Основы программирования

Алгоритмы с возвратом

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.

Постановка задачи

Бэктрекинг (backtracking, на русском его называют «Поиск с возвратом» или «Полный перебор») – механизм решения переборных задач. Позволяет выбрать либо одну приемлемую альтернативу, либо наилучшую.

Суть режима возвратов состоит в следующим. В программе встречаются точки «развилки» — точки в которых необходимо выбрать один из вариантов поведения («альтернатив»). В дальнейшим может оказаться, что выбранный вариант «неудачен» и нужно вернуться к точке развилки и выбрать какую-то другую альтернативу.

Постановка задачи

Особенность Бэктрекинга состоит в том, происходит откат по управлению, а также по данным. При откате программа «забывает» все, что она сделала в результате выбора неудачной альтернативы — данные восстанавливают свои предыдущие значения, отменяются все последствия выбора альтернативы. После отката берем следующую альтернативу и заново выполняем все действия. Проверяем, какие альтернативы для нас приемлемы. Если в данной развилке все альтернативы оказываются неудачными, распространение неуспеха происходит на вышележащие развилки.

Бэктрекинг обеспечивает:

— откат по управления;

— откат по данным;

— перебор списка альтернатив.

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

Задача о нахождении маршрута шахматного коня, проходящего через все поля доски по одному разу.

Эта задача известна по крайней мере с XVIII века.

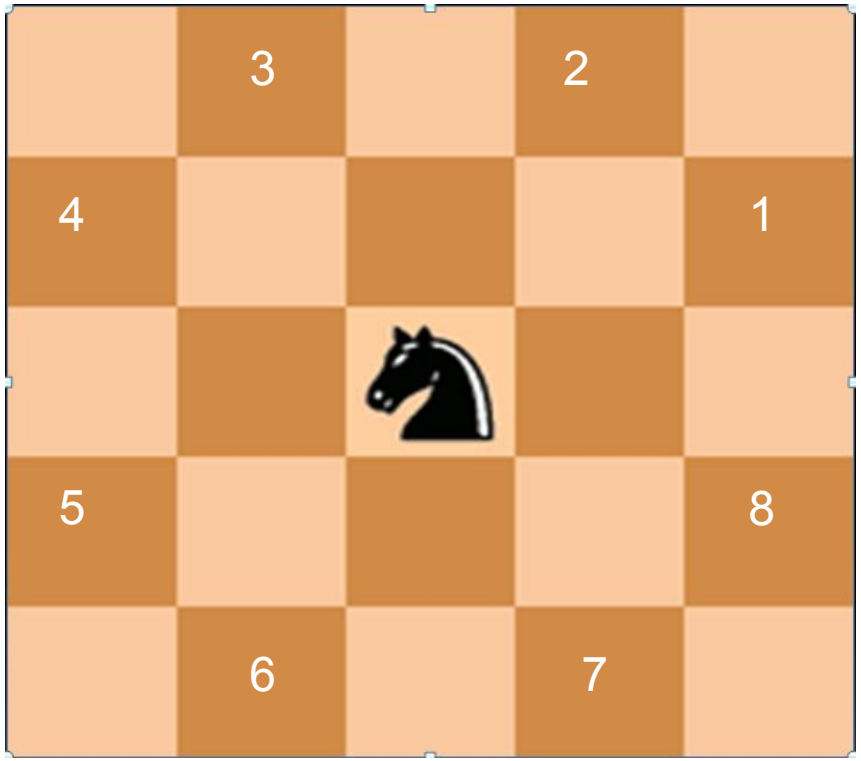
Леонард Эйлер посвятил ей большую работу *«Решение одного любопытного вопроса, который, кажется, не подчиняется никакому исследованию»* (датируется 26 апреля 1757 года).

Помимо рассмотрения задачи для коня, Эйлер разобрал аналогичные задачи и для других фигур. С тех пор обобщённая задача носит имя *«нахождение эйлерова маршрута».*

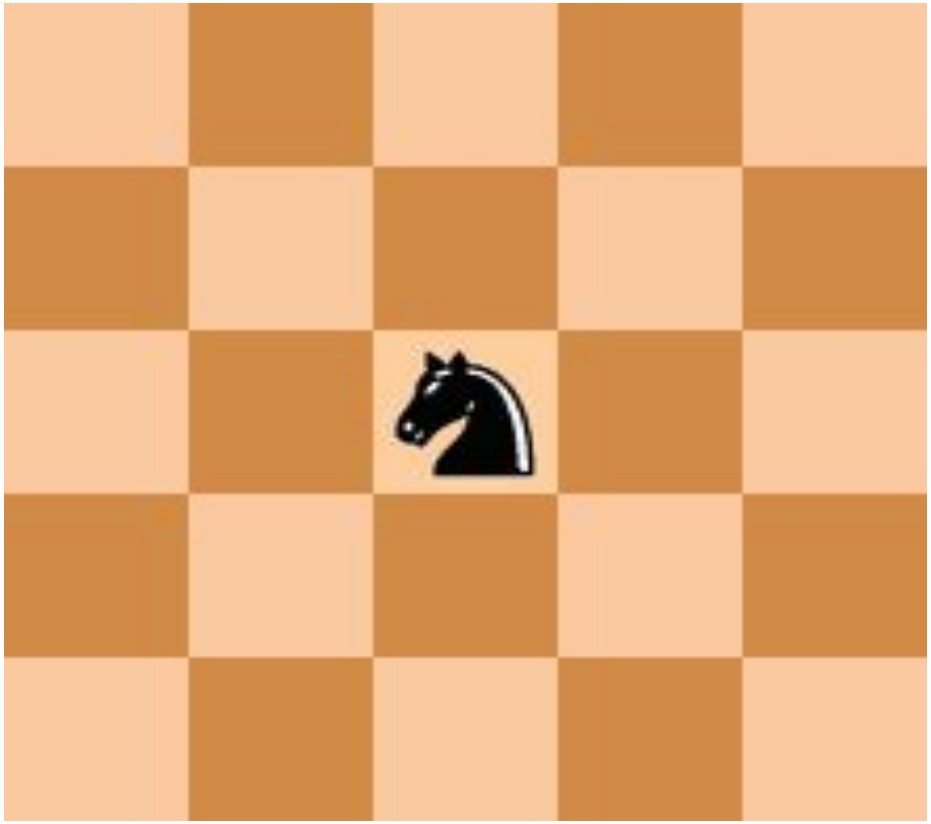
Задача о ходе коня

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.



