Лекция 20

АЛГОРИТМЫ С ВОЗВРАТОМ

План лекции

- Классы задач P и NP, сводимость, NP-полные и NP-трудные задачи
- Метод поиска с возвратом
- Алгоритмы решения классических задач комбинаторного поиска
- Метод ветвей и границ

Классы P и NP

- Класс P (polynomial) -- множество задач, время решения которых ограничено полиномом от размера входных данных
  - увеличение числа на 1 в двоичной записи
  - проверка связности графа, вычисление кратчайших расстояний
  - приведите другие примеры
- Класс NP (non-deterministic polynomial) -- множество задач, время проверки правильности решения которых ограничено полиномом от размера входных данных
  - все задачи класса P – почему?
  - приведите другие примеры
  - приведите пример задачи НЕ из класса NP
- Неизвестно, совпадают ли классы P и NP
  - Стивен Кук 1971, Леонид Левин 1973

Классы P и NP

- Класс P (polynomial) -- множество задач, время решения которых ограничено полиномом от размера входных данных
  - увеличение числа на 1 в двоичной записи
  - проверка связности графа, вычисление кратчайших расстояний
  - приведите другие примеры
- Класс NP (non-deterministic polynomial) -- множество задач, время проверки правильности решения которых ограничено полиномом от размера входных данных
  - все задачи класса P – почему?
  - приведите другие примеры
  - приведите пример задачи НЕ из класса NP
- Неизвестно, совпадают ли классы P и NP
  - Стивен Кук 1971, Леонид Левин 1973

Классы P и NP

- Класс P (polynomial) -- множество задач, время решения которых ограничено полиномом от размера входных данных
  - увеличение числа на 1 в двоичной записи
  - проверка связности графа, вычисление кратчайших расстояний
  - приведите другие примеры
- Класс NP (non-deterministic polynomial) -- множество задач, время проверки правильности решения которых ограничено полиномом от размера входных данных
  - все задачи класса P – почему?
  - приведите другие примеры
  - приведите пример задачи НЕ из класса NP
- Неизвестно, совпадают ли классы P и NP
  - Стивен Кук 1971, Леонид Левин 1973

Сводимость и NP-полные задачи

- Задача п сводится к задаче П, если существует такой алгоритм а решения задачи п, использующий алгоритм А решения задачи П, что если А -- полиномиальный алгоритм, то и а -- полиномиальный алгоритм

Сводимость и NP-полные задачи

- NP-полная задача -- это такая задача из класса NP, к которой сводится любая другая задача из класса NP

- Примеры NP-полных задач
  - Найти в графе цикл, содержащий все вершины (коммивояжёр)
  - Раскрасить вершины графа в C цветов так, чтобы концы каждого ребра были разного цвета (раскраска графа)
  - Найти множество вершин графа, содержащее хотя бы один из концов любого ребра (вершинное покрытие)
  - Дано множество M и (не все) его подмножества П1, П2, ..., Пх. Найти наименьший набор Пк1, Пк2, ..., Пку, покрывающий все множество M (покрытие множества)

Сводимость и NP-полные задачи

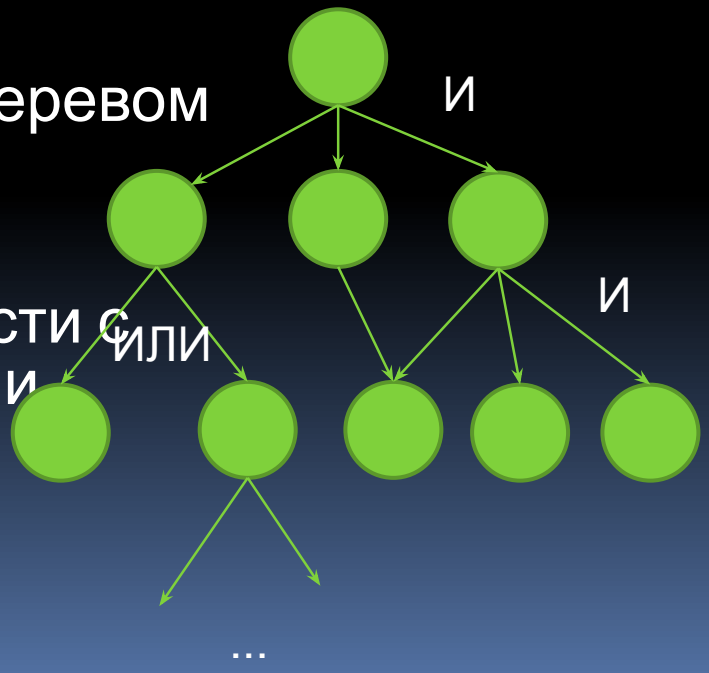
- Задача П называется NP-трудной, если существует NP-полная задача П’, которая сводится к задаче П
- Поиск оптимального решения NP-полной задачи -- NP-трудная задача
- Приведите конкретные примеры NP-трудных задач

