.4

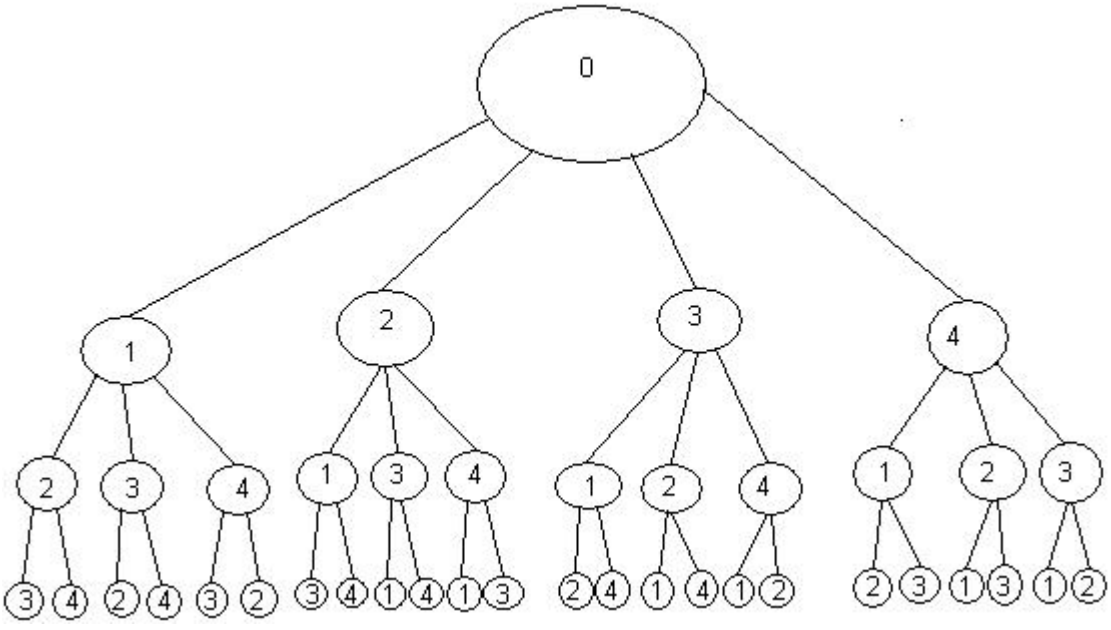

0
1
2
3
4
2 3 4
1 3 4
1 2 4
1 2 3
3 4 2 4 3 2
3 4 1 4 1 3
2 4 1 4 1 2
2 3 1 3 1 2

Рис 1.5

N - Количество предметов. Пусть MaxW - объем рюкзака, P_i – стоимость i-го предмета, W_i – вес i-го предмета.

{передаем Nab - номер набранной группы, OstW-вместимость, stoim-цена набранного (еще не набрали нисколько)}

```
Procedure Search(Nab, OstW, Stoim:integer);
var i:integer;
begin
{здесь OstW-вес который следует набрать из оставшихся. Stoim-стоимость текущего решения}
{Nab - набор предметов если наполнили рюкзак и набрали стоимость больше чем имеется, то считаем это новым решением}
if (Nab > N) and (Stoim > Max) then begin
{найдено решение}
BestP:=NowP;
Max:=Stoim;
End
{иначе если количество взятых <= обьема.
забиваем рюкзак дальше}
else if Nab<=N then
{иначе если набрано меньше чем влазит}
for i:=0 to OstW div W[Nab] do begin
{идем от 0 до ост. места}
NowP[Nab]:=i;
{берем предмет Nab пока есть место в ранце}
Search(Nab+1,OstW-i*W[Nab],Stoim+i*P[Nab]);
{после каждого взятия предмета увеличиваем
стоимость набора и уменьшаем место в рюкзаке
на вес предмета, так же увеличиваем количество
```

раз взятия предмета}

end;

2.4 Метод ветвей и границ

По существу данный метод - это вариация полного перебора, с исключениями заведомо не оптимальных решений. Для полного перебора можно построить дерево решений. Если у нас есть какое то оптимальное решение P, мы пытаемся улучшить его, но если на рассматриваемой в текущий момент ветви решение заведомо хуже чем P то следует остановить поиск и выбрать другую ветвь для рассмотрения. Например, на рис 1.5. есть ограничение на вес рюкзака W=5. Тогда используя метод ветвей и границ можно сократить дерево перебора до такого, рис 1.6. Видно сразу, что количество вариантов для перебора уменьшилось сразу. А именно осталось 8 вариантов исхода, вместо 24 ранее. Но не всегда получается отсеять достаточно много вариантов чтобы скорость работы была заметно увеличена, всегда можно подобрать такие входные данные, для которых метод ветвей и границ даст оценку по времени идентичную полному перебору.

