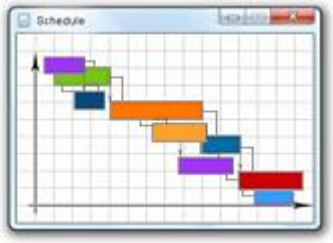
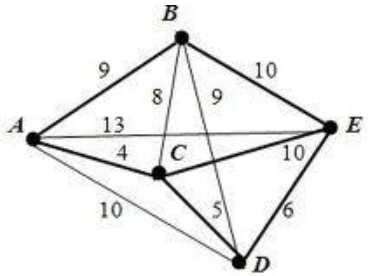
Карпов В.Э.

Планирование

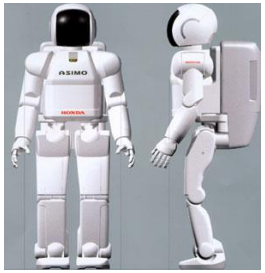
Задачи планирования



Календарное планирование



Планирование маршрутов



Планирование действий

Планирование маршрута. Поиск на графе

Волновой алгоритм (алгоритм Ли) - алгоритм поиска кратчайшего пути на планарном графе (поиск в ширину), 1961

Распространение волны

ЦИКЛ

ДЛЯ каждой ячейки loc, помеченной числом d пометить все соседние свободные не помеченные ячейки числом d + 1

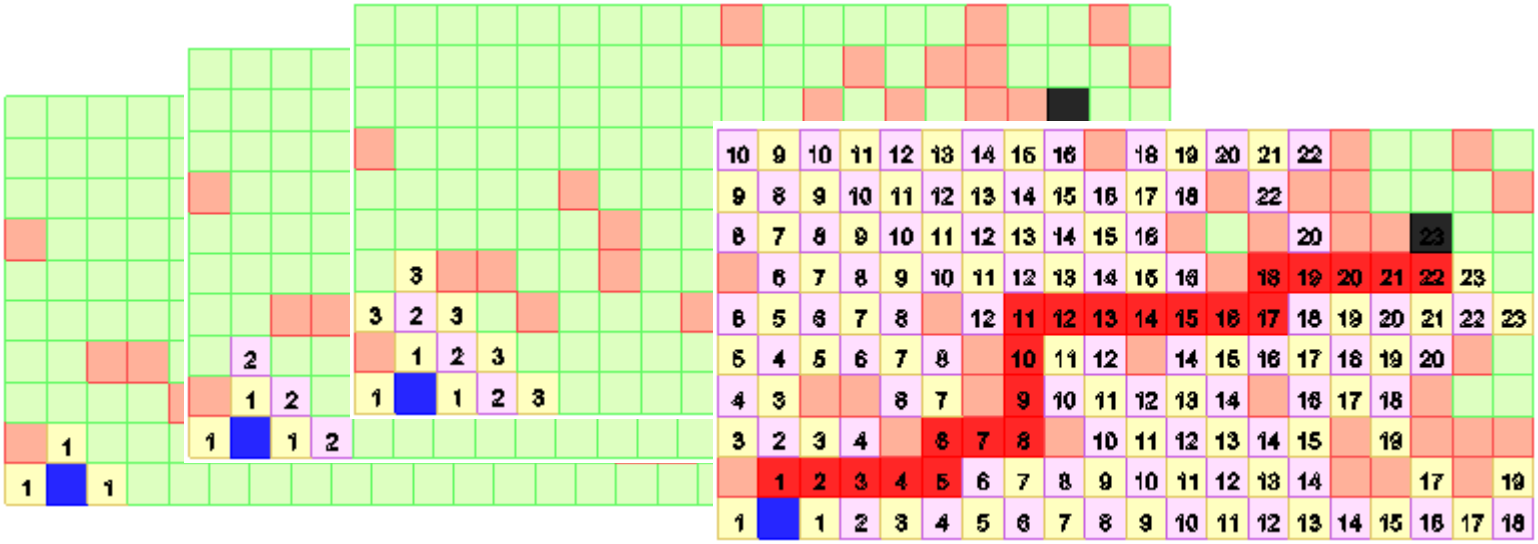
КЦ

d := d + 1

ПОКА (финишная ячейка не помечена) И (есть возможность распространения волны, шаг < количества ячеек)

Дмитрий Куратник «Реализация волнового алгоритма…»

https://habrahabr.ru/post/264189/



Реализация

```
def Wave(W, start, goal):
    def neighbors(u): return W[u]
    L_CLOSE = {}
    L_CLOSE[start] = [start, 0]
    L_OPEN = graph.TQue()
    L_OPEN.put(start)
    fScore = {}
    fScore[start] = 0
    while not L_OPEN.empty():
        current = L_OPEN.get()
        if(current==goal):
            return reconstruct_path(L_CLOSE, start, goal)
        for nb in neighbors(current) :
            if nb in L_CLOSE: continue
            if nb in L_OPEN.data: continue
            fScore[nb] = fScore[current] + W[current][nb]
            L_OPEN.put(nb)
            L_CLOSE[nb] = [current, fScore[nb]]
return None
```

```
def reconstruct_path(came_from, start, goal):
    current = goal
    path = [current]
    while current != start:
        current = came_from[current][0]
        path.append(current)
    path.reverse()
    return path
```

Список смежностей

```
W = {
    1: {2:7, 3:9, 6:14},
    2: {1:7, 3:10, 4:15},
    3: {1:9, 2:10, 4:11, 6:2},
    4: {2:15, 3:11, 5:6},
    5: {4:6, 6:9},
    6: {1:14, 3:2, 5:9}
  }
```

Пример

Поиск в ширину

```
----+-----------------------------------+----
  0 |   .  G 18 17 16 15 14 13 12 13 | 0
  1 |   . 18 17 16 15 14 13 12 11 12 | 1
  2 |   #  #  #  #  #  #  #  # 10  # | 2
  3 |   9  8  7  6  5  6  7  8  9 10 | 3
  4 |   8  7  6  5  4  5  6  7  8  9 | 4
  5 |   7  6  5  4  3  4  5  6  7  8 | 5
  6 |   6  5  4  3  2  3  4  5  6  7 | 6
  7 |   5  4  3  2  1  2  3  4  5  6 | 7
  8 |   4  3  2  1  S  1  2  3  4  5 | 8
  9 |   5  4  3  2  1  2  3  4  5  6 | 9
----+-----------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+-----------------------------------+----
```

```
----+-----------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+-----------------------------------+----
  0 |   .  G  .  .  .  .  .  .  .  . | 0
  1 |   . 18 17 16 15 14 13 12 11  . | 1
  2 |   #  #  #  #  #  #  #  # 10  # | 2
  3 |   .  .  .  .  5  6  7  8  9  . | 3
  4 |   .  .  .  .  4  .  .  .  .  . | 4
  5 |   .  .  .  .  3  .  .  .  .  . | 5
  6 |   .  .  .  .  2  .  .  .  .  . | 6
  7 |   .  .  .  .  1  .  .  .  .  . | 7
  8 |   .  .  .  .  S  .  .  .  .  . | 8
  9 |   .  .  .  .  .  .  .  .  .  . | 9
----+-----------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+-----------------------------------+----
```

Алгоритм Дейкстры

Эдскер Дейкстра, 1959. Поиск кратчайших пути от одной из вершин графа до всех остальных.