Метод поиска с возвратом

- Метод проб и ошибок, он же backtracking
  - Примерно 1950 год
  - Derrick Henry Lehmer, 1905-1991
- Популярный метод в «искусственном интеллекте»
- Делим задачу на несколько меньших задач до тех пор пока не получим задачи с известным решение

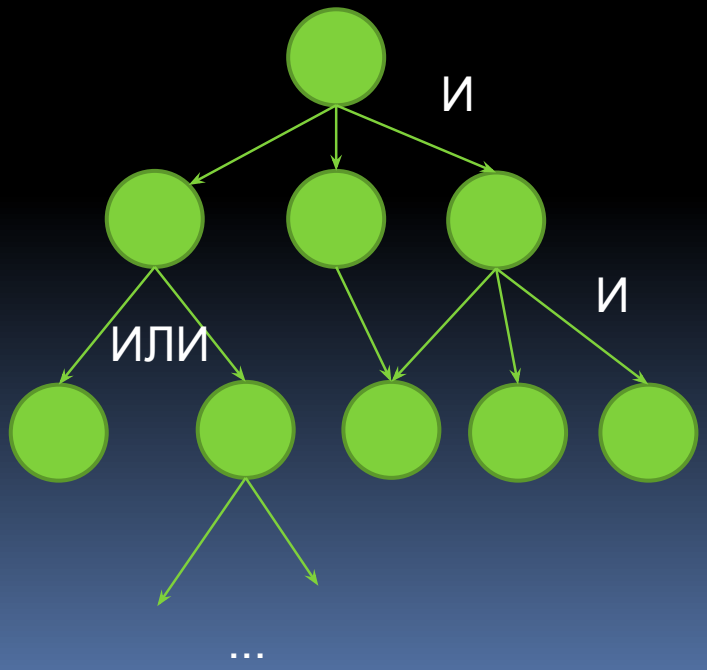
Метод поиска с возвратом

- Граф подзадач
  - Вершины -- задачи
    - И-вершины -- для решения нужно решить все подзадачи
    - Или-вершины -- для решения нужно решить хотя бы одну из подзадач
  - Дуги направлены от задачи к её подзадачам
- Граф подзадач часто бывает деревом
- Размер графа подзадач может «экспоненциально» быстро расти с ростом размера основой задачи



Метод поиска с возвратом

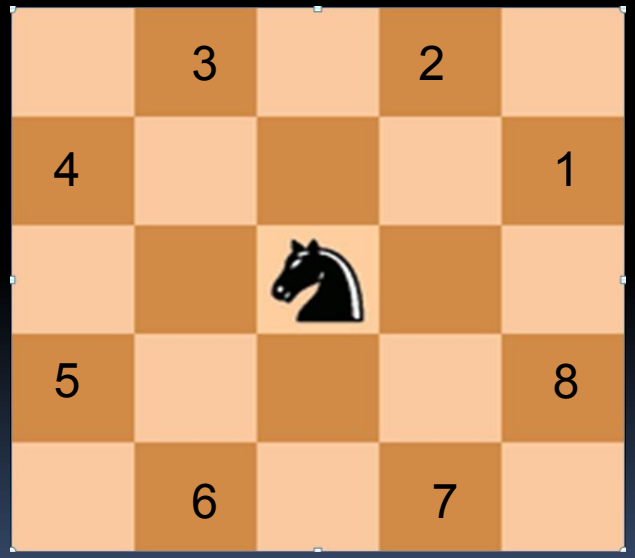
- Решение задачи П -- это обход графа подзадач П, начиная с вершины П по следующим правилам
  - Эвристики позволяют находить решение быстро и не обходить весь граф
    - Найти хорошую эвристику трудно