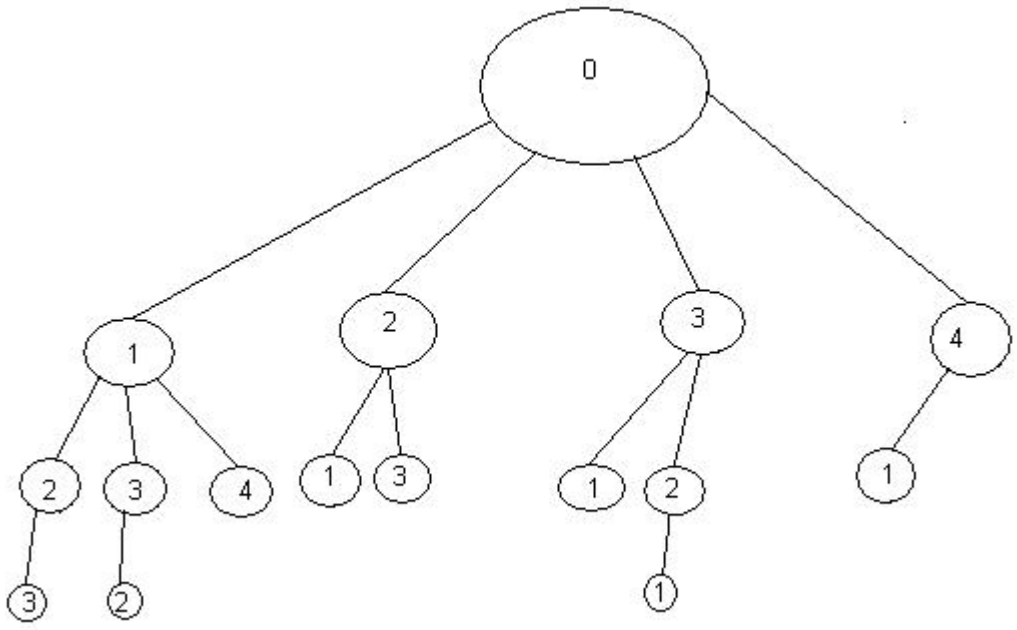


Рис 1.6

2.5 Жадный алгоритм

В случае применения жадного алгоритма поступаем так: сортируем предметы по убыванию стоимости единицы каждого. $P_i = \frac{W_i}{V_i}$, где Pi - относительная стоимость единицы предмета i, W_i - вес предмета i, V_i - стоимость предмета i. Всего N предметов. Пытаемся поместить в рюкзак все что помещается, и одновременно наиболее дорогое по параметру P. Оценим сложность метода. Для сортировки нам потребуется $O(N * \log(N))$ плюс проход по N предметам в цикле. Итого $O(N * \log(N)) + O(N)$ что в общем случае равно $O(N * \log(N))$. Скорость работы относительно других алгоритмов высока, но если посмотреть более внимательно, видно, что точное решение мы получим не всегда. Обратим внимание на следующую таблицу Таб1.1.

Номер предмета (i)	Вес предмета (кг)	Цена (У.е)	Относительная цена (У.е/кг)
1	10	60	6
2	20	100	5
3	30	120	4

Как видно предметы уже отсортированы. Пусть в рюкзак помещается 50кг, следуя алгоритму, берем первый предмет, затем второй, третий предмет уже не помещается. Таким образом, в рюкзаке у нас 30кг стоимостью 160у.е, оставшееся место 20кг. Но если бы мы взяли второй и третий предметы, общий вес поместился в рюкзак, и стоимость его была бы 220у.е. Жадный алгоритм не дает оптимального решения, поэтому он является приближенным алгоритмом.[7] Оказывается качество решения можно улучшить, если сравнить полученный результат с максимальным коэффициентом V_{max} ; $V_{max} = \mathrm{Max}(V_1 .. V_N)$. Предполагается, что все предметы не превосходят размера рюкзака, в противном случае их можно просто исключить из рассмотрения.[3]

