Алгоритмы с возвратом

Лекция 9

Постановка задачи

Интересная область программирования— задачи так называемого «искусственного интеллекта»: ищем решение не по заданным правилам вычислений, а путем проб и ошибок.

Обычно процесс проб и ошибок разделяется на отдельные задачи, и они наиболее естественно выражаются в терминах рекурсии и требуют исследования конечного числа подзадач.

В общем виде весь процесс можно мыслить как процесс поиска, строящий (и обрезающий) дерево подзадач.

Во многих проблемах такое дерево поиска растет очень быстро, рост зависит от параметров задачи и часто бывает экспоненциальным.

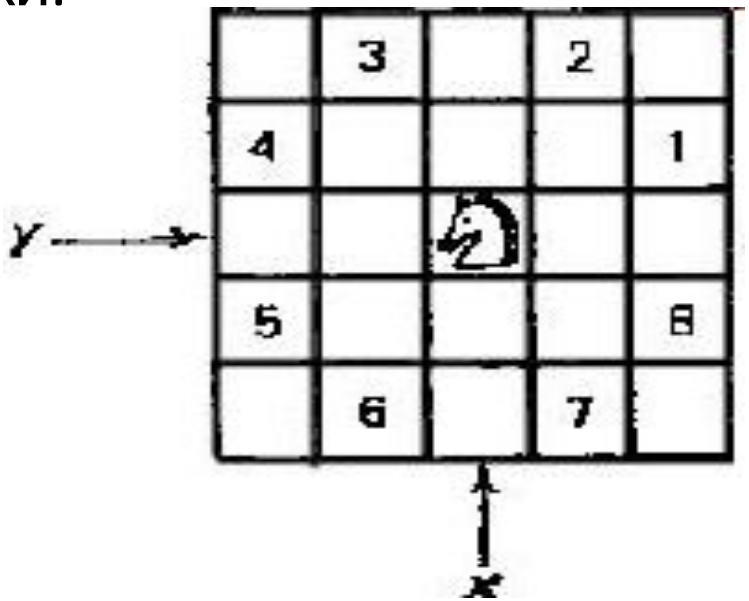
Иногда, используя некоторые эвристики, дерево поиска удается сократить и свести затраты на вычисления к разумным пределам.

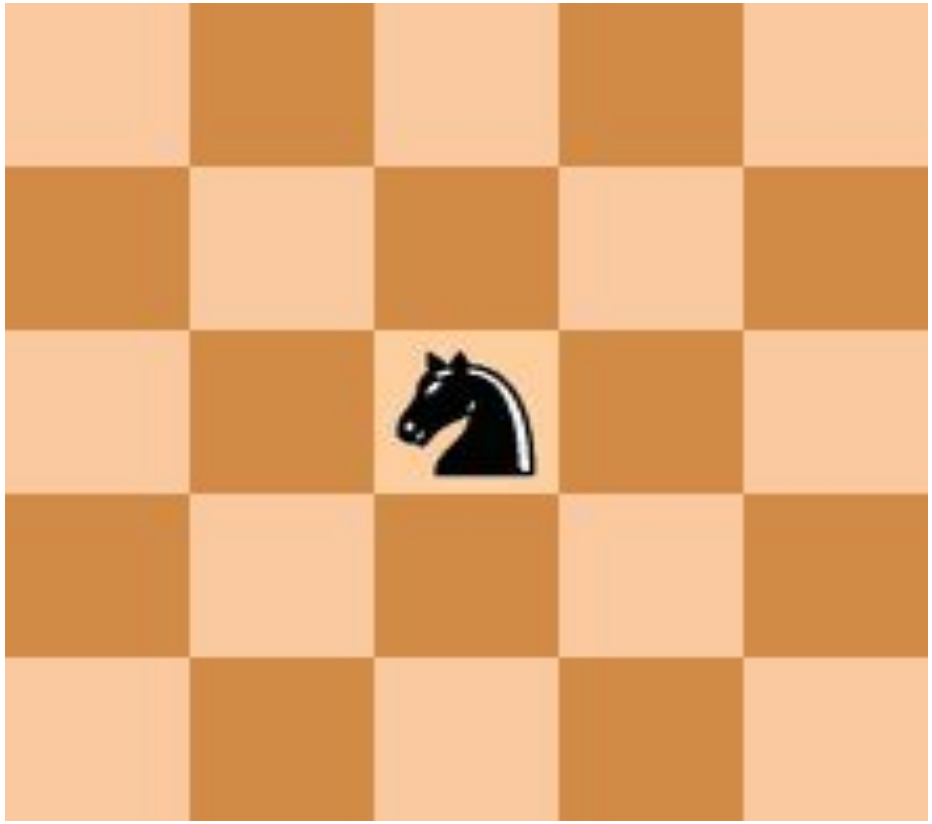
Начнем с демонстрации основных методов на хорошо известном примере — задаче о ходе коня.