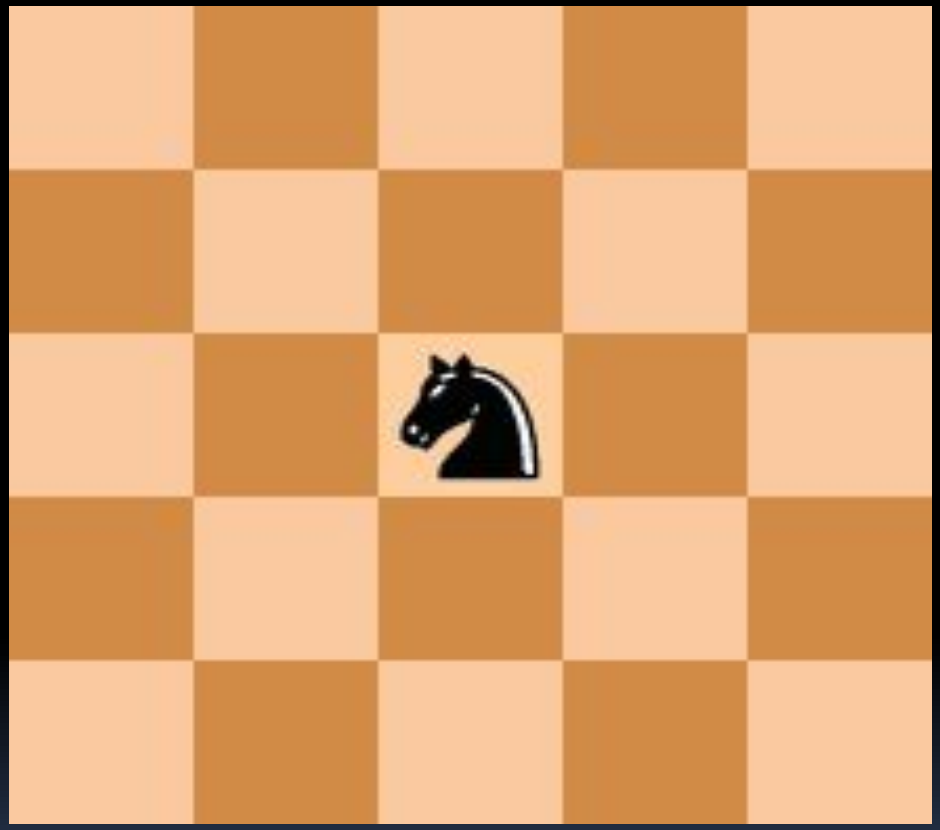
Обход шахматной доски конём

- «Требуется найти последовательность ходов, начинающуюся с поля (x0,y0), при которой конь побывает на каждом поле доски NxN ровно один раз»

- К какой NP-полной задаче проще всего свести обход шахматной доски шахматным конем? Как?



Пример обхода доски 5x5



Алгоритм поиска с возвратом

```
int knight_tour(доска Д, поле П, номер хода Н) {
   if (Д заполнена) return 1;
   Д[П] = Н;
   for (Х = ход коня с поля П) {
      if (Д[Х(П)]==0 &&
            knight_tour(Д, Х(П), Н+1))
         return 1;
   }
   Д[П] = 0;
   return 0;
}
```

Каким будет граф подзадач?