Задача о ходе коня



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);
}
```

Алгоритм выполнения очередного хода

(поиск 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;
}
```

Выбор представления данных

Доску можно представлять как матрицу *h[n][n]*:

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:

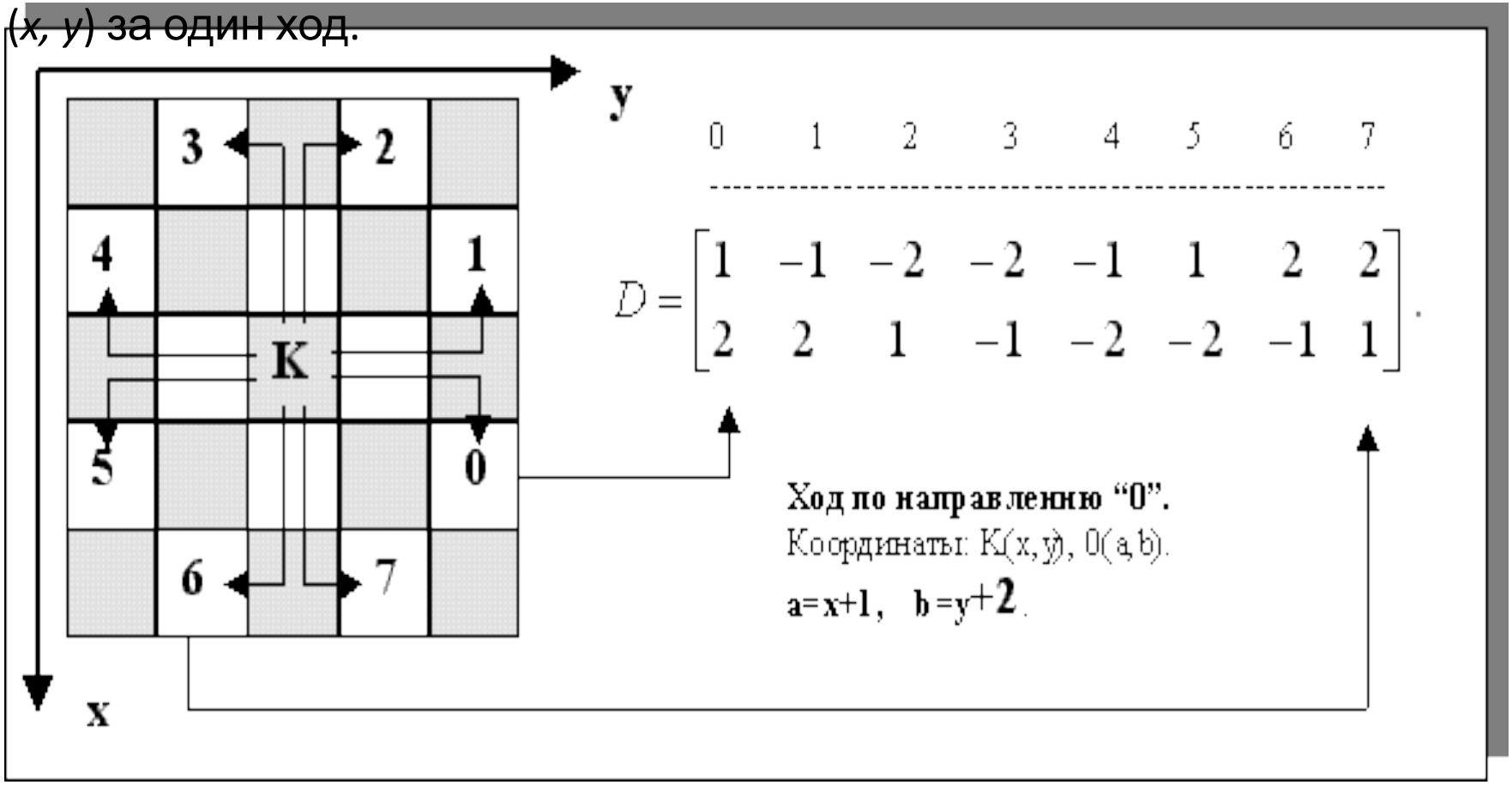
$$D = \begin{bmatrix} 1 & -1 & -2 & -2 & -1 & 1 & 2 & 2 \\ 2 & 2 & 1 & -1 & -2 & -2 & -1 & 1 \end{bmatrix}.$$

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

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

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.