Задача о ходе коня

Дана доска размером *n***n*. Вначале на поле с координатами (x_0, y_0) помещается конь — фигура, перемещающаяся по обычным шахматным правилам.

Задача заключается в поиске последовательности ходов, при которой конь точно один раз побывает на всех полях доски.





Алгоритм выполнения очередного хода

```
Try(int i) {
  инициализация выбора хода;
  do
    выбор очередного хода из списка возможных;
    if (выбранный ход приемлем) {
    запись хода;
    if (ход не последний) {
      Try(i+1);
      if(неудача)
         отменить предыдущий ход;
    }
    }
  while(неудача) && (есть другие ходы);
}
```

Выбор представления данных

Доску можно представлять как матрицу *h*:

h [*x*][*y*] = 0 – поле (*x*, *y*) еще не посещалось

h [*x*][*y*] = *i* – поле (*x*, *y*) посещалось на *i*-м ходу

Выбор параметров

Параметры должны определять начальные условия следующего хода и результат (если ход сделан).

В первом случае достаточно задавать координаты поля (*x*, *y*), откуда следует ход, и число *i*, указывающее номер хода.

Очевидно, условие «ход не последний» можно переписать как $i < n^2$.

Кроме того, если ввести две локальные переменные *u* и *v* для позиции возможного хода, определяемого в соответствии с правилами хода коня, то условие «ход приемлем» можно представить как конъюнкцию условий, что новое поле находится в пределах доски

```
(0 ≤ u < n && 0 ≤ v < n)
```

и еще не посещалось `h[u][v] == 0`.

Отмена хода: `h[u][v] = 0`.

Введем локальную переменную *q* для результата.

Конкретизация схемы

```
int Try(int i, int x, int y) {
  int u,v; int q = 0;
  инициация выбора хода;
  do {
   // <u,v> - координаты следующего хода;
   if((0<=u)&&(u<n)&&(0<=v)&&(v<n)
      &&(h[u][v]==0)) {
      h[u][v]= i;
      if (i < n*n) {
      q = Try(i+1,u,v);
      if (!q) h[u][v]=0;
      }
      else q = 1;
   }
  }
  while(!q) && (есть другие ходы);
  return q;
}
```

Выбор ходов

Полю с координатами (x_0,y_0) присваивается значение 1, остальные поля помечаются как свободные.

Если задана начальная пара координат x, y, то для следующего хода *u*, *v* существует максимально восемь возможных вариантов.

Получать *u, v* из *x, y* можно, если к последним добавлять разности между координатами, хранящиеся либо в массиве разностей, либо в двух массивах, хранящих отдельные разности.

Рассмотрим вспомогательную матрицу:

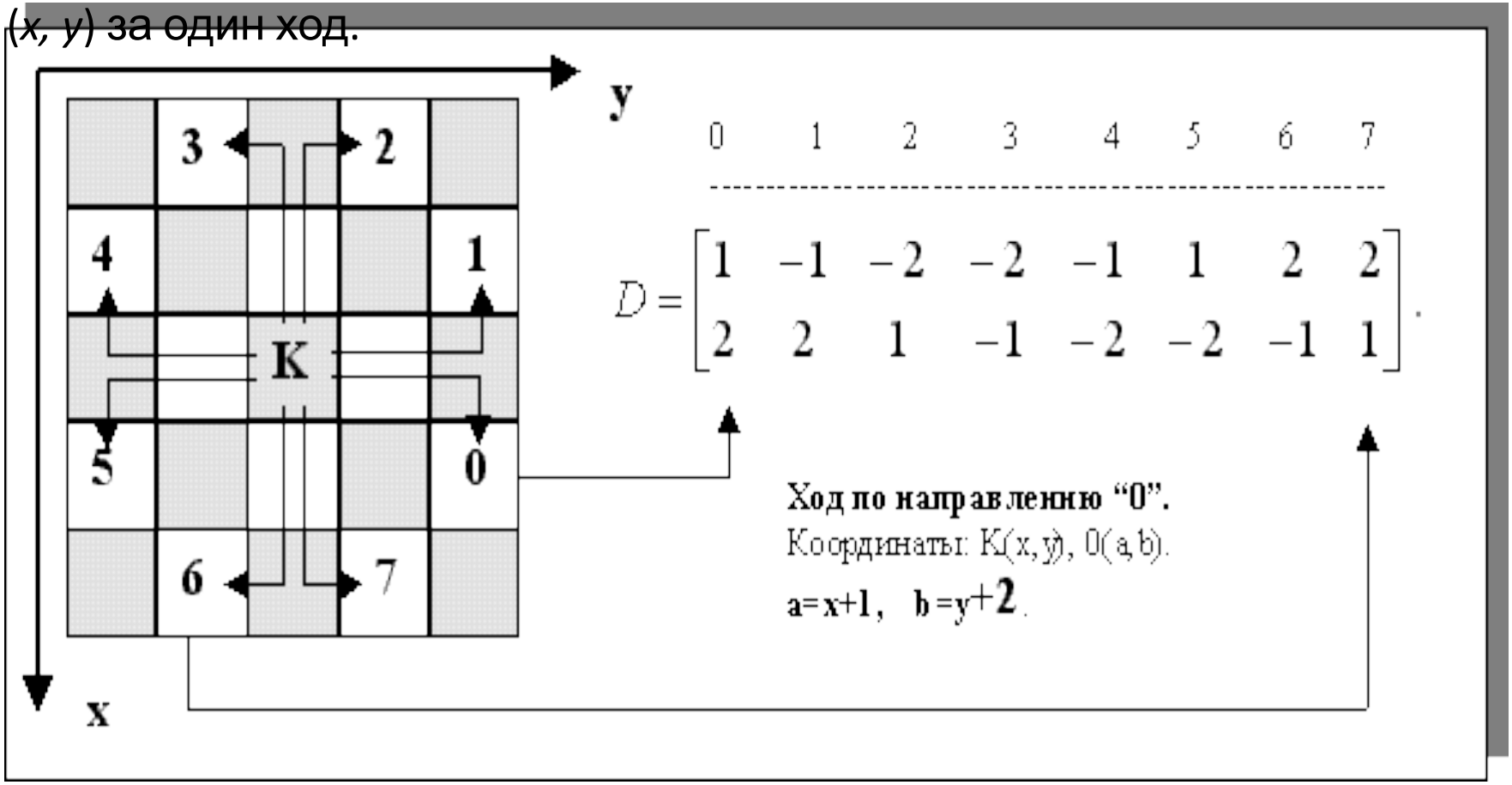
$$D = \begin{bmatrix} 1 & -1 & -2 & -2 & -1 & 1 & 2 & 2 \\ 2 & 2 & 1 & -1 & -2 & -2 & -1 & 1 \end{bmatrix}.$$

Для поля (*x, y*) построим последовательность ходов:

$(x + D_{0,k}, y + D_{1,k})$ $(k = 0, 1, \ldots, 7)$

и отберем из них те, которые не выводят за пределы поля.

Ниже приведен фрагмент доски. Конь *K* стоит в позиции (*x, y*). Клетки с
цифрами вокруг *K* - это поля, на которые конь может переместиться из
(*x, y*) за один ход.



Правило Варнсдорфа, 1823

На каждом ходу ставь коня на такое поле, из которого можно совершить наименьшее число ходов на еще не пройденные поля. Если таких полей несколько, разрешается выбирать любое из них.

Долгое время не было известно, справедливо ли оно.

Верно для доски от 5x5 до 76x76.