Доска

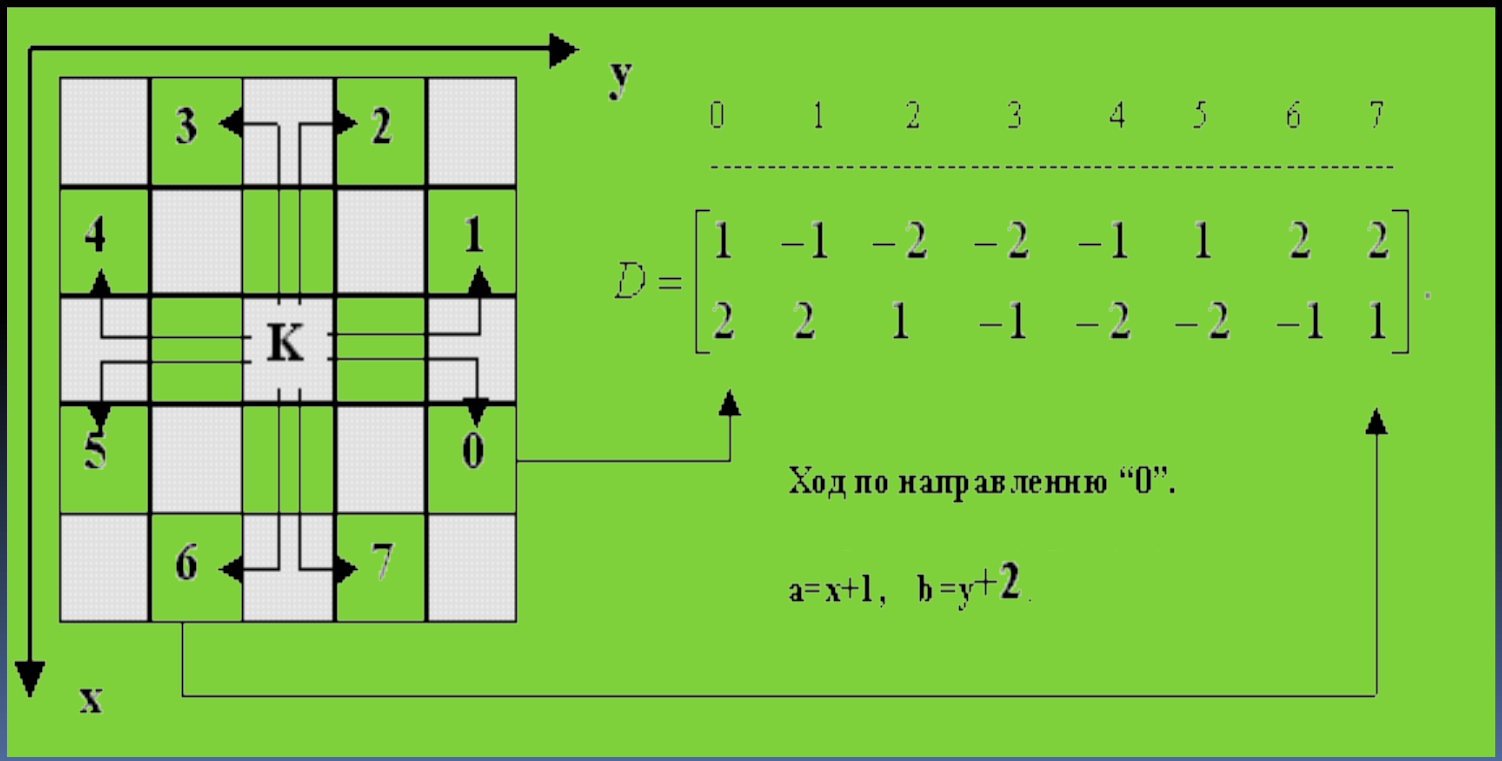
Представление доски матрицей h

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

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

Ходы шахматного коня

- Конь K стоит в позиции (x, y)
- Конь может переместиться из (x, y) на клетки с цифрами за один ход



Реализация 1

```
int knight_tour(int h[], int x, int x, int n) {
    int dx[] = {1,-1,-2,-2,-1,1,2,2};
    int dy[] = {2,2,1,-1,-2,-2,-1,1};
    if (n > N*N) return 1; // N глобальная константа
    h[x][y] = n;
    for (i=0; i<8; ++i) {
        int u = x+dx[i], v = y+dy[i];
        if (u>=0 && u<N && v>=0 && v<N &&
            h[u][v]==0 && knight_tour(h,u,v,n+1))
            return 1;
    }
    h[x][y] = 0;
    return 0;
}
```