Каждой вершине сопоставляется метка – минимальное известное расстояние от этой вершины до start. На каждом шаге алгоритм «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация:

Метка вершины start полагается равной 0, метки остальных вершин – бесконечности. Все вершины графа помечаются как непосещённые.

Шаг алгоритма:

Если все вершины посещены, алгоритм завершается.

Выбрать не посещённую вершину ***u***, имеющую минимальную метку.

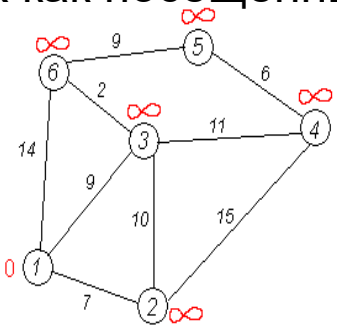
Цикл для каждого соседа n вершины ***u***, кроме отмеченных как посещённые,

L = метка(u)+длина_ребра(u, n) -- новая длина пути L

Если L < метка (n), то вес(n):= L

КЦикла

Пометить ***u*** как посещённую.



Реализация

```
def Dijkstra(G, source):
    def neighbors(u): return W[u]

    # Формируем список вершин
    L_OPEN = {}
    for e in W: L_OPEN[e] =  None

    fScore = {}
    prev = {}

    for u in L_OPEN:
        fScore[u] = Infinity
        prev[u] = u
    fScore[source] = 0
```

```
while len(L_OPEN)>0:
        # Выбираем вершину current с минимальным fScore
        minval, current = min((fScore[i], i) for i in L_OPEN)

        if current == goal:
            return fScore, prev

        # Удаляем current из L_OPEN
        del L_OPEN[current]

        for nb in neighbors(current):
            alt = fScore[current] + W[current][nb]
            if alt<fScore[nb]:
                fScore[nb] = alt
                prev[nb] = [prev[current], nb]
return fScore, prev
```

Пример

```
----+-------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+-------------------------------+----
  0 |   .  G  .  .  .  .  .  .  .  . | 0
  1 |   . 18 17 16 15 14 13 12 11  . | 1
  2 |   #  #  #  #  #  #  #  # 10  # | 2
  3 |   .  .  .  .  .  .  .  .  9  . | 3
  4 |   .  .  .  .  .  .  .  .  8  . | 4
  5 |   .  .  .  .  .  .  .  .  7  . | 5
  6 |   .  .  .  .  .  .  .  .  6  . | 6
  7 |   .  .  .  .  .  .  .  .  5  . | 7
  8 |   .  .  .  .  S  1  2  3  4  . | 8
  9 |   .  .  .  .  .  .  .  .  .  . | 9
----+-------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+-------------------------------+----
```

A*

Алгоритм поиска маршрута с наименьшей стоимостью от одной вершины к другой.

1968, Питер Харт, Нильс Нильсон, и Бертрам Рафаэль.

Порядок обхода вершин определяется эвристической функцией f(x)

$$f(x) = g(x)+h(x)$$

- g(x) – функции оценки расстояния (стоимости) от начальной вершины до рассматриваемой вершины (x)
- h(x) - функция эвристической оценки расстояния (стоимости) от рассматриваемой вершины (x) до конечной

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния до целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии. Это - расширение алгоритма Дейкстры (1959).

Реализация

```
def A_Star(G, start, goal):
    def neighbors(u): return W[u]
    # Выбираем эвристику (см. выше)
    heuristic = heuristic1
    # Список обработанных вершин
    L_CLOSE = []
    # Список вершин, подлежащих обработке
    L_OPEN = graph.TQue()
    L_OPEN.put(start)
    # Для всех вершин. Наиболее эффективно достижимые вершины из текущей.
    # Т.е. содержит наиболее эффективный предыдущий шаг.
    cameFrom = {}

    # Для всех вершин. Стоимость пути от вершины start до текущей (от start до start = 0)
    # gScore := map with default value of Infinity
    gScore = {}
    gScore[start] = 0

    # Для всех вершин. Стоимость пути от start до goal при условии, что путь проходит через текущую.
    # Стоимость частично известна, частично определяется через эвристику
    fScore = {}
    fScore[start] = heuristic(G, start, goal)
```

Реализация. Основной цикл

```
while not L_OPEN.empty():
    # Выбираем вершину current с минимальным fScore (определяет приоритет обработки вершин)
    minval, current = min((fScore[i], i) for i in L_OPEN.data)
    if current == goal: return reconstruct_path(cameFrom, current)

    L_OPEN.remove(current)
    L_CLOSE.append(current)
    for neighbor in neighbors(current) :
        # Игнорируем уже обработанные вершины
        if neighbor in L_CLOSE: continue

        # Расстояние от start до neighbor
        tentative_gScore = gScore[current] + W[current][neighbor]

        if not neighbor in L_OPEN.data: L_OPEN.put(neighbor)  # Заносим новую вершину
        else:
            if tentative_gScore >= gScore[neighbor]: continue

        # Этот путь пока наилучший
        cameFrom[neighbor] = current
        gScore[neighbor] = tentative_gScore
        fScore[neighbor] = gScore[neighbor] + heuristic(G, neighbor, goal)
```

Примеры эвристик

```
def heuristic1(G, a, b): # Хорошая эвристика, но недооцененная
    x1, y1 = G.Num2RowCol(a)
    x2, y2 = G.Num2RowCol(b)
    h = math.sqrt((x1 - x2)**2 + (y1 - y2)**2)
    return h
def heuristic2(G, a, b): # Хорошая эвристика
    x1, y1 = G.Num2RowCol(a)
    x2, y2 = G.Num2RowCol(b)
    h = abs(x1 - x2) + abs(y1 - y2)
   return h
def heuristic3(G, a, b): # Плохая (переоцененная) эвристика
    x1, y1 = G.Num2RowCol(a)
    x2, y2 = G.Num2RowCol(b)
    h = ((x1 - x2)**2 + (y1 - y2)**2)
    return h
```