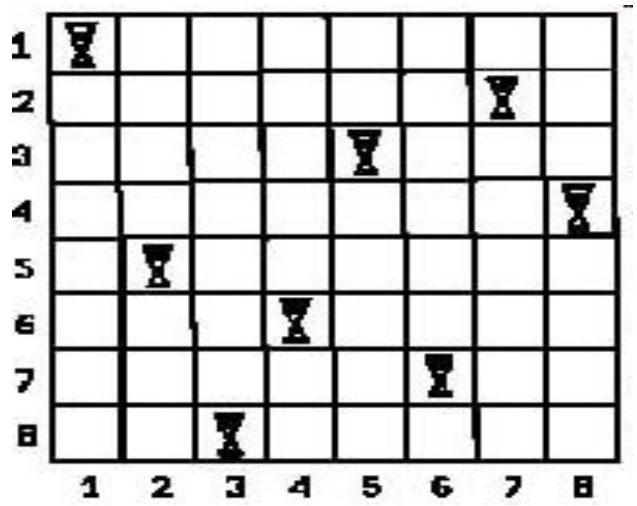
Опровержение правила Варнсдорфа: для любого исходного поля доски указаны контрпримеры, построенные с помощью ЭВМ. Иными словами, с какого бы поля конь ни начал движение, следуя правилу Варнсдорфа, его можно завести в тупик до полного обхода доски.

Задача о восьми ферзях

Задача о восьми ферзях — хорошо известный пример использования методов проб и ошибок и алгоритмов с возвратами.

В 1850 г. эту задачу исследовал К. Ф. Гаусс, однако полностью он ее так и не решил.

Восемь ферзей нужно расставить на шахматной доске так, чтобы ни один ферзь не угрожал другому.



Пример

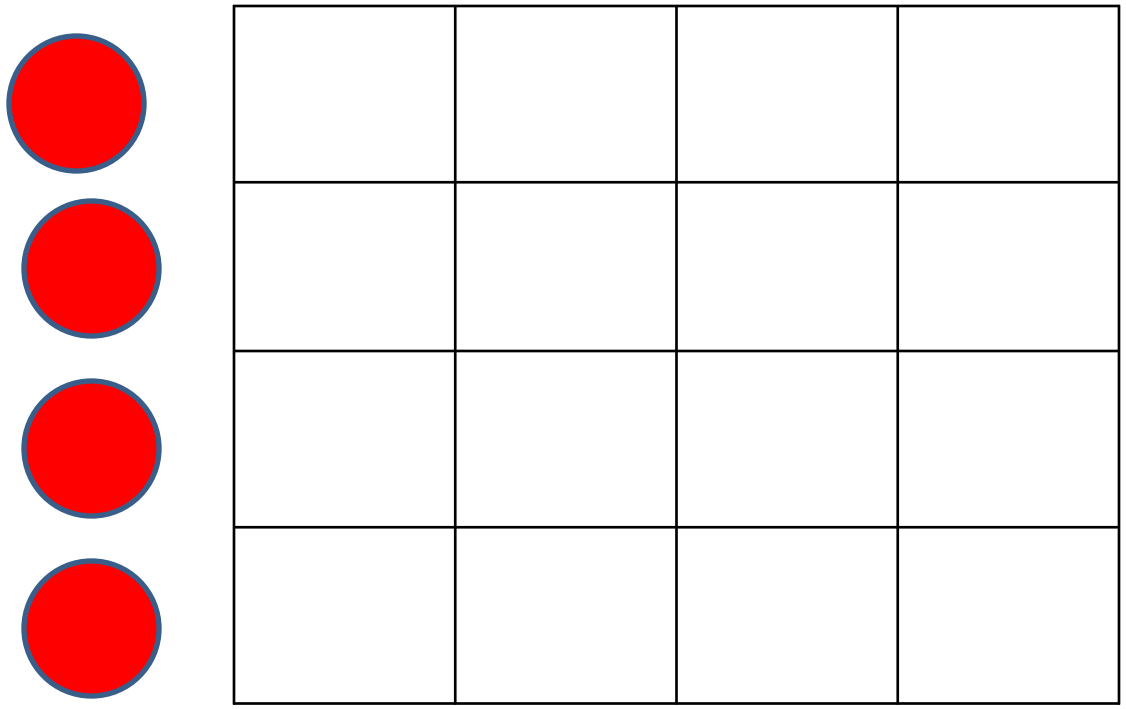


Схема нахождения всех решений

(n – количество шагов, m – количество вариантов на каждом шаге)

```
Try(int i)
{
  int k;
  for (k = 1; k <= m; k++)
  {
   выбор k-го кандидата;
   if (подходит)
   {
      его запись;
      if (i < n) Try(i+1);
      else печатать решение;
      стирание записи   ;
   }
  }
}
```

Задача о стабильных браках

Имеются два непересекающихся множества *A* и *B*. Нужно найти множество пар <*a*, *b*>, таких, что $a \in A$, $b \in B$, и они удовлетворяют некоторым условиям.