Реализация 2

```
int knight_tour(int step, int x, int y, int h[], int n)
{
   static const int dx[] = {1,-1,-2,-2,-1,1,2,2};
   static const int dy[] = {2,2,1,-1,-2,-2,-1,1};
   int u, v, q = 0, i = 0;
   do {
   u = x+dx[i], v = y+dy[i]; // координаты следующего хода
   if (0<=u&&u<n&&0<=v&&v<n&&h[u][v]==0) {
       h[u,v]= step;
       if (step < n*n) {
           q = knight_tour(step+1,u,v,h,n);
           if (!q) h[u][v]=0;
       }
       else q = 1;
   }
   } while(!q && i<8);
   return q;
}
```

Реализация 3

```
int knight_tour(int step, int x, int y, int h[], int n)
{
   static const int dx[] = {1,-1,-2,-2,-1,1,2,2};
   static const int dy[] = {2,2,1,-1,-2,-2,-1,1};
   int u, v, i = 0;
   if (step >= n*n) return 1; // обход закончен
   do {
    u = x+dx[i], v = y+dy[i]; // координаты следующего хода
    if (0<=u && u<n && 0<=v && v<n && h[u][v]==0) {
        h[u,v] = step;
        if (1 == knight_tour(step+1,u,v,h,n)) return 1;
        h[u][v] = 0; // отменяем ход
    }
   } while (i<8);
   return 0;
}
```

Реализация 4

```
int knight_tour(int step, int x, int y, int h[], int n) {
  static const int dx[] = {1,-1,-2,-2,-1,1,2,2};
  static const int dy[] = {2,2,1,-1,-2,-2,-1,1};
  int i;
  if (step >= n*n) return 1; // обход закончен
  for (i = 0; i < sizeof(dx)/sizeof(dx[0]); ++i) {
    int u = x+dx[i], v = y+dy[i]; // координаты следующего хода
    if (0<=u && u<n && 0<=v && v<n && 0 == h[u*n+v]) {
      h[u*n+v] = step;
      if (knight_tour(step+1,u,v,h,n)) return 1; // обход закончен
      h[u*n+v] = 0; // отменяем ход
    }
  }
  return 0; // больше ходов нет и решение не найдено
}
```

Реализация 5

```
int knight_tour(int step, int x, int y, int h[], int n)
{
  static const int dx[] = {1,-1,-2,-2,-1,1,2,2};
  static const int dy[] = {2,2,1,-1,-2,-2,-1,1};
  int i;
  if (step >= n*n) return 1; // обход закончен
  h[x*n+y] = step;
  for (i = 0; i < sizeof(dx)/sizeof(dx[0]); ++i) {
    int u = x+dx[i], v = y+dy[i]; // координаты следующего хода
    if (u<0 || n<=u || v<0 || n<=v) continue;
    if (0 == h[u*n+v] && knight_tour(step+1,u,v,h,n))
      return 1; // обход закончен
  }
  h[x*n+y] = 0; // отменяем ход
  return 0; // больше ходов нет и решение не найдено
}
```

Реализация 6

```
int knight_tour(int step, int p, int h[], int n)
{
  if (step >= n*n) return 1; // обход закончен
  if (p < 0 || p >= n*n) return 0; // выход за границу
  h[p] = step;
  if (0 == h[p-2*n-1] && knight_tour(step+1,p-2*n-1,h,n)) return 1;
  if (0 == h[p-2*n+1] && knight_tour(step+1,p-2*n+1,h,n)) return 1;
  if (0 == h[p-  n-2] && knight_tour(step+1,p-  n-2,h,n)) return 1;
  if (0 == h[p-  n+2] && knight_tour(step+1,p-  n+2,h,n)) return 1;
  if (0 == h[p+  n-2] && knight_tour(step+1,p+  n-2,h,n)) return 1;
  if (0 == h[p+  n+2] && knight_tour(step+1,p+  n+2,h,n)) return 1;
  if (0 == h[p+2*n-1] && knight_tour(step+1,p+2*n-1,h,n)) return 1;
  if (0 == h[p+2*n+1] && knight_tour(step+1,p+2*n+1,h,n)) return 1;
  h[p] = 0; // отменяем ход
  return 0; // больше ходов нет и решение не найдено
}
```