Рассмотрим непрерывную задачу о ранце, условия для нее те же самые, отличие лишь в том, что мы можем взять часть предмета. То есть предметы можно делить. Пусть у нас есть тот же набор что и в таб. 1.1, тогда следуя жадному алгоритму, берем первый и второй предметы, полностью третий предмет не помещается т.к места осталось всего на 20кг, но мы можем брать части предметов, тогда возьмем $^2/_3$ веса третьего предмета, соответственно и $^2/_3$ его стоимости, таким образом мы нагрузили рюкзак полностью, стоимость груза стала равна 240у.е. Для непрерывной задачи о рюкзаке жадный алгоритм будет давать оптимальное решение.[7]

```
{обнуляем список взятых предметов} fillchar(Take, sizeof(Take), False);
i:= 0; {пока текущий вес набора + следующий предмет, который будет взят меньше предела вместимости}
While NowWeight+W[i+1] < MaxWeight do begin
inc(i);
{увеличиваем сумму цен на цену текущего предмета}
BestPrice:= BestPrice + P[i];
{увеличиваем сумму весов на вес тек. предмета}
NowWeight:= NowWeight + W[i];
{записываем что взяли этот предмет}
Take[i]:= true;
end;
```

2.6 Сравнительный анализ методов

Минусы Полного перебора

- Входные данные не велики, для N=7 программа укладывается в 1с. Уже для N=10 требуется примерно 40с.
- Временная сложность O(N!)

Плюсы Полного перебора

- Полный перебор дает точное решение.
- Простота реализации

Минусы Метода ветвей и границ

- В худшем случае работает как полный перебор.

Плюсы Метода ветвей и границ

- Возможно значительное сокращение времени работы.
- Простота реализации.

Минусы Жадного алгоритма

- Всегда можно предоставить такой набор, при котором решение будет не точным.

Плюсы Жадного алгоритма

- Высокое время работы, ограниченное только скоростью сортировки, в среднем O(NlogN).
- Может работать с большими значениями N.
- Не использует дополнительных ресурсов компьютера.
- Простота реализации.

Минусы ДП – алгоритма:

- Веса предметов целые, если брать вещественные значения, ДП - алгоритм неприменим!
- Использование большого количества оперативной памяти для хранения таблиц промежуточных решений.
- Для больших значений N количества предметов ДП – алгоритм работает $O(2^N)$.

Плюсы ДП – алгоритма:

- Высокая скорость работы по сравнению с другими алгоритмами (для не больших значений $N<50$).
- Получаем точное решение.
- Имеем оптимальные загрузки рюкзака для всех его весов от 1 до MaxW

На диаграмме 1. показано соотношение времени работы алгоритмов. По вертикальной оси в 1/10000 секунд. По горизонтальной оси в зависимости от количества предметов. Для ДП алгоритма для количества n предметов брался вес w=10*n, так как скорость работы ДП алгоритма зависит от произведения w *n.