Для выбора таких пар существует много различных критериев; один из них называется «правилом стабильных браков».

Пусть A — множество мужчин, а B — женщин. У каждых мужчины и женщины есть различные предпочтения возможного партнера.

Если среди *n* выбранных пар существуют мужчины и женщины, не состоящие между собой в браке, но предпочитающие друг друга, а не своих фактических супругов, то такое множество браков считается нестабильным.

Если же таких пар нет, то множество считается стабильным.

Алгоритм поиска супруги для мужчины *m*

Поиск ведется в порядке списка предпочтений именно этого мужчины.

```
Try(int m) {
  int r;
  for (r=0; r<n; r++) {
   выбор r-ой претендентки для m;
   if (подходит)  {
      запись брака;
      if (m - не последний) Try(m+1);
      else записать стабильное множество;
   }
   отменить брак;
  }
}
```

Выбор структур данных

Будем использовать две матрицы, задающие предпочтительных партнеров для мужчин и женщин: For*Lady* и For*Man*.

ForMan [*m*][*r*] — женщина, стоящая на *r*-м месте в списке для мужчины *m*.

ForLady [*w*][*r*] — мужчина, стоящий на *r*-м месте в списке женщины *w*.

Результат — массив женщин *x*, где *x*[*m*] соответствует партнерше для мужчины *m*.

Для поддержания симметрии между мужчинами и женщинами и для эффективности алгоритма будем использовать дополнительный массив *y*: *y*[*w*] — партнер для женщины w.

Конкретизация схемы

Предикат “подходит” можно представить в виде конъюнкции single и stable, где stable — функция, которую нужно еще определить.

```
Try (int m) {
  int r, w;
  for (r=0; r<n; r++) {
   w = ForMan[m][r];
   if (single[w] && stable) {
      x[m]= w; y[w]= m;
  single[w]=0;
        if (m < n) Try(m+1);
      else record set;
   }
   single[w]=1;
  }
}
```

Стабильность системы

Мы пытаемся определить возможность брака между *m* и *w*, где *w* стоит в списке *m* на *r*-м месте.

Возможные источники неприятностей могут быть:

1) Может существовать женщина *pw*, которая для *m* предпочтительнее *w*, и для *pw* мужчина m предпочтительнее ее супруга.

2) Может существовать мужчина *pm*, который для *w* предпочтительнее *m*, причем для *pm* женщина *w*

1) Исследуя первый источник неприятностей, мы сравниваем ранги женщин, которых *m* предпочитает больше *w*. Мы знаем, что все эти женщины уже были выданы замуж, иначе бы выбрали ее.

```
stable = 1; i = 1;
while((i<r)&& stable){
 pw = ForMan[m][i];
 i = i+1;
 if(!single[pw]) {
   stable = (ForLady[pw][m] > ForLady[pw][y[pw]]};
 }
}
```

2) Нужно проверить всех кандидатов *pm*, которые для *w* предпочтительнее «суженому». Здесь не надо проводить сравнение с мужчинами, которые еще не женаты. Нужно использовать проверку *pm* <*m:* все мужчины, предшествующие *m*, уже женаты.

Напишите проверку 2) самостоятельно!

Задача о кубике

Задано описание кубика и входная строка.
Можно ли получить входную строку, прокатив кубик?

Перенумеруем грани кубика с 123456 на 124536:
1 – нижняя;
6 – верхняя; (1+6 = 7)
3 – фронтальная;
4 – задняя; (3+4 = 7)
2 – боковая левая;
5 – боковая правая (2+5 = 7).
Тогда соседними для *i*-й будут все, кроме *i*-й и (7-i)-й.