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
}
```

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.

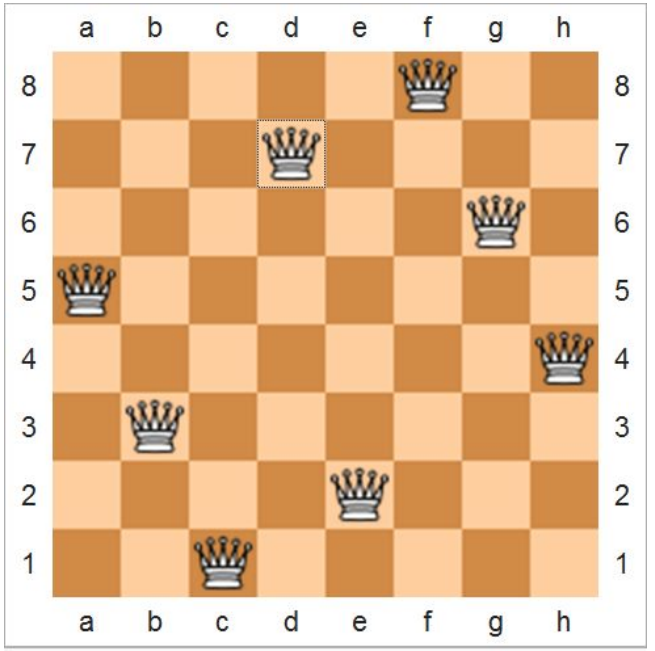
В настоящее время найдено **26 534 728 821 054** варианта закрытых маршрутов, количество открытых – ещё до конца не рассчитано.

Задача о восьми ферзях

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

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

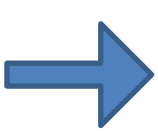
- Формулировка задачи -- Max Bezzel, 1848
- Первое решение -- Franz Nauck, 1850
  * Перечислил все 92 решения
  * Расширил на N ферзей на доске NxN
- В 1850 г. эту задачу исследовал К. Ф. Гаусс, однако полностью он ее так и не решил.
- Эйдзгер Дейкстра решил эту задачу с помощью программирования

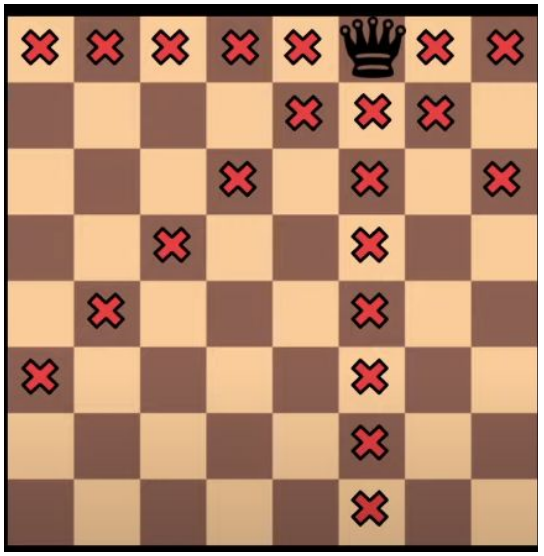


Эта задача используется для проверки скорости работы алгоритмов с возвратом

Задача о восьми ферзях

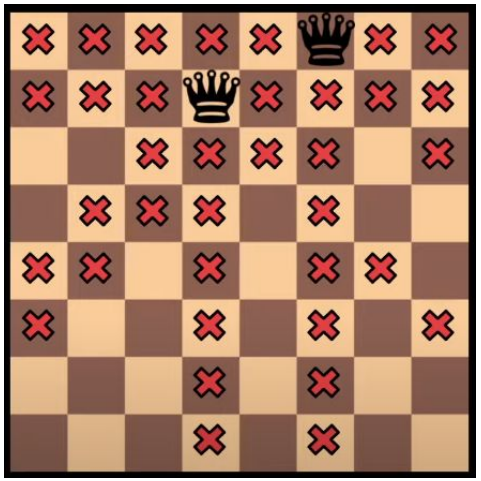
Допустим мы поставили ферзя сюда и отметили все клетки которые он атакуют.