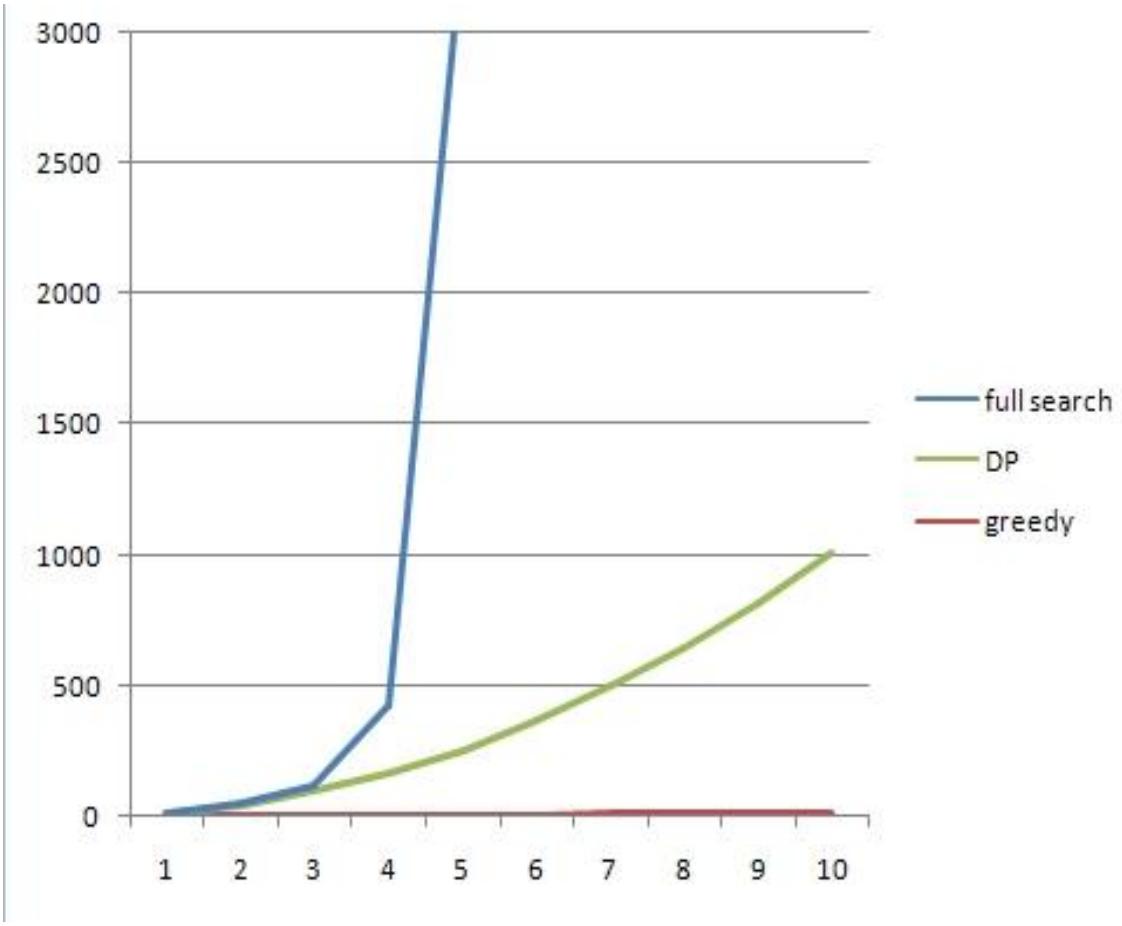


Диаграмма 1

2.7 Модификации задачи

1. Необходимо вывести только максимальную стоимость, набор нас не интересует.
2. В результате работы нужно получить не только максимальную стоимость, но и сам набор.
3. На размер рюкзака несколько ограничений (многомерность задачи).

4. Каждый предмет можно брать только лишь один раз.

5. Предметы можно брать произвольное количество раз.

6. Количество раз помещения предмета в рюкзак фиксировано. Либо для каждого предмета свое, либо для всех предметов одно.

7. Некоторые предметы должны обязательно быть уложены в рюкзак (имеют приоритет).

8. Находить несколько оптимальных решений (одинаковой стоимости, но с разным содержимым).

Заключение

В ходе исследования задачи о рюкзаке были выявлены три основных алгоритма решения. Полный перебор, ДП – программирование, жадный алгоритм. Так же был рассмотрен Метод ветвей и границ, но как сокращение полного перебора. Все методы разделены на две группы. Первая группа – точные методы, сюда входят ДП – алгоритмы, Полный перебор и Метод ветвей и границ. Вторая группа – приближенные методы, к таким методам относится Жадный алгоритм. Выбор использования того или иного метода спорный вопрос, все зависит от постановки задачи, а так же от того, какие цели поставлены. Если требуется найти точное решение, то конечно нужно использовать точные методы, при небольшом наборе входных данных (предметов до 10-20), подойдет перебор или метод ветвей и границ в силу простоты реализации, при больших, следует использовать ДП – алгоритм. Если же точность решения не так важна, или входные данные таковы, что ни один из точных методов не работоспособен, остается применять только приближенные алгоритмы. Но остается возможность комбинирования различных методов для ускорения, или даже применение каких либо “уловок” для конкретного примера. Надеяться же на построение полиномиального алгоритма нет смысла, так как данная задача NP-полна. Безусловно, данная задача очень важна с точки зрения ее приложения в реальной жизни. Не смотря на свою “древность”, рюкзак не только не забывается, наоборот, интерес к нему задаче растет. Оптимальная загрузка транспорта помогает сокращать расходы, получать большую прибыль. Также задача применяется в криптографии и прикладной математике.