Пример

```
----+--------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+--------------------------------+----
  0 |   .  G  .  .  .  .  .  .  .  . | 0
  1 |   . 18 17 16 15 14 13 12 11  . | 1
  2 |   #  #  #  #  #  #  #  # 10  # | 2
  3 |   .  .  .  .  5  6  7  8  9  . | 3
  4 |   .  .  .  .  4  .  .  .  .  . | 4
  5 |   .  .  .  .  3  .  .  .  .  . | 5
  6 |   .  .  .  .  2  .  .  .  .  . | 6
  7 |   .  .  .  .  1  .  .  .  .  . | 7
  8 |   .  .  .  .  S  .  .  .  .  . | 8
  9 |   .  .  .  .  .  .  .  .  .  . | 9
----+--------------------------------+----
    |   0  1  2  3  4  5  6  7  8  9 |
----+--------------------------------+----
```

Замечания к A*

1. A* оптимально эффективен для заданной эвристики h

2. Очередь с приоритетами
- Если вершины добавляются по принципу LIFO, то в случае вершин с одинаковой оценкой A* «пойдёт» в глубину.
- Если вершины добавляются по принципу FIFO, то для вершин с одинаковой оценкой алгоритм, напротив, будет реализовывать поиск в ширину.

3. Временная сложность – полиномиальная, если

$|h(x)-h^*(x)|<=O(\log h^*(x))$, где h^* — оптимальная эвристика

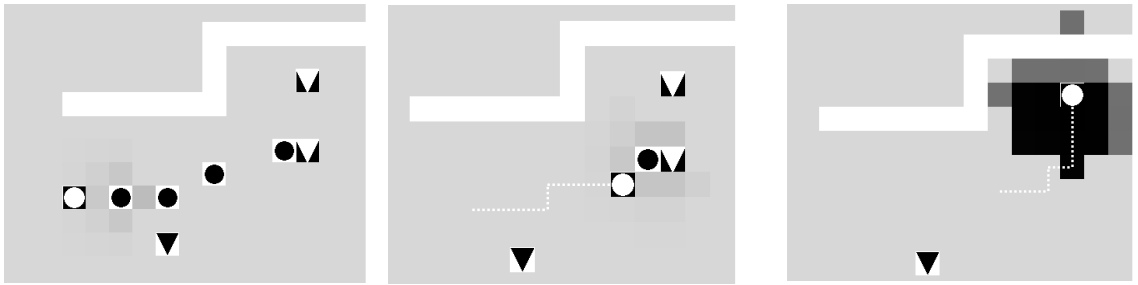
4. Ресурсы: экспоненциальное количество узлов

5. Если h(x) = 0 для всех вершин, то получится алгоритм Дейкстры.

Планирование маршрута. Метод потенциалов

- Поле F = (s1, s2, …,s_{NxM})
- Направление движения определяется направлением суммарного поля

$$\overline{E} = -\sum_{i}^{N\times M} \overline{h_i} \qquad |h_i| = \frac{\varphi_i}{{r_{i,robot}}^k}$$



О – робот

- препятствия

– ячейки с высоким потенциалом (отталкивают робота),

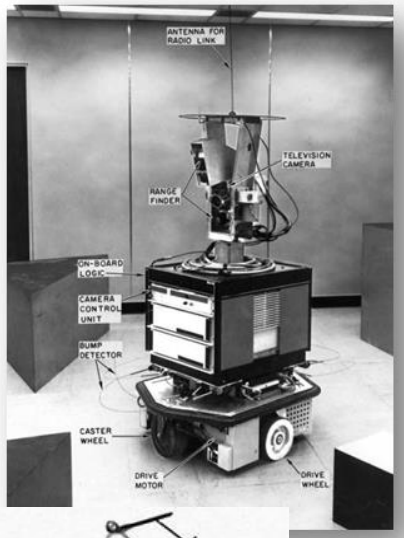
– ячейки с низким потенциалом (притягивают робота)

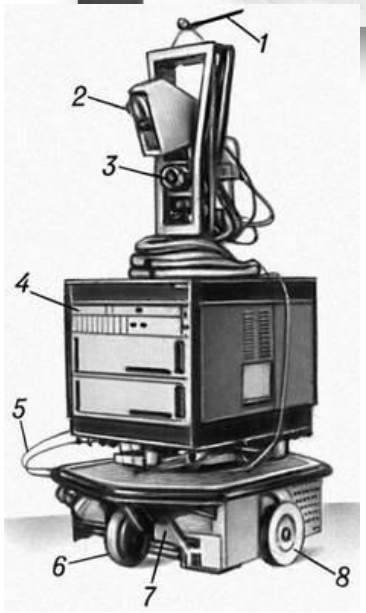
Методы планирования действий

Год	Название	Особенности
1966	GPS	«Анализ целей и средств» (АЦС).
1969	QA3	«Доказательства теорем методом резолюций» (ДТМР). Предикаты 1-го порядка.
1971	STRIPS	ДТМР в рамках только одного заданного состояния + АЦС.
1972	STRIPS + MACROP	Найденные планы обобщаются в треугольные таблицы.
1973	HACKER	АЦС + обучение по прецедентам + защита целей.
1974	InterPlan	Упорядочивания подцелей.
	ABSTRIPS	Иерархия абстрактных пространств.
	WARPLAN	Упорядочивание + регрессия целей.
1975	NOAN	Частичное упорядочивание + иерархии.
1983	SIPE	Логическое + вычислительное планирование.
1987	TWEAK	Метод накладывания ограничений.
1993	BURIDAN	Вероятностное планирование.
2000е	PDDL	Унификация всех планировщиков.

Робот Шейки

- 1969 г. Стэнфордский (SRI) интегральный робот Шейки (Shakey). Первый мобильный робот, «управляемый искусственным интеллектом».
- Бортовая ЭВМ SDS-940, телекамера, лазерный дальномер и датчики столкновения на бампере. Данныепередавались по радиоканалу на стационарные ЭВМ PDP-10 и PDP-15.
- Обратные команды передавались также по радиоканалу.
- Скорость перемещения - 1 метр в 10-15 мин.