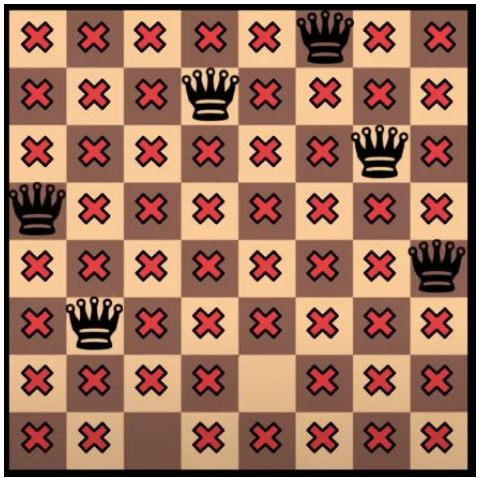


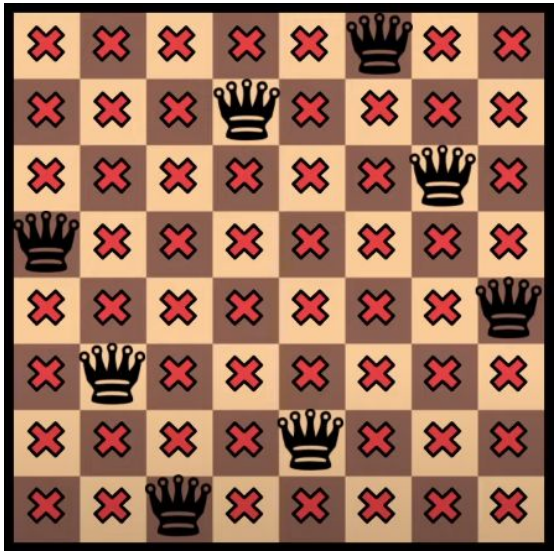


Затем мы поставили второго ферзя, и так далее до тех пор пока все клетки не стали Атакованным.

Итак в данном примере все 8 ферзей мы смогли разместить но нахождение этого расположения не такая уж тривиальная задача.

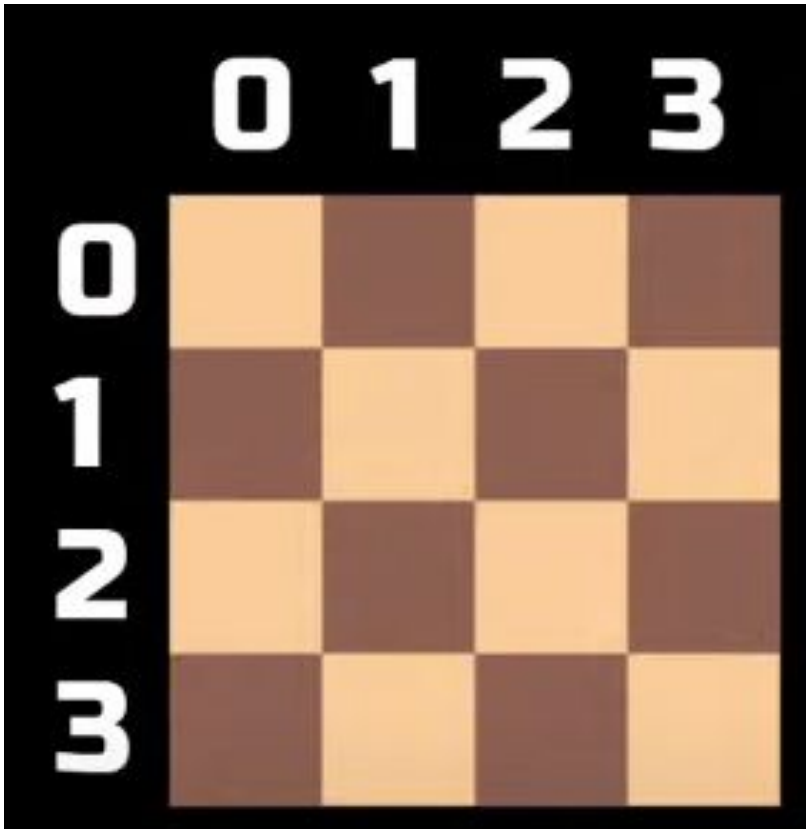






Задача о восьми ферзях

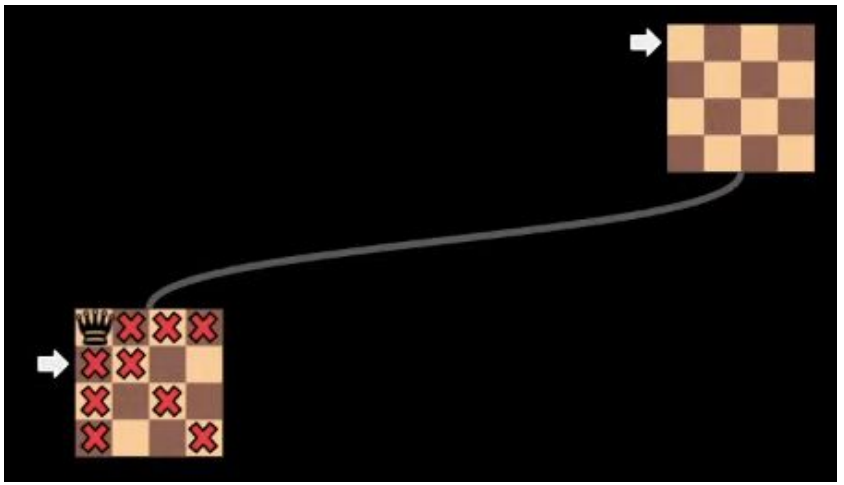
Математики очень часто для того чтобы решить задачу решают ее более простую версию то есть задачу для меньшего размера.
Также поступим и мы. Давайте возьмем ему шахматную доску размера 4 на 4 и попытаемся расставить все на нее четыре ферзя



Задача о восьми ферзях

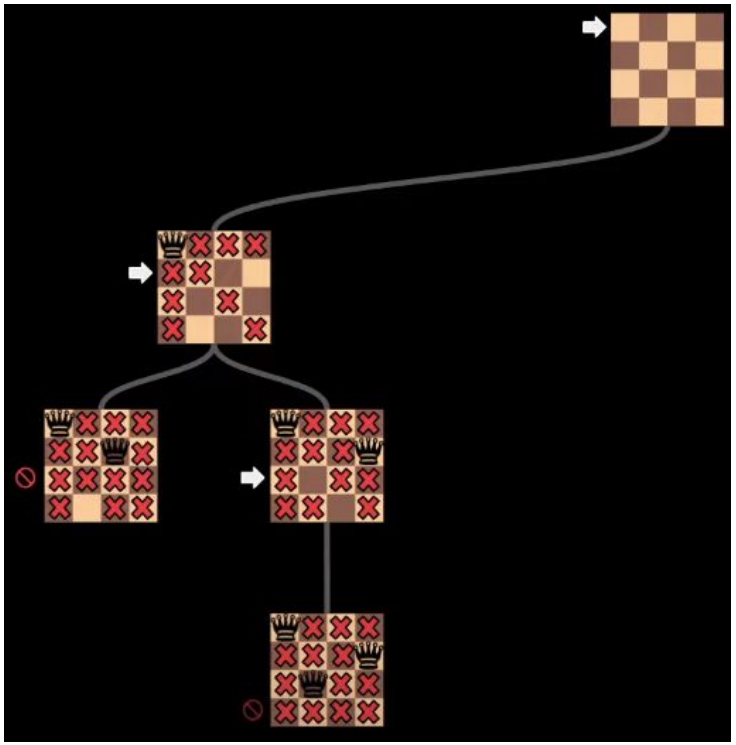
Понятно, что в конечном решении в каждой строчке и должен быть хотя бы один ферзь.

Поэтому давайте поставим ферзя на первую строчку. Мы получим вот такую конфигурацию.



Затем мы можем поставить ферзя уже либо на третью (в таком случае мы получаем тупик, то есть дальше у нас точно мы не придем к Решению)