Реализация 7

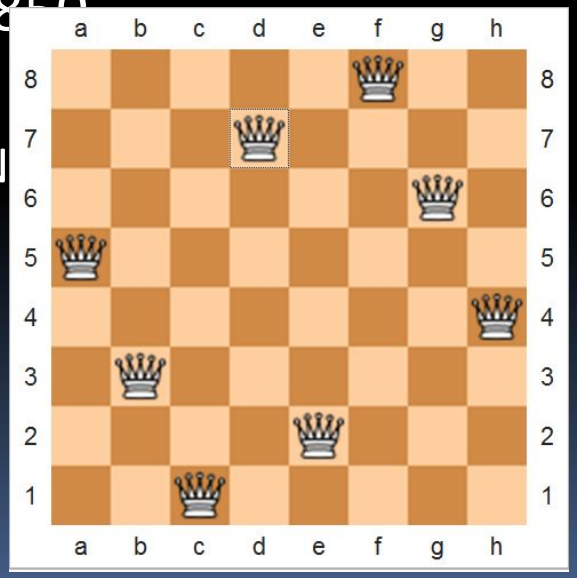
```
int knight_tour(int step, int p, int h[], int n)
{
  if (step >= n*n) return 1; // обход закончен
  if (p < 0 || p >= n*n || h[p] != 0) return 0; // занято, вне поля
  h[p] = step;
  if (knight_tour(step+1,p-2*n-1,h,n)) return 1;
  if (knight_tour(step+1,p-2*n+1,h,n)) return 1;
  if (knight_tour(step+1,p-  n-2,h,n)) return 1;
  if (knight_tour(step+1,p-  n+2,h,n)) return 1;
  if (knight_tour(step+1,p+  n-2,h,n)) return 1;
  if (knight_tour(step+1,p+  n+2,h,n)) return 1;
  if (knight_tour(step+1,p+2*n-1,h,n)) return 1;
  if (knight_tour(step+1,p+2*n+1,h,n)) return 1;
  h[p] = 0; // отменяем ход
  return 0; // больше ходов нет и решение не найдено
}
```

Пример эвристики

- Эвристика Варнсдорфа
  - "На каждом ходу ставь коня на такое поле, из которого можно совершить наименьшее число ходов на еще не пройденные поля. Если таких полей несколько, выбирай любое из них."
- Позволяет обойти без возвратов доски от 5x5 до 76x76
- С помощью ЭВМ найдены размеры N > 76 такие, что с какого бы поля конь ни начал движение, эвристика Варнсдорфа заводит его в тупик

Задача о восьми ферзях

- «Требуется расставить 8 ферзей на шахматной доске так, чтобы ни один ферзь не угрожал другому»
- Формулировка -- Max Bezzel, 1848
- Первое решение -- Franz Nauck, 18
  - Перечислил все 92 решения
  - Расширил на N ферзей на доске NxN



- Используется для проверки скорости работы алгоритмов с возвратом

Задача о восьми ферзях

Construction Through Decomposition:

A Linear Time Algorithm for the N-queens Problem

1984

Bruce Abramson **Mordechai M. Yung**

Department of Computer Science

Columbia University

New York, N.Y. 10027

Пример расстановки 4 ферзей

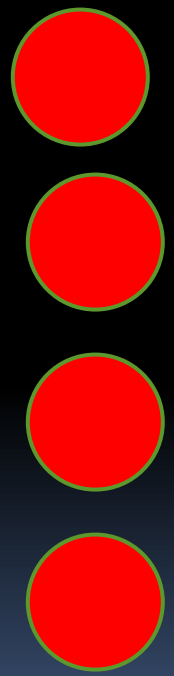


Схема нахождения всех решений

```
int place_queen(int N, доска Д, ферзь Ф, поле П)
{
    if (Ф >= N) return 1; // нашли решение
    Д[П] = Ф;
    for (X = свободное поле Д) {
        if (ни один ферзь не угрожает X &&
                place_queen(N, Д, Ф+1, X))
            return 1;
    }
    Д[П] = 0;
    return 0;
}
```