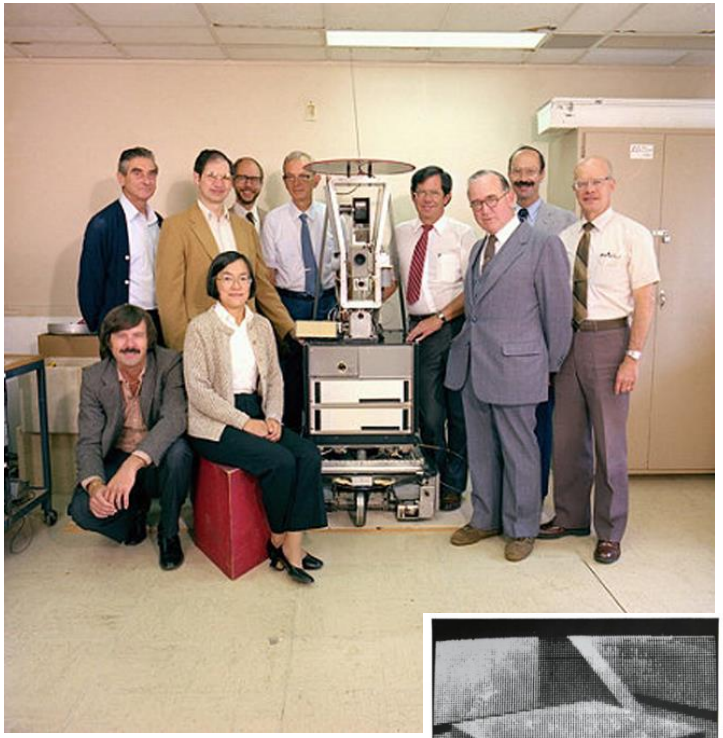
Наследие Shakey

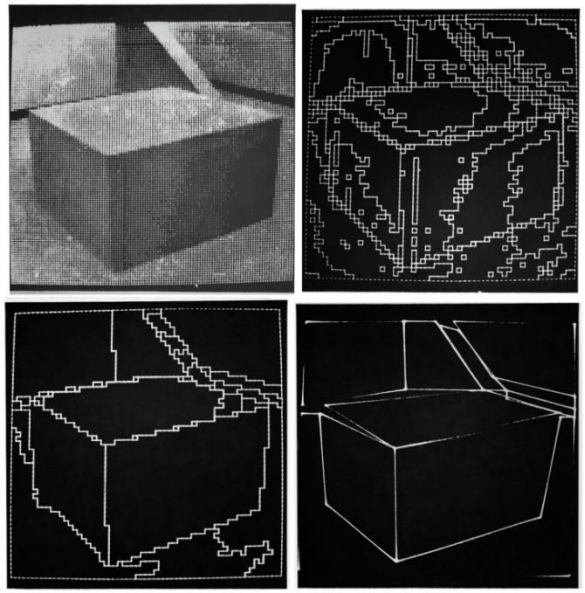
Центр Искусственного интеллекта при Стэнфордском Исследовательском институте (SRI International), 1966-1972.

DARPA: грант $750,000 ($ 5 млн сегодня).

Результаты:

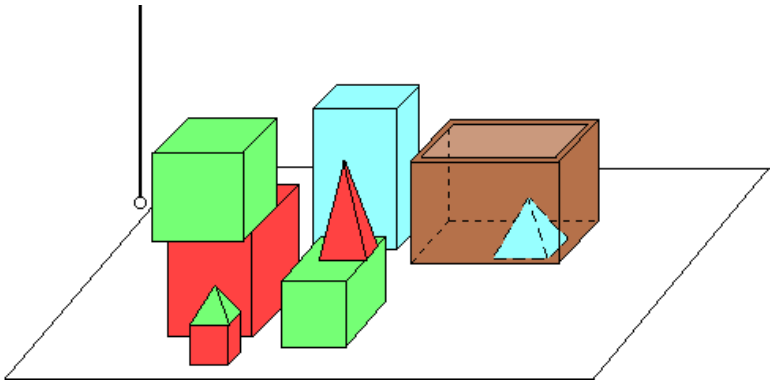
- LISP
- Преобразование Хафа (определение геометрических примитивов)
- Архитектура Shakey’s Layered Control Architecture. Stanley in DARPA grand challenge2005.
- Robust Action Execution Method. Робот PR2 (Willow Garage).
- **Adaptive Cell Decomposition.** Стандартная процедура для планирования путей (CAD, CAM - Computer-aided manufacturing).
- STRIPS Planning Systems. Основной метод планирования поведения (плоть до игр).
- Эвристический алгоритм A*: Его модификация Field D* have использовалась для навигации Curiosity на Марсе.



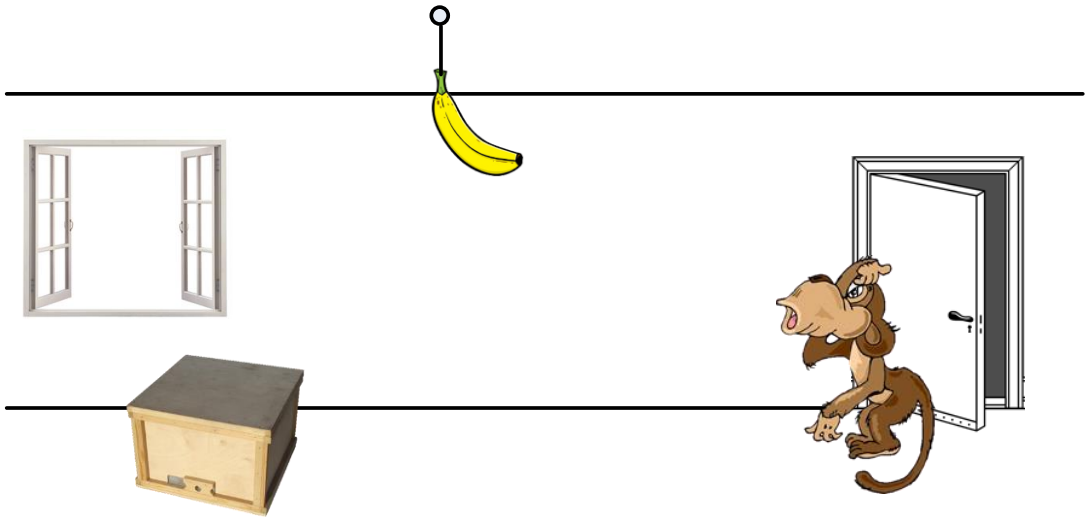


Постановка общей задачи планирования

- Планирование - это **поиск последовательности действий**, ведущих к намеченной цели.
- Целенаправленная последовательность действий не может рассматриваться в отрыве от **контекста** исполнения этих действий.