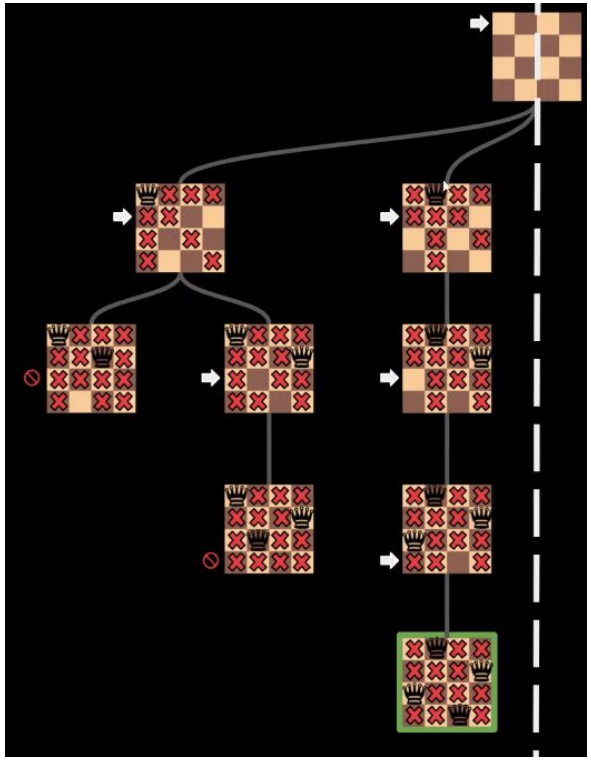
Либо поставить на четвертую строчку, таком случае у нас третьим шагом мы сможем поставить в третью строчку, но опять приходим к тупику.



То есть, изначально эта ветка я не могла нас привести к какому-то решению.

Задача о восьми ферзях

Поэтому подставим ферзя на вторую клетку и посмотрим куда мы пойдем.



Итак решение мы нашли.
Для 3 и 4 клеток, все будет симметрично поэтому мы не будем рассматривать эти обходы.
Этот метод и применил Дейкстра, и он
Называл его backtracking (на русском его называют очень часто «Поиск с возвратом»)

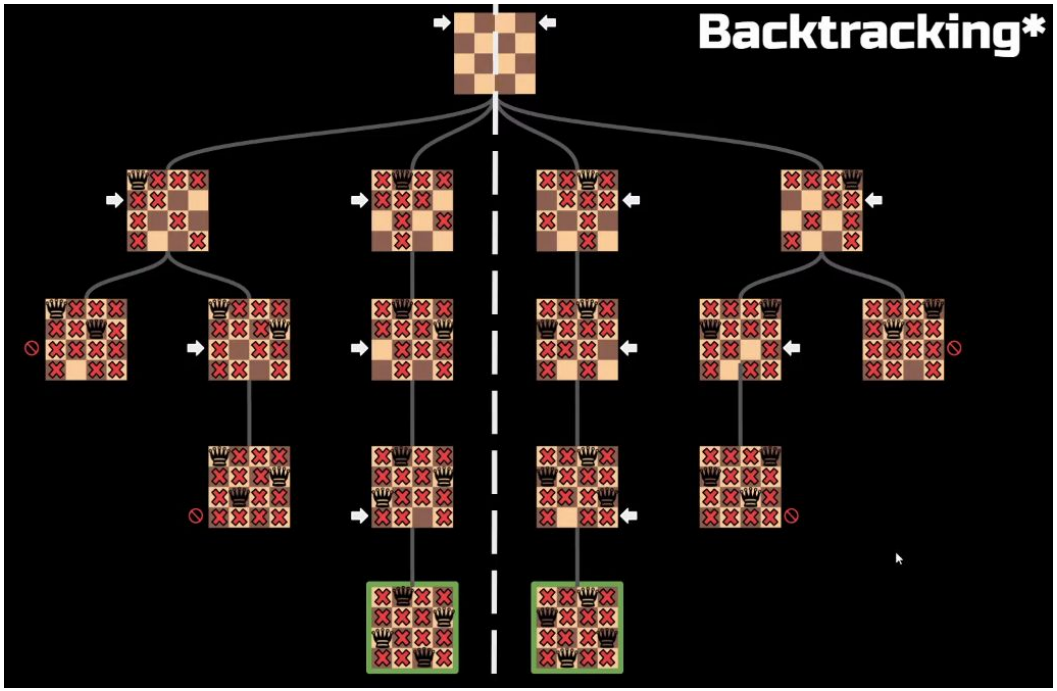


Схема нахождения всех решений

(псевдокод)

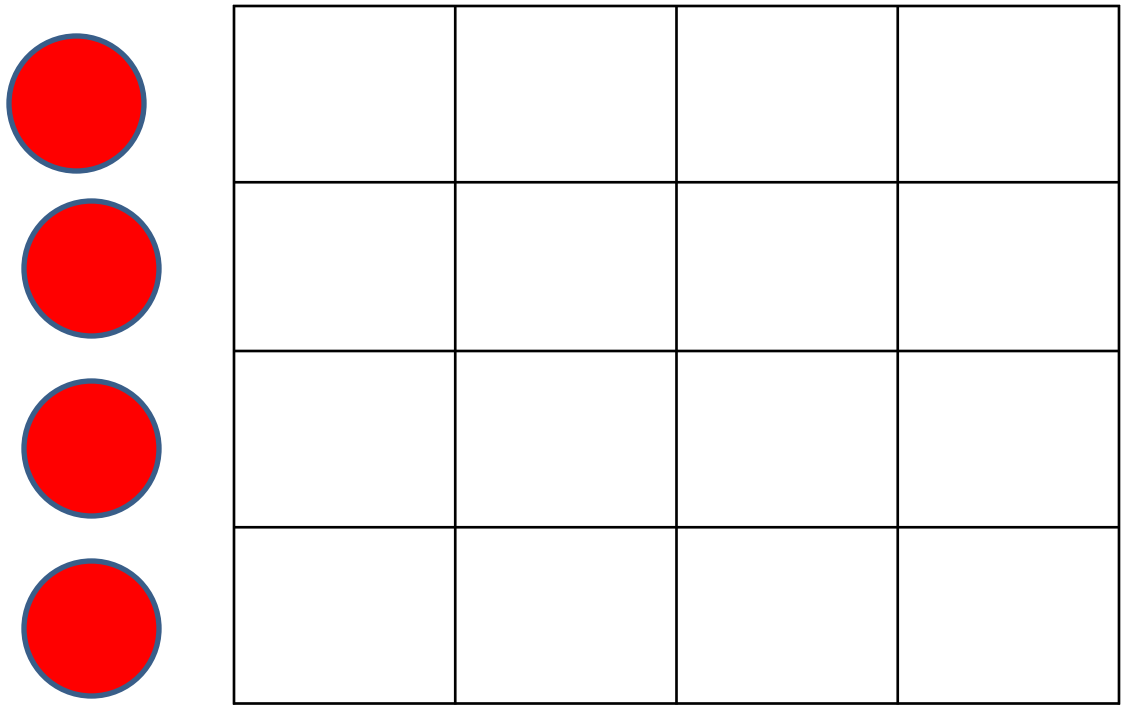
```
int place_queen(int N, доска Д, ферзь Ф, поле
  П)
{
  if (Ф >= N) return 1; // нашли решение
  Д[П] = Ф;
  for (Х = свободное поле Д) {
   if (ни один ферзь не угрожает Х &&
        place_queen(N, Д, Ф+1, Х))
      return 1;
  }
  Д[П] = 0;
  return 0;
}
```

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   ;
   }
  }
}
```