Литература

1. Вирт, Н. Алгоритмы и структуры данных [Текст] / Н. Вирт. – Пер. с англ.- М.Мир, 1989.-360 с., ил.
2. Визгунов, Н.П. Динамическое программирование в экономических задачах с применением системы MATLAB [Текст] / Н.П. Визгунов. – Н.Новгород.: ННГУ, 2006. – 48 с.
3. Кузюрин, Н.Н Сложность комбинаторных алгоритмов. Курс лекций [Текст] / Н.Н. Кузюрин, С.А.Фомин. – 2005. – 79 с.
4. Гери, М. Вычислительные машины и труднорешаемые задачи [Текст] / М. Гери, Д. Джонсон. – М.: Мир, 1982 – 416 с.
5. Окулов, С. М - Программирование в алгоритмах [Текст] / С.М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2004. – 341 с.: ил.
6. Окулов, С.М. Информатика в задачах [Текст] / С.М. Окулов, А.А, Пестов, О.А. Пестов. – Киров: Изд-во ВГПУ, 1998. — 343с.
7. Царев, В.А. Проектирование, анализ и программная реализация структур данных и алгоритмов: Учебное пособие [Текст] / В.А. Царев, А.Ф. Дробанов. – Череповец., 2007. – 30 с.
8. Акулич, И.Л Динамическое программирование в примерах и задачах: Учеб. пособие для студентов эконом. спец. вузов [Текст] / И.Л. Акулич. – М.: Высш. шк., 1986. – 319 с., ил.
9. Хаггари, Р. Дискретная математика для программистов [Текст] / Р. Хаггари. – М.: Техносфера, 2003. – 320с.
10. Кормен, Т. Алгоритмы: построение и анализ [Текст] / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.

Приложение 1

Автор задачи: С.М. Окулов

В ресторане собираются N посетителей. Посетитель с номером i, имеющий сумму денег Pi и полноту Si, подходит к двери ресторана во время Ti. Входная дверь ресторана имеет K состояний открытости. Состояние открытости двери может изменяться на одну единицу за единицу времени: дверь открывается на единицу или остается в том же состоянии. В начальный момент времени дверь закрыта (состояние 0). Посетитель с номером i входит в ресторан только в том случае, если дверь открыта специально для него, то есть когда состояние открытости двери совпадает с его степенью полноты Si. Если в момент прихода посетителя состояние двери не совпадает с его степенью полноты, то посетитель уходит и больше не возвращается.

Ресторан работает в течение времени T.

Требуется, правильно открывая и закрывая дверь, добиться того, чтобы за время работы ресторана в нем собрались посетители, общая сумма денег у которых максимальна.

Решение

Это классическая задача на метод динамического программирования. Состояния открытости двери можно представить “треугольной решеткой”, изображенной на рисунке. Каждая вершина определяет степень открытости q в момент времени t. Некоторым вершинам решетки приписаны веса, равные сумме денег у посетителя, приходящего в момент времени t и имеющего степень полноты, равную q. Требуется найти путь по решетке, проходящий через вершины, сумма весов которых имеет максимальное значение. Следует отметить, что нет необходимости хранить оценки всех вершин. Нам необходимо подсчитать оценки для момента времени t. Это возможно, если известны их значения для всех предыдущих моментов времени, однако для подсчета достаточно помнить только оценки для момента времени t–1. Приведем более простой пример входных данных:

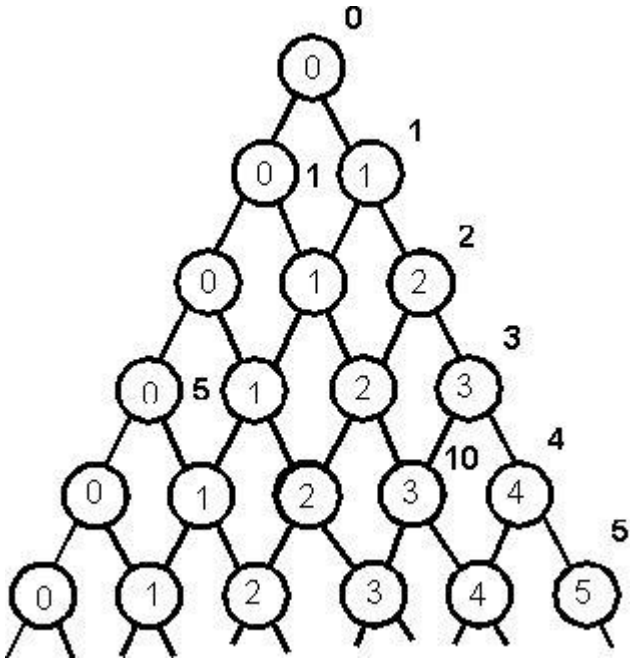
3 5 6

3 4 1

5 10 1

1 4 1

Соответствующая решетка приведена на рисунке. Справа на рисунке указаны моменты времени. Слева от некоторых вершин решетки приведены суммы денег у посетителей, приходящих в этот момент времени и имеющих степень полноты, совпадающей со степенью открытости двери, записанной в вершине решетки. Ответ для данного примера очевиден: 11.



Основная процедура, остальные очевидны:

```
Procedure Solve;
Var i,j:Integer; {массивы для формирования оценок вершин решетки}
SOld,SNew:Array[0..MaxK] Of LongInt;
Begin
SOld[0]:=0;
For i:=1 To K Do SOld[i]:=-MaxLongInt;
SNew:=SOld;
For i:=1 To Tend Do Begin{цикл по моментам времени}
```

```
SNew[0]:=SOld[0];
For j:=1 To i Do{цикл по достижимым состояниям}
SNew[j]:=Max(SOld[j-1],SOld[j]);
{формирование оценок вершин}
For j:=1 To N Do {цикл по посетителям}
If (T[j]=i) And (SNew[S[j]]<>-MaxLongInt)
{если время прихода посетителя совпадает
с рассматриваемым моментом времени
и состояние достижимо, то изменяем
оценку вершины, соответствующей полноте
посетителя}
Then Inc(SNew[S[j]],P[j]);
SOld:=SNew;{запоминаем массив оценок}
End;
Res:=-MaxLongInt;
For i:=1 To K Do Res:=Max(Res,SNew[i]);
{находим максимальное значение
в окончательном массиве оценок}
End;
```

Приложение 2

Реализация метода ДП - программирования для задачи о рюкзаке:

```
program DP2;
{$APPTYPE CONSOLE}
uses SysUtils;
Const MaxW = 200; MaxN = 100;
var Value:array [0..MaxW,0..MaxN] of integer; {массив значений(сколько можно набрать для 1..W весов в 1..N предметов)}
Take :array [0..MaxW,0..MaxN] of boolean; {массив значений брали предмет или нет}
W,P :array [0..MaxN] of integer; {Массив весов, массив цен}
i, N, Weight, MaxWeight :integer;
procedure Init;
begin
assign(input,'input.txt');
reset(input);
readln(N, MaxWeight);
for i:=1 to N do readln(W[i], P[i]);
close(input);
end;
procedure Solve;
begin
fillchar(Take, sizeof(Take), false); {обнуляем}
fillchar(Value, sizeof(Value), 0);
for Weight:=1 to MaxWeight do begin {выбираем оптимум для веса Weight}
for i:=1 to N do {берем предметы с 1 по N}
{если вес предмета больше чем текущий вес рюкзака}
{или предыдущий набор дороже выбираемого}
```

```
if (W[i]> Weight) or (Value[Weight, i-1] >= Value[Weight-W[i], i-1]+P[i])
then begin
Value[Weight, i]:= Value[Weight, i - 1];
{тогдеа берем предыдущий набор}
Take[Weight, i]:= false; {говорим что вещь i не взята}
end
else begin {иначе добавляем к предыдущему набору текущий предмет}
Value[Weight, i]:= Value[Weight - W[i], i-1] +P[i];
Take[Weight, i]:= true; {говорим что вещь i взята}
end;
end;
end;
procedure Done;
begin
assign(output,'output.txt');
rewrite(output);
Writeln('Наилучший набор ', Value[MaxWeight, N]);
Weight:= MaxWeight;
for i:= N downto 1 do if Take[Weight, i] then begin
Write(i,' ');
Weight:= Weight-W[i];
end;
close(output);
end;
begin
Init;
Solve;
Done;
end.
```