Обезьяна и банан



Братко И.
Программирование на языке Пролог для искусственного интеллекта. М., Мир. 1990. 560 с

Исходное состояние мира определяется так:
(1) Обезьяна у двери.
(2) Обезьяна на полу.
(3) Ящик у окна.
(4) Обезьяна не имеет банана.

Разрешенные ходы, переводящие мир из одного состояния в другое:

(1) схватить банан,
(2) залезть на ящик,
(3) подвинуть ящик,
(4) перейти в другое место.

Программа 1. Плохая

```
1.  /** Задача "Обезьяна и банан”**/
2.  :- dynamic is_at/2.
3.  :- dynamic monkey_is_off_box.
4.  /*----- База данных разрешенных состояний -------*/
5.  monkey_is_off_box.
6.  /* Разрешенные состояния для банана */
7.  is_at(bananas,1). is_at(bananas,2). is_at(bananas,3). is_at(bananas,4).
8.  is_at(bananas,5). is_at(bananas,6). is_at(bananas,7). is_at(bananas,8).
9.  is_at(bananas,9). is_at(bananas,10).
10. /* Разрешенные состояния для ящика */
11. is_at(box,1). is_at(box,2). is_at(box,3). is_at(box,4).
12. is_at(box,5). is_at(box,6). is_at(box,7). is_at(box,8).
13. is_at(box,9). is_at(box,10).
14. /* Разрешенные состояния для обезьяны */
15. is_at(monkey,1). is_at(monkey,2). is_at(monkey,3). is_at(monkey,4).
16. is_at(monkey,5). is_at(monkey,6). is_at(monkey,7). is_at(monkey,8).
17. is_at(monkey,9). is_at(monkey,10).
```

Продолжение

```
1.  /*---------- Правила поиска бананов ---------- */
2.  solve_the_problem :-
3.    request_position(PosBanan, PosBox, PosMonk),
4.    write("\n* Обезьяна думает: 'Я хочу взять эти бананы !'\n"),
5.    go_and_get_bananas(PosBanan, PosBox, PosMonk),
6.    write("* Обезьяна добралась до бананов.\n"),
7.    write("* ЗАДАНИЕ ВЫПОЛНЕНО.\n"),
8.    halt.

9.  /* Предикат запроса начального расположения объектов */
10. request_position(PosBanan, PosBox, PosMonk) :-
11.   write("\n\nВведите местоположение бананов (1-10) : "),
12.   %%% read(PosBanan),
13.   PosBanan = 1,
14.   write("Введите местоположение ящика (1-10) : "),
15.  PosBox = 10, %%% read(PosBox),
16.   write("Введите местоположение обезьяны (1-10): "),
17.  PosMonk = 3. %%% read(PosMonk)
```

Продолжение

```
1.  /* Основной предикат цели */
2.  go_and_get_bananas(PosBanan, PosBox, PosMonk) :-
3.    is_at(bananas, PosBanan), % Проверки допустимости позиций
4.    is_at(box, PosBox),
5.    is_at(monkey, PosMonk),
6.    monkey_works(PosBanan, PosBox, PosMonk).

7.  /* Управление поведением обезьяны */
8.  monkey_works(PosBanan, PosBox, PosMonk) :-
9.    move_to(monkey, PosBox, PosMonk),
10.   writef(' - Обезьяна переходит из позиции %d в позицию %d.\n', [PosMonk, PosBox]),
11.   move_to(box, PosBanan, PosBox),
12.   writef(' - Обезьяна передвигает ящик из позиции %d в позицию %d.\n', [PosBox, PosBanan]),
13.   climb_box,
14.   grasp_bananas.

15. move_to(monkey, PosDest, PosSrc) :- % Передвижение обезьяны
16.   is_at(monkey, PosDest),
17.   is_at(monkey, PosSrc),
18.   go_to(PosSrc, PosDest).

19. move_to(box, PosDest, PosSrc) :- % Перемещение ящика
20.   is_at(box, PosDest),
21.   is_at(box, PosSrc),
22.   push_box(PosSrc, PosDest).
```

Окончание

```
1.  go_to(Pos1, Pos2) :- % Передвижение обезьяны из Pos1 в Pos2
2.    monkey_is_off_box,
3.    retract(is_at(monkey, Pos1)),% Коррекция базы данных положений
4.    assert(is_at(monkey, Pos2)). % обезьяны
5.
6.  push_box(PosSrc, PosDest) :- % Передвинуть ящик из PosSrc в PosDest
7.    monkey_is_off_box,     % Перемещение обезьяны
8.    move_to(monkey, PosDest, PosSrc), % вместе с ящиком
9.    retract(is_at(monkey, PosSrc)),   % Коррекция
10.   retract(is_at(box, PosSrc)),      % базы данных
11.   assert(is_at(monkey, PosDest)),   % положений обезьяны
12.   assert(is_at(box, PosDest)).      % и ящика

13. climb_box :- % Забраться на ящик
14.   monkey_is_off_box,
15.   retract(monkey_is_off_box),
16.   write(" - Обезьяна забирается на ящик.\n").

17. grasp_bananas :- % Схватить бананы
18.   write(" - Обезьяна достает бананы.\n").

19. goal :-   solve_the_problem.
```

```
* Обезьяна думает: 'Я хочу взять эти бананы !'
 - Обезьяна переходит из позиции 3 в позицию 10.
 - Обезьяна передвигает ящик из позиции 10 в позицию 1.
 - Обезьяна забирается на ящик.
 - Обезьяна достает бананы.
* Обезьяна добралась до бананов.
* ЗАДАНИЕ ВЫПОЛНЕНО.
```

Корректное решение. Ходы и состояния

ход(Состояние1, М, Состояние2)

Например:

ход(состояние(Р1, наполу, В, Н),
перейти(Р1, Р2), % Перейти из Р1 в Р2
состояние(Р2, наполу, В, Н)).

В этом предложении делаются утверждения:

- выполненный ход состоял в том, чтобы "перейти из некоторой позиции Р1 в некоторую позицию Р2";
- обезьяна находится на полу, как до, так и после хода;
- ящик находится в некоторой точке В, которая осталась неизменной после хода;
- состояние "имеет банан" остается неизменным после хода.

Главный предикат

можетзавладеть(Состояние): обезьяна может завладеть бананом, находясь в состоянии **Состояние**

%%% может 1: обезьяна уже его имеет

можетзавладеть(состояние(-, -, -, имеет)).