Пример



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*.

Подобная ситуация характерна для многих похожих задач, в которых нужно сделать распределение с учетом предпочтений, например выбор университета студентами, выбор новобранцев различными родами войск и т. п. Пример с браками особенно интуитивен; однако следует заметить, что список предпочтений остается неизменным и после того, как сделано распределение по парам. Такое предположение упрощает задачу, но представляет собой опасное искажение реальности (это

Задача о стабильных браках

Алгоритм:

- Женщины делают предложение наиболее предпочитаемому мужчине;
- Каждый мужчина из всех поступивших предложений выбирает наилучшее и отвечает на него «может быть», на все остальные отвечает «нет»;
- Женщины, получившие отказ, обращаются к следующему мужчине из своего списка предпочтений
- Женщины, получившие ответ «может быть», ничего не делают;
- Если мужчине пришло предложение лучше предыдущего, то он прежней претендентке (которой ранее сказал «может быть») говорит - «нет», а новой претендентке говорит - «может быть»;
- шаги повторяются, пока у всех Женщин не исчерпается список предложений, в этот момент мужчины отвечают «да» на те предложения «может быть», которые у них есть в настоящий момент.

Для алгоритма требуется порядка n^2 шагов.

Задача о стабильных браках

В каждый последующий день (3-й,4-й,5-й…) процесс повторяется. Каждая женщина, получившая отказ ранее, делает предложение следующему кандидату. И мужчины опять имею т возможность сменить пару на более лучшую в его списке. Если они получили лучшее предложение, они опять разрывают помолвки и т.д.

Условие задачи таково, что в определённый момент алгоритм завершается. Более того, когда это происходит, окончательные помолвки считаются стабильными.

Пример:

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
Дарси	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	Шарлотта	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Задача о стабильных браках

Запустим алгоритм:

В 1-й день каждая девушка делает предложение лучшему юноше из своего

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
Дарси	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	Шарлотта	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Бингли получает **три** предложения (от Шарлотты, Джейн и Лидии).

Викли получает **одно** предложение от Элизабет. Остальные – пока ничего.

У Бингли теперь есть выбор и он предпочитает **Джейн** (т.к. она выше всех в его рейтинге), отвергая предложения Лидии и Шарлотты.

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
Дарси	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	Шарлотта	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Задача о стабильных браках

Таким образом, в конце 1-го дня у нас две предварительные помолвки:

Элизабет + Викли и Джейн + Бингли

На 2-й день Шарлотта и Лидия, которых отклонили ранее, делают следующие предложения по **своим спискам** предпочтений. Элизабет и Джейн нет нужды искать пару – у них, пока всё хорошо. Согласно спискам, Шарлотта приглашает Дарси, а Лидия - Викли.

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
Дарси	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	Шарлотта	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Викли **предварительно помолвлен** с Элизабет, но ему **больше нравится** Лидия.

Он **отказывает** Элизабет, и она **возвращается** к поиску пары.

Таким образом, в конце 2-го дня у нас три предварительные помолвки:

Шарлотта + Дарси, Джейн + Бингли и Лидия + Викли.

Задача о стабильных браках

На 3-й день Элизабет выбирает следующего по своему списку – Дарси. Дарси получает предложение от Элизабет, которую он **предпочитает** Шарлотте (она ниже стоит в его списке).

Шарлотта **отвергнута**, теперь свободна выбирать на следующий день, и снова имеем три предварительные помолвки:

Элизабет + Дарси, Джейн + Бингли и Лидия + Викли

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
~~Дарси~~	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	~~Шарлотта~~	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Задача о стабильных браках

На 4-й день Шарлотта делает предложение следующему по своему списку – Коллинсу. Мы видим, что на самом деле он не высокого о ней мнения, он предпочитает Джейн, Элизабет и Лидию.

Но **он не получил ни одного лучшего предложения** и ему остаётся только **согласиться**. И теперь алгоритм завершается, т.к. все обручены:

Шарлотта	Элизабет	Джейн	Лидия
Бингли	Викли	Бингли	Бингли
Дарси	Дарси	Викли	Викли
Коллинс	Бингли	Дарси	Дарси
Викли	Коллинс	Коллинс	Коллинс

Бингли	Коллинс	Дарси	Викли
Джейн	Джейн	Элизабет	Лидия
Элизабет	Элизабет	Джейн	Джейн
Лидия	Лидия	Шарлотта	Элизабет
Шарлотта	Шарлотта	Лидия	Шарлотта

Шарлотта + Коллинс, Элизабет + Дарси, Джейн + Бингли и Лидия + Викли

Задача о стабильных браках

Теперь попробуем чуток программировать. Для простоты разобьём выбор по дням. Вспомним наши условия:

В 1-й день каждая женщина делает предложение первому мужчине в своём списке. Таким образом, некоторые мужчины получают много предложений, некоторые мало, некоторые – не получают совсем. Потом, те, у кого несколько предложений, делают выбор – они принимают предложения тех, кто была **выше** всех в их, личном, списке. В конце у нас получаются предварительные помолвки. Т.е., предложения некоторых женщин, **пока**, не отклонены. Такие пары предварительно помолвлены (кандидаты на стабильную пару), но алгоритм ещё не закончен.

На 2-й день, каждая **отклонённая** женщина, делает предложение следующему варианту в **своём** списке, **независимо** от того помолвлен он или нет! Некоторые мужчины получают **новые** предложения, другие – не получают ничего. Теперь мужчина имеет возможность поменять партнёра, т.е. у него **была** предварительная помолвка, но теперь появился **лучший** вариант. Он **может расторгнуть** предыдущую помолвку и **заключить** новую.

Алгоритм поиска супруги для мужчины *m*

Возможное направление поиска решения – пытаться распределить по парам членов двух множеств одного за другим, пока не будут исчерпаны оба множества.

Пусть Try(m) означает алгоритм поиска жены для мужчины m, и пусть этот поиск происходит в соответствии с порядком списка предпочтений, заявленных этим мужчиной. Первая версия, основанная на этих предположениях, такова:

```
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 жене для мужчины *m*.

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

На самом деле массив y избыточен, так как в нем представлена информация, уже содержащаяся в x. Действительно, соотношения x[y[w]] = w, y[x[m]] = m выполняются для всех m и w, которые состоят в браке.

Поэтому значение *y*[*w*] можно было бы определить простым поиском в x.

Однако ясно, что использование массива *y*[*w*] повысит эффективность алгоритма. Информация, содержащаяся в массивах x и y, нужна для определения стабильности предполагаемого множества браков. Поскольку это множество строится шаг за шагом посредством соединения индивидов в пары и проверки стабильности после каждого предполагаемого брака, массивы x и y нужны даже еще до того, как будут определены все их компоненты.

Конкретизация схемы

Условие «подходит» можно представить в виде конъюнкции определения стабильности брака, реализующейся функцией stable, и того, что претендентка свободна, что отражается в массиве single. Для уточнения стабильности нужно помнить, что оно следует из сравнения рангов, которые можно вычислить по значениям ForMan и ForLady. С учетом введенных структур данных функция Try преобразуется к следующему виду:

```
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, 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

На гранях кубика изображены буквы. Будем считать, что буква, находящаяся на нижней стороне кубика, отпечатывается на бумаге. Перекатывая кубик с грани на грань, мы получим слово из отпечатанных букв. Повороты букв не учитываем.