Задача о рюкзаке

- Дано n вещей
  - i-я вещь имеет вес w_i, и стоимость c_i
- Дано число K – вместимость рюкзака
- Найти набор вещей максимальной стоимости при условии, что их общий вес не превышает K
  - $t_i = 0$, если вещь не взята
  - $t_i = 1$, если вещь взята

$$\sum_{i=1}^{i \le n} t_i w_i \le K$$

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

Схема перебора всех решений и выбора оптимального

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

Метод ветвей и границ

- Вариант полного перебора
- Нахождение оптимальных решений среди допустимых
- Отсечение заведомо неоптимальных допустимых решений
- Ленд и Дойг 1960 общая задача целочисленного линейного программирования
  - A. H. Land and A. G. Doig An automatic method of solving discrete programming problems
- Литтл, Мурти, Суини и Кэрел 1963 задача коммивояжера

Метод ветвей и границ

- Целевая функция
  - В задаче о рюкзаке это

$$\sum_{i=1}^{i\leq n} t_i c_i \rightarrow max$$

- Ограничения
  - В задаче о рюкзаке это

$$\sum_{i=1}^{i\leq n} t_i w_i \leq K$$

- Допустимые решения удовлетворяют ограничениям
- Оптимальные решения – это допустимые решения, дающие максимальное значение целевой функции

Метод ветвей и границ

- Разбиение множества допустимых решений на подмножества меньших размеров
- Подмножества допустимых решений образуют *дерево поиска* (*дерево ветвей и границ*)
- Для каждого подмножества допустимых решений оцениваем *снизу* и *сверху* множество значений целевой функции
  - Если нижняя граница совпадает с верхней границей, то Ц.Ф. достигает максимума (минимума) на данном подмножестве допуст. решений
  - Если *нижняя* граница для значений Ц.Ф. на подмножестве А больше *верхней* границы для значений Ц.Ф. на подмножестве В, то А не содержит минимума Ц.Ф., а В не содержит максимума Ц.Ф.

Метод ветвей и границ

- Ищем оптимальное решение при помощи обхода дерева ветвей и границ
  - Вид обхода выбираем в зависимости от задачи
- На каждом шаге обхода проверяем, содержит ли данное подмножество допустимых решений оптимальное решение
  - да, если верхняя граница == нижняя граница
    - обновляем известный min (max)
  - нет, если нижняя граница > известный min (верхняя граница < известный max)
    - не исследуем (пропускаем) подмножество допустимых решений
  - неизвестно
    - разбиваем подмножество допустимых решений на части и добавлем в дерево новые вершины

Метод ветвей и границ для решения задачи о рюкзаке

- Множество допустимых решений задаём массивом t[] и номером x рассматриваемой вещи
  - значения t[0] … t[x] уже зафиксированы
  - t[0]*w[0]+t[1]*w[1]+…+t[x]*w[x] <= K
  - значения t[x+1] … t[n] еще не зафиксированы
- Оценка снизу для множества допустимых решений t, x
  - тривиальная -- t[0]*c[0]+t[1]*c[1]+…+t[x]*c[x]
  - приведите примеры более "умных" оценок

Схема перебора всех решений и выбора оптимального (копия)

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

Детализация метода ветвей и границ для задачи о рюкзаке

- Обозначим
  - *tw* – общий вес рюкзака к данному моменту
  - *av* – оценка сверху на конечную ценность рюкзака
  - maxv – максимум, известный на данный момент
- "Включение приемлемо"

  $tw + w[i] \leq K$
- "Проверка оптимальности"

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

- “Приемлемо невключение”

  $av < maxv$

Заключение

- Классы задач P и NP, сводимость, NP-полные и NP-трудные задачи
- Метод поиска с возвратом
- Алгоритмы решения классических задач комбинаторного поиска
- Метод ветвей и границ

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;
  }
}
```

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(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* предпочтительнее его супруги.

1) Исследуя первый источник неприятностей, мы сравниваем ранги женщин, котрых *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!

Перебор ходов

- Из поля (x, y) достижимы не более 8 полей

  (u, v) = (x + D[0,k], y + D[1,k]), k = 0, 1, ..., 7

  где массив D[2][8] заполнен следующим образом

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

- Для (x, y) вблизи края доски не рассматриваем k, для которых (u, v) лежат за пределами доски