Попробуем построить слово, начиная со всех шести граней.

Результат (в переменной *q*) 1, если можно получить слово, записанное в глобальной строке *w*, начиная *n*-го символа, перекатывая кубик, лежащий *g*-ой гранью.

```
int chkword(g, n) {
  if((n>strlen(w)) || (w[n]== ' '))
   return 1;
  if(CB[g] != w[n]) break;
  for(i=1; i<=6; i++) {
   if((i != g) && (i+g != 7))
      q=chkwrd(i,n+1);
      if (q) return 1;
  }
  return 0;
}
```

Нахождение оптимальной выборки (задача о рюкзаке)

Пусть дано множество вещей $\{x_1, x_2, x_3, \ldots x_n\}$.

Каждая *i*-я вещь имеет свой вес w_i, и свою стоимость c_i.

Нужно из этого множества выбрать такой набор вещей, что их общий вес не превышал бы заданного числа *K*, а их общая стоимость была бы максимальной.

t_i = 0, если вещь не взята, и

t_i = 1, иначе

$$\sum_{i=1}^{i \le n} t_i w_i \le K \qquad \sum_{i=1}^{i \le n} t_i c_i \to max$$

Схема перебора всех решений и выбора оптимального

```
Try(int i)
{
  if (включение приемлемо)
  {  включение i-го объекта;
   if (i < n) Try(i+1);
   else проверка оптимальности;
   исключение i-го объекта;
  }
  if (приемлемо невключение )
  {
   if (i < n) Try(i+1);
   else проверка оптимальности;
  }
}
```

Метод ветвей и границ

— метод для нахождения оптимальных решений различных задач оптимизации. Метод — есть вариация полного перебора с отсечением подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Впервые метод ветвей и границ был предложен Лендом и Дойгом в 1960 для решения общей задачи целочисленного линейного программирования. Интерес к этому методу и фактически его “второе рождение” связано с работой Литтла, Мурти, Суини и Кэрела, посвященной задаче коммивояжера. Начиная с этого момента, появилось большое число работ, посвященных методу ветвей и границ и различным его модификациям.

Дерево поиска

В основе метода ветвей и границ лежит идея последовательного разбиения множества допустимых решений на подмножества меньших размеров. Процедуру можно рекурсивно применять к полученным подмножествам. Эти подмножества образуют дерево, называемое *деревом поиска* или *деревом ветвей и границ*. Узлами этого дерева являются построенные подмножества.

На каждом шаге разбиения осуществляется проверка того, содержит ли данное подмножество оптимальное решение или нет. Проверка осуществляется посредством вычисления оценок *снизу* и *сверху* для целевой функции на данном подмножестве. Если для пары подмножеств получается такая ситуация, что *нижняя* граница для первого подмножества дерева поиска больше, чем *верхняя* граница для второго подмножества, то тогда первое подмножество можно исключить из дальнейшего рассмотрения.

Если нижняя граница для узла дерева совпадает с верхней границей, то это значение является минимумом функции и достигается на соответствующем подмножестве.

Использование метода ветвей и границ для решения задачи о рюкзаке

Пусть в переменной оптимум будет храниться лучшее из полученных к этому времени решений. Процедура Try вызывается рекурсивно для исследования очередного объекта до тех пор, пока все объекты не будут рассмотрены. При этом возможны два заключения: либо включать объект в текущую выборку, либо не включать. Оба варианта должны быть рассмотрены.

Пусть *opts* – оптимальная выборка, полученная к данному моменту,

maxv – ее ценность, *t* – текущая выборка.

Объект можно включать в выборку, если он подходит по весовым ограничениям.

Критерием неприемлимости будет то, что после данного исключения общая ценность выборки будет не меньше полученного до этого момента оптимума.

Оценки

Будем рассматривать следующие оценки:

tw – общий вес выборки к данному моменту;

av – общая ценность текущей выборки, которую можно еще достичь.

Условие “включение приемлемо” можно сформулировать в виде выражения: $tw + w_i \leq K$.

Проверка оптимальности будет следующей:

```
if (av  > maxv)   {
   opts = t;
maxv = av;
}
```

Условие “приемлемо невключение” проверяется с помощью выражения:

$av > maxv + c_i$.