% может 2: Сделать что-нибудь, чтобы завладеть им

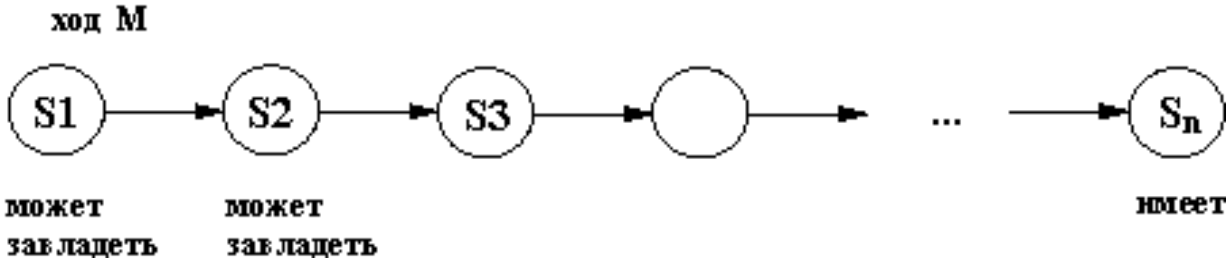
можетзавладеть(Состояние1) :-

% сделать что-нибудь

ход(Состояние1, Ход, Состояние2),

% теперь может завладеть

можетзавладеть(Состояние2).



Текст программы. Начало

```
1.  /* состояние (Позиция_обезьяны, Положение_обезьяны_относительно_пола,
2.   *           Позиция_ящика, Имение_банана)
3.   * Положение_обезьяны_относительно_пола: на_полу, на_ящике
4.   * Имение_банана: имеет, не_имеет
5.   * Позиция_обезьяны, Позиция_ящика: середина, у_двери, у_окна
6.   ***/
7.  %%% Разрешенные ходы
8.  ход( состояние( середина, на_ящике, середина, не_имеет),
9.    схватить,          % Схватить банан
10.   состояние( середина, на_ящике, середина, имеет)).
11.
12. ход( состояние( Pos, на_полу, Pos, Has),
13.   залезть,           % Залезть на ящик
14.   состояние( Pos, на_ящике, Pos, Has) ).
15.
16. ход( состояние( P1, на_полу, P1, Has),
17.   подвинуть( P1, P2), % Подвинуть ящик с P1 на P2
18.   состояние( P2, на_полу, P2, Has) ).
19.
20. ход( состояние( Pos1, на_полу, PosBox, Has),
21.   перейти( Pos1, Pos2),   % Перейти с Pos1 на Pos2
22.   состояние( Pos2, на_полу, PosBox, Has) ).
```

Продолжение

```
% может 1: Обезьяна уже его имеет
можетзавладеть( состояние( _, _, _, имеет) ).

 % может 2: Сделать что-нибудь, чтобы завладеть им
можетзавладеть( Состояние1) :-
  % сделать что-нибудь
  ход( Состояние1, Ход, Состояние2),
  write(Состояние1),   write(" -- "),   write(Ход), write(" --> "),  write(Состояние2), nl,
  % теперь может завладеть
  можетзавладеть( Состояние2).

goal :-
  можетзавладеть( состояние( у_двери, на_полу, у_окна, не_имеет) ),
  %можетзавладеть( состояние( у_окна, на_полу, у_окна, не_имеет) ),
  %можетзавладеть( состояние( у_двери, на_полу, середина, не_имеет) ),
  %можетзавладеть( состояние( середина, на_полу, середина, не_имеет) ),
  write("OK"),   halt.
goal :-   write("Failure"),  halt.
```

Примеры работы

1. можетзавладеть(состояние(у_двери, на_полу, у_окна, не_имеет))

состояние(у_двери,на_полу,у_окна,не_имеет) - перейти(у_двери,_G1057) -> состояние(_G1057,на_полу,у_окна,не_имеет)
состояние(у_окна,на_полу,у_окна,не_имеет) - залезть -> состояние(у_окна,на_ящике,у_окна,не_имеет)
состояние(у_окна,на_полу,у_окна,не_имеет) - подвинуть(у_окна,_G1071) -> состояние(_G1071,на_полу,_G1071,не_имеет)
состояние(_G1071,на_полу,_G1071,не_имеет) - залезть -> состояние(_G1071,на_ящике,_G1071,не_имеет)
состояние(середина,на_ящике,середина,не_имеет) - схватить -> состояние(середина,на_ящике,середина,имеет)

2. можетзавладеть(состояние(у_окна, на_полу, у_окна, не_имеет))

состояние(у_окна,на_полу,у_окна,не_имеет) - залезть -> состояние(у_окна,на_ящике,у_окна,не_имеет)
состояние(у_окна,на_полу,у_окна,не_имеет) - подвинуть(у_окна,_G1057) -> состояние(_G1057,на_полу,_G1057,не_имеет)
состояние(_G1057,на_полу,_G1057,не_имеет) - залезть -> состояние(_G1057,на_ящике,_G1057,не_имеет)
состояние(середина,на_ящике,середина,не_имеет) - схватить -> состояние(середина,на_ящике,середина,имеет)

3. можетзавладеть(состояние(у_двери, на_полу, середина, не_имеет))

состояние(у_двери,на_полу,середина,не_имеет) - перейти(у_двери,_G1057) ->
состояние(_G1057,на_полу,середина,не_имеет)
состояние(середина,на_полу,середина,не_имеет) - залезть -> состояние(середина,на_ящике,середина,не_имеет)
состояние(середина,на_ящике,середина,не_имеет) - схватить -> состояние(середина,на_ящике,середина,имеет)

4. можетзавладеть(состояние(середина, на_полу, середина, не_имеет))

состояние(середина,на_полу,середина,не_имеет) - залезть -> состояние(середина,на_ящике,середина,не_имеет)
состояние(середина,на_ящике,середина,не_имеет) - схватить -> состояние(середина,на_ящике,середина,имеет)

Контекст и модель мира

- Модель контекста (или внешних условий), в котором выполняются действия, называется **моделью мира** (world model).
- Модель мира может включать статическую и динамическую составляющую. Статические элементы модели мира: набор моделей *объектов* мира и *отношений* между ними.

Описание состояния в мире кубиков

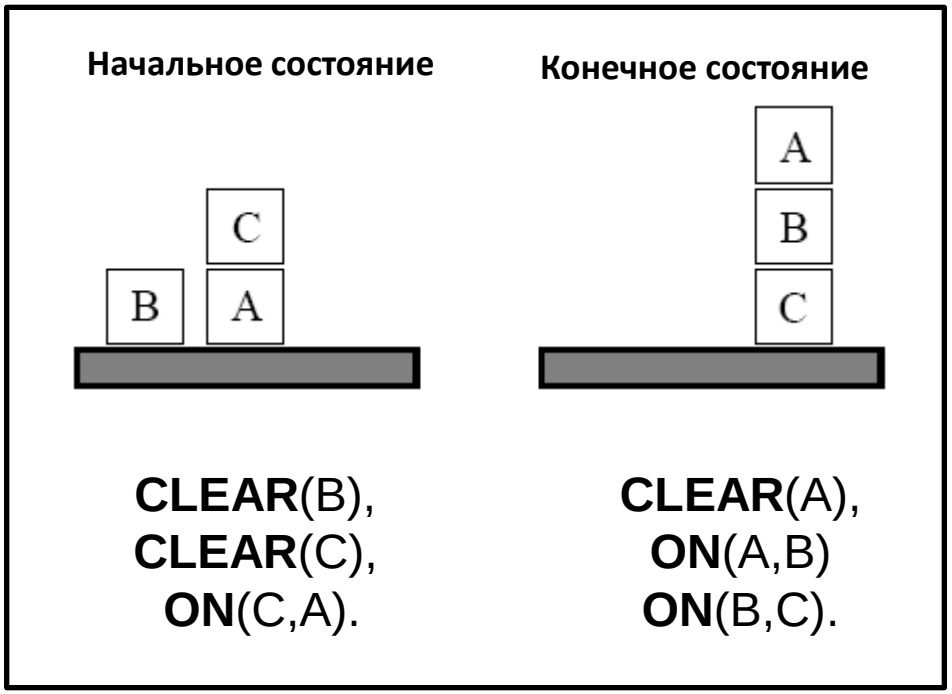
Предикаты:

CLEAR — “верхняя грань кубика свободна",
ON - "находится на кубике”

Действия:

move(Block,From,To)

can(move(Block, From, To), Preconditions),
adds(move(Block, From, To), Additions),
deletes(move(Block, From, To), Deletes).



Действие

- Конструктивным элементом для процесса планирования является **действие. Действие** определяет, какие изменения в модели мира произойдут, если действие будет выполнено.
- В качестве модели действия в интеллектуальном планировании используется сущность, описывающая:
  - вид деятельности (просто название),
  - условия, когда это действие выполнимо,
  - и эффекты, которые это действие производит.

Действие :

<**Name**, **Precondition**, **Effect**>

- **Name** — это имя действия
- **Precondition** — предусловие действия — те условия, при которых возможно выполнение действия
- **Effect —** эффект действия — описание изменений, которое действие производит.

Задача планирования

- **П**оиск последовательности действий, применение которой в начальном состоянии модели мира, приведет к такому состоянию, в котором достигается заранее заданная цель.
- Последовательность действий, полученная в результате решения задачи планирования, называется **планом**.
- Программное средство, осуществляющее решение задачи планирования, называется **планировщиком**.

General Problem Solver (GPS)

Алан Ньюэл, Герберт Саймон, 1957

- Искать решение в первую очередь в "наиболее перспективных" ветвях поиска.
- Методология "анализ средств и целей" (means-ends analysis): сначала отыскивалось различие между текущим объектом и объектом, который мы хотим получить. Это различие относилось к одному из ряда классов различий. С каждым классом был сопоставлен **набор действий**, способных уменьшить различие между текущим и целевым объектами.

Поиск в GPS

- На каждом шаге поиска GPS искал различие объектов и выбирал один из релевантных операторов, который и пытался затем применить к текущему объекту.
- Поиск подходящей последовательности операторов выполнялся **в глубину** до тех пор, пока операторы оказывались применимы, а ветвь поиска "выглядела перспективной". Если ветвь поиска была бесперспективной, то выполнялся откат.
- Всегда выбирается релевантный оператор, уменьшающий различие объектов, даже если он и не применим к текущему объекту. Вместо того, чтобы отказаться от неприменимого к текущему объекту оператора, GPS **пытался преобразовать текущий объект** в объект, пригодный для применения выбранного оператора.
- Применение такой стратегии привело к появлению **рекурсивной**, целеориентированной процедуры, которая фиксирует историю поиска в графе с частично проработанными узлами.

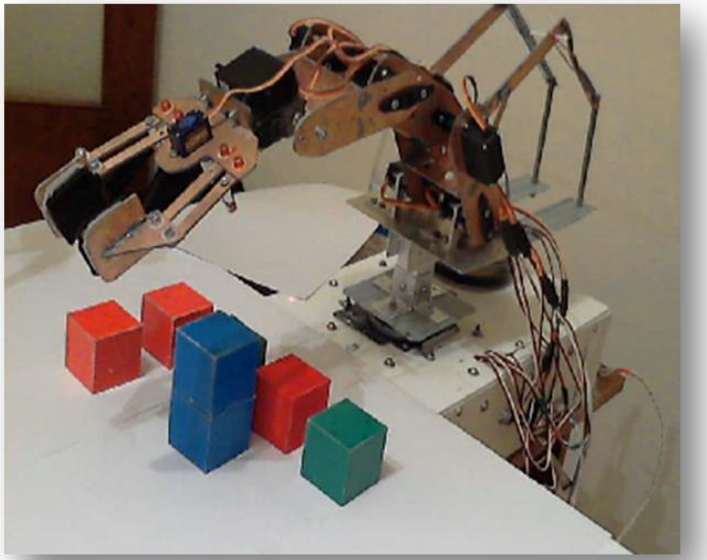
Процедура GPS

Recursive Procedure **GPS**(G)

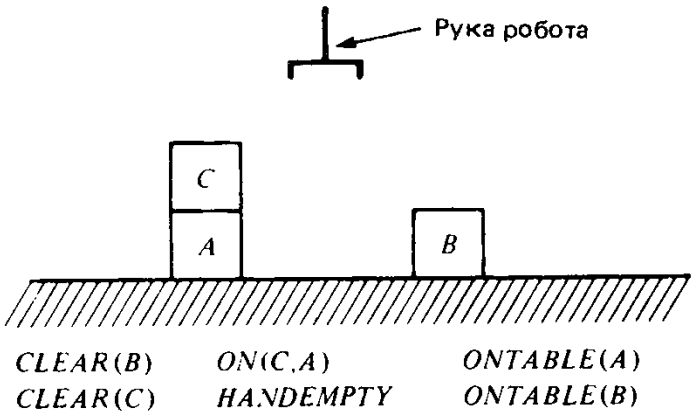
1. **until** S соответствует G, **do**: // основной цикл процедуры **GPS** является итеративным
2. **begin**
3. d ← различие между S и G // точка возврата
4. f ← П-правило, относящееся к уменьшению d // другая точка возврата
5. p ← формула предусловия соответствующего частного случая f
6. **GPS**(p) // рекурсивное обращение для решения подпроблемы
7. S ← результат применения к S соответствующего частного случая f
8. **end**

STRIPS

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver, 1971
- Ричард Файкс, Нильс Нилсон



Описания состояний и цели



Действия:

pickup - "взять со стола",

putdown - "поставить на стол",

stack - "поставить на другой кубик",

unstack - "снять с другого кубика".

Моделирование действий робота

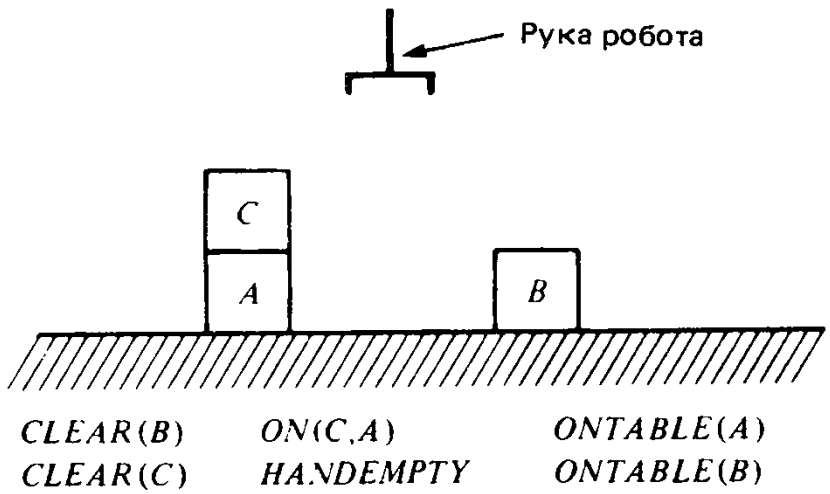
- Действия робота изменяют одно состояние (или конфигурацию) мира на другое.
- П-правило содержит три компоненты.

1. **Формула предварительного условия**, которое должно логически следовать из фактов в описании состояния для того, чтобы данное П-правило было приложимо к этому состоянию (конъюнкция литералов).
2. **Список вычеркиваний**. Когда некоторое П-правило применяется к описанию состояния, подстановка соответствия применяется к литералам списка вычеркиваний.
3. **Формула добавлений**. Она состоит из конъюнкции литералов (возможно, содержащих свободные переменные). Когда П-правило применяется к описанию состояния, то подстановка соответствия применяется к этой формуле добавлений и результирующий частный случай для этого соответствия добавляется к прежнему описанию состояния.

Пример

Взять кубик со стола.

- Предусловия: кубик находится на столе, рука робота не занята и сверху на кубике ничего нет.
- Результат - рука держит этот кубик.
- **pickup (x)**
  - Предусловие: **ONTABLE (x) Λ HANDEMPTY Λ CLEAR (x).**
  - Список вычеркиваний: **ONTABLE (x), HANDEMPTY, CLEAR (x).**
  - Формула добавлений: **HOLDING (x).**
- Новое описание состояния:
- **CLEAR (C), ON (C, A), ONTABLE (A), HOLDING (B).**

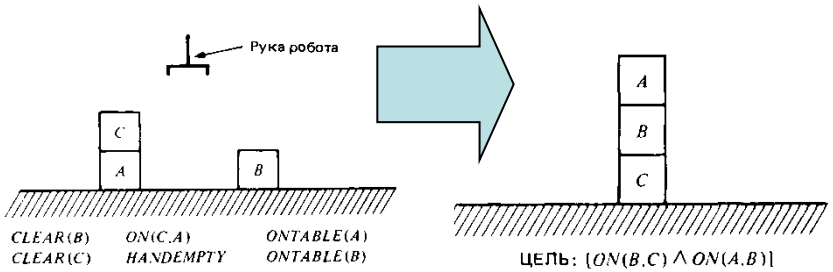


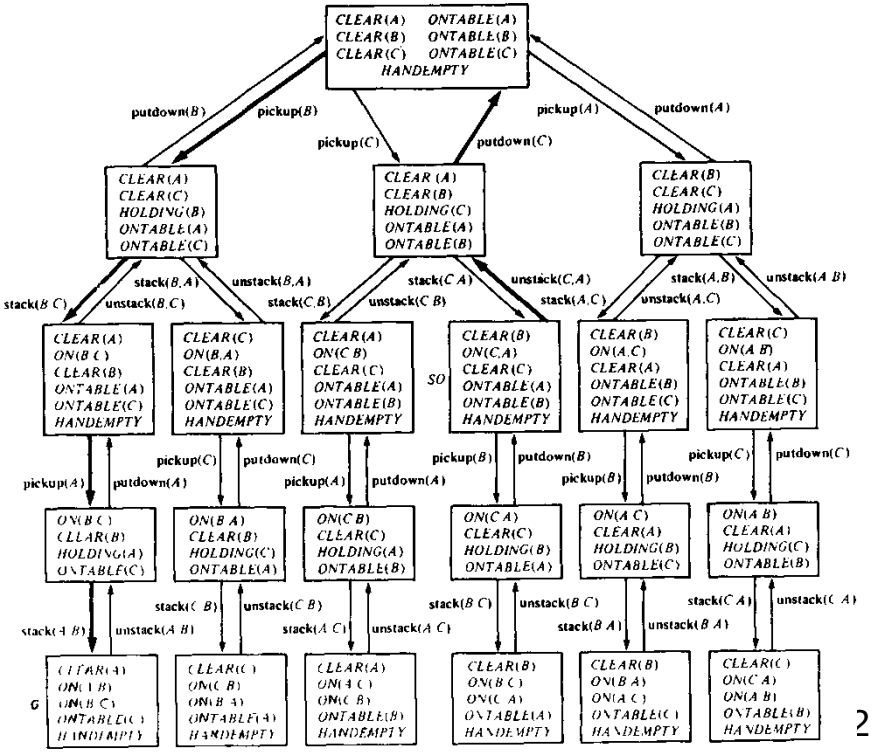
Проблема фрейма

- П-правило системы STRIPS обращается с большинством ППФ в описании состояния, как с «неизменным фоном».
- Проблема определения того, какие ППФ описания состояния следует изменять, а какие нет, в области ИИ называется **проблемой фрейма**.
- Обычно компоненты состояния мира достаточно независимы, что позволяет предполагать, что результаты действий носят относительно **локальный характер** (STRIPS-допущение).