Приложение 3

Реализация полного перебора для задачи о рюкзаке:

```
program FS;
{$APPTYPE CONSOLE}
uses SysUtils;
type mas = array[1..50] of integer;
Var N, MaxW:integer;{количество предметов, максимальный вес}
W,P,BestP,NowP:mas;
Max:Integer;
procedure Init;
var i:integer;
begin
assign(input,'input.txt');
reset(input);
readln(N, MaxW);
for i:=1 to N do readln(W[i], P[i]);
close(input);
end;
{передаем Nab - номер набранной группы, OstW-вместимость, stoim-цена набранного (еще не набрали нисколько)}
Procedure Search(Nab, OstW:integer; Stoim:integer);
var i:integer;
begin
{здесь OstW-вес который следует набрать из оставшихся. Stoim-стоимость текущего решения}
{Nab - набор предметов. если наполнили рюкзак
и набрали стоимость больше чем имеется, то считаем это новым решением}
if (Nab > N) and (Stoim > Max) then begin {найдено решение}
```

```
BestP:=NowP;
Max:=Stoim;
end
{иначе если количество взятых <= обьема. забиваем рюкзак дальше}
else if Nab<=N then {иначе если набрано меньше чем влазит}
for i:=0 to OstW div W[Nab] do begin {идем от 0 до ост. места}
NowP[Nab]:=i; {берем предмет Nab 0..OstW div W[Nab] раз}
Search(Nab+1,OstW-i*W[Nab],Stoim+i*P[Nab]);
{после каждого взятия предмета увеличиваем стоимость набора
и уменьшаем место в рюкзаке на вес предмета, так же увеличиваем
количество раз взятия предмета}
end;
end;
procedure print(name:string; out_:mas; num:integer);
var i:integer;
begin
if num=0 then begin
Writeln('Наилучший набор ', Max);
Writeln;
Write(' Номер предмета:');
for i:=1 to n do write(i: 3);
Writeln;
end else begin
Write(name);
for i:=1 to n do write(out_[i]: 3);
Writeln;
end;
end;
procedure Done;
begin
```

```
assign(output,'output.txt');
rewrite(output);
print('Наилучший набор ',bestP,0);
print(' Количество взятых:',BestP,1);
print(' Вес предмета:',W,1);
print('Стоимость предмета:',P,1);
close(output);
end;
begin
init;
Search(1, MaxW, 0);
done;
end.
```

Приложение 4

Реализация Жадного алгоритма для задачи о рюкзаке:

```
program Greedy;
{$APPTYPE CONSOLE}
uses SysUtils;
var W, P:array [1..15000] of integer; {веса, цены}
Price:array [1..15000] of real; {относительная ценность}
Take:array [1..15000] of boolean; {использование предметов}
i, N, BestPrice, NowWeight, MaxWeight:integer;
{Количество предметов, Лучшая стоимость, Текущий вес, Макс. вес}
{Считаем что предметы уже отсортированы}
procedure Init;
begin
assign(input,'input.txt');
reset(input);
readln(N, MaxWeight);
for i:=1 to N do readln(W[i], P[i]);
close(input);
end;
procedure Solve;
begin
fillchar(Take, sizeof(Take), False);
i:=0;
while NowWeight+W[i+1]<MaxWeight do begin
inc(i);
BestPrice:= BestPrice + P[i];
NowWeight:= NowWeight + W[i];
Take[i]:= true;
end;
```

```
end;
procedure Done;
begin
assign(output,'output.txt');
rewrite(output);
i:=1;
writeln('Максимальная стоимость ',BestPrice);
writeln('Вес предметов максимальной стоимости ',NowWeight);
writeln('Используемые предметы');
writeln('Вес Цена');
while Take[i] do begin
writeln(W[i],' ',P[i]);
inc(i);
end;
close(output);
end;
begin
init;
solve;
done;
end.
```