Итак, нам даны описание кубика и входная строка.

Можно ли получить входную строку, перекатывая кубик с грани на грань?

Для решения задачи перенумеруем грани кубика с 123456 на 124536, а именно:

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 перекатывать кубик поочередно со всех шести граней. Для хранения букв, записанных на гранях кубика, введем массив CB, в CB[1] содержится буква с нижней грани, в CB[2] - с боковой левой грани и т. д. Результат работы программы будем хранить в переменной q. Значение q будет равно 1, если можно получить слово, записанное в глобальной строке w, начиная с n-го символа, перекатывая кубик, лежащий на грани *g*. Ниже приведена рекурсивная функция, реализующая это решение.

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; // 1. дошли до конца строки!
  if( CB[g] != w[n] ) return 0; // 3. буква на
  грани не совпадает с буквой в слове
  for(i = 1; i <= 6; i++) // начинаем
  поочередно со всех 6-ти граней
  {
   if((i != g) && (i + g != 7)) // 5. если не
  эта грань и не противоположная
      q = chkwrd(i, n + 1); // 6. продолжим
  процесс с выбранной гранью и следующей буквой
      if (q) return 1; // 7.
  }
  return 0;
}
```

В основной программе следует выполнить ввод массива CB и строки w и выполнить приведенную ниже последовательность операторов:

```
q = 0;
n = 0 ;
chkwrd(1,0);
```

В функции chkword в строке 1 осуществляется проверка конца заданного слова w. Если смогли дойти до конца этого слова, то возвращаем значение 1. В строке 3 выполняется прерывание перекатывания кубика с грани *g* в том случае, если буква, записанная на этой грани, не совпадает с буквой в строке w, находящейся на позиции n. В строках 5-7 организована дальнейшая проверка получения оставшейся части заданного слова. При этом перекатываем кубик, поочередно начиная со всех граней, выполняя проверку на совпадение букв, записанных на соответствующей грани кубика и в слове на позиции n + 1. Результат проверки записывается в переменную **q**.

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.

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

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. Каждый предмет можно взять **ТОЛЬКО ОДИН РАЗ!**

Критерием неприемлимости будет то, что после данного исключения общая ценность выборки будет не меньше

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:

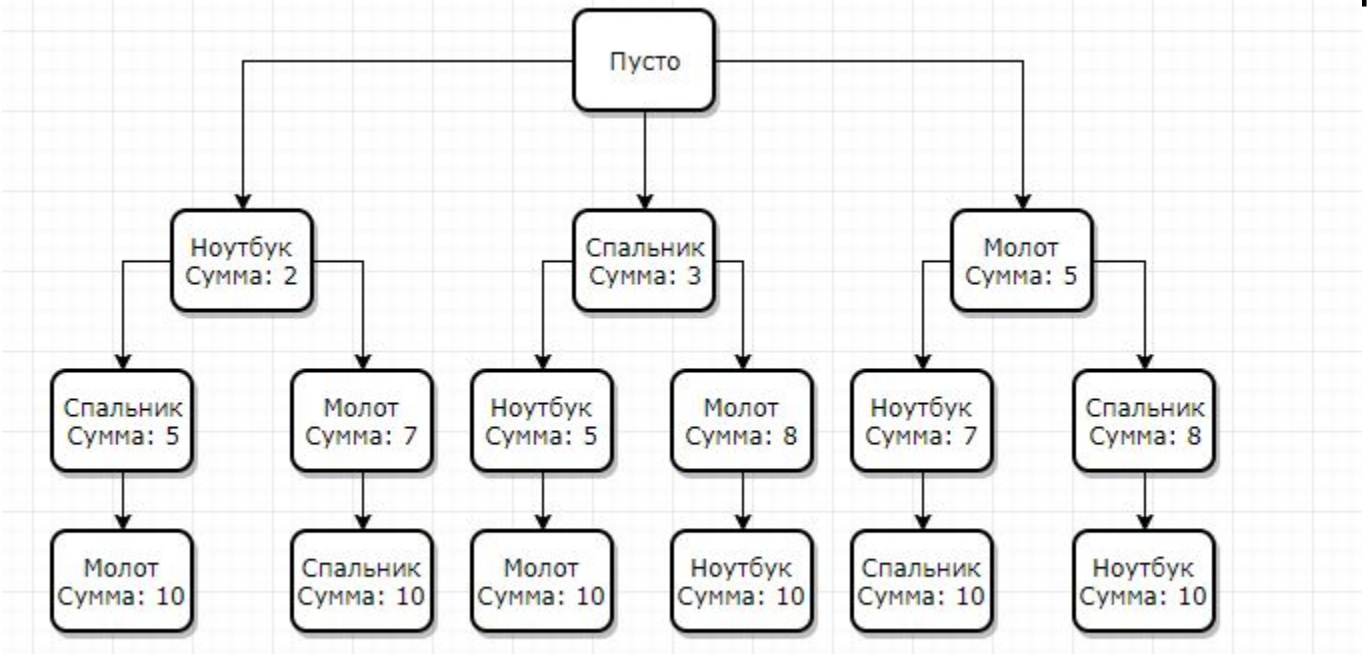
$av > maxv + c_i$.

Оценки

Таким образом узел ***Дерева полного перебора***, это наше текущее состояние, т.е. количество предметов, которое лежит в данный момент в рюкзаке. Также у нас есть ветви, это соединительные пути к другим узлам (состояниям), т.е. как будто мы кладём одну вещь в рюкзак.

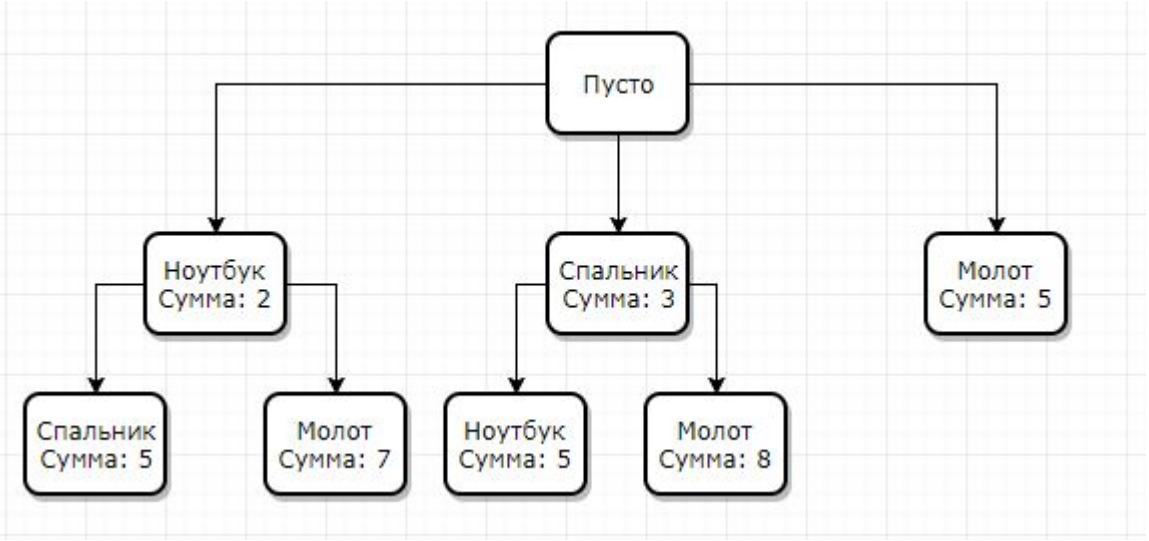
Допустим у нас есть пустой рюкзак грузоподъёмностью (5кг) и нам надо положить: ноутбук (2кг), спальник (3кг), молот (5кг).

Тогда дерево полного перебора будет выглядеть следующим образом:



Оценки

В этом дереве мы можем заметить, что у нас есть узлы, которые превышают вес рюкзака и при этом у них есть ещё и потомки, а зачем нам продолжать рассматривать следующие варианты, если дальше будет только хуже? Поэтому мы ограничиваем весом 5 и получаем такой граф:



На выходе получаем, что есть 2 варианта, либо мы кладём в рюкзак: ноутбук и спальник, либо молот.

В зависимости от реализации, можно искать все варианты, первый или задать дополнительные критерии оптимальности.

Пятиминутка

1. Задача о стабильных браках

Даны два множества, состоящие из 5 мужчин и 5 женщин. Известны предпочтения каждого человека. Из этих множеств нужно построить пары <женщина, мужчина> таким образом, чтобы браки были стабильными.

Предпочтения задаются двумя матрицами:

ForMan [m][r] — номер женщины, стоящей на r-м месте в списке предпочтений для мужчины m.

ForLady [w][r] — номер мужчины, стоящего на r-м месте в списке предпочтений женщины w.

ForMan

1	4	5	2	1	3
2	5	1	2	4	3
3	3	4	2	5	1
4	4	1	2	3	5
5	3	5	4	1	2

ForLady

1	5	4	3	1	2
2	3	5	4	2	1
3	4	1	5	2	3
4	1	4	5	2	3
5	5	1	4	3	2

Составить стабильные пары: для каждой женщины записать номер подходящего ей мужчины:

1	
2	
3	
4	
5	

Пятиминутка

2. Задача о рюкзаке

Дан набор из 5 вещей:

Номер вещи	1	2	3	4	5
Стоимость	10	2	4	8	6
Вес	6	5	3	6	3

Построить оптимальную выборку из 5 предметов при ограничении **10 кг** и определить:

1) стоимость оптимальной выборки_____

2) набор вещей, которые ее составляют_____

Алгоритм выполнения очередного хода

Этот код представляет реализацию алгоритма поиска решения в игре, которая может быть представлена в виде дерева ходов. Каждый узел в дереве представляет собой ход, а каждый лист - это конечное состояние игры.

1. **`i`** - это текущий уровень в дереве игры, функция **`Try(int i)`** – это функция, которая рекурсивно пытается найти решение игры..

2. **`инициализация выбора хода`** - это первоначальная инициализация наших переменных или состояния игры.

3. **`выбор очередного хода из списка возможных`** - это процесс выбора следующего хода из доступных вариантов.

4. **`if (выбранный ход приемлем)`** - проверка, является ли выбранный ход приемлемым.

5. **`запись хода`** - запись выбранного хода в какой-то нашей структуре данных, которая будет использована для отката хода, если это будет необходимо.

6. **`if (ход не последний)`** - проверка, является ли текущий ход последним в игре.

7. **`Try(i+1)`** - рекурсивный вызов функции для следующего уровня игры, если наш ход не последний.

8. **`if(неудача)`** – проверка, удачен ли выбранный ход.

9. **`отменить предыдущий ход`** - отмена предыдущего хода, если текущий ход оказался неудачным.

10. **`while(неудача) && (есть другие ходы)`** - цикл, который продолжается, пока не будет найдено приемлемое решение или не закончатся возможные ходы.

Обратите внимание, что в данном коде нет явного указания на то, что происходит, если решение найдено. Обычно это было бы означало завершение рекурсии и возврат к предыдущим уровням.

Алгоритм выполнения очередного хода (поиск с возвратом)

Это рекурсивная функция, которая пытается найти решение для "задачи о коне" (knight's tour problem). Задача состоит в том, чтобы коня по полю переместить, посетив каждое поле только один раз.

`if (Д заполнена) return 1;` - Если доска (массив Д) полностью заполнена (т.е., все поля посещены), функция возвращает 1, что означает успех.

`Д[П] = Н;` - Текущему полю на доске присваивается номер хода (Н).

`for (Х = ход коня с поля П) {...}` - Для каждого возможного хода коня из текущего поля (П) проверяется, можно ли сделать этот ход (т.е., не выходит ли он за пределы доски и не был ли он уже посещен ранее), и если можно, рекурсивно вызывается функция knight_tour для этого хода.

`if (Д[Х(П)]==0 && knight_tour(Д, Х(П), Н+1)) return 1;` - Если следующий рекурсивный вызов функции **`knight_tour`** вернул 1 (т.е., успешно нашла решение), функция возвращает 1.

`Д[П] = 0;` - Если не удалось найти решение для текущего поля, оно обнуляется, чтобы позволить посетить его позже.

`return 0;` - Если не удалось найти решение для текущего поля и для всех его возможных ходов, функция возвращает 0, что означает неудачу.

Обратите внимание, что функция не проверяет, можно ли сделать ход конем (т.е., нет ли препятствий или не выходит ли коня за пределы доски), это должна быть сделана до вызова функции **`knight_tour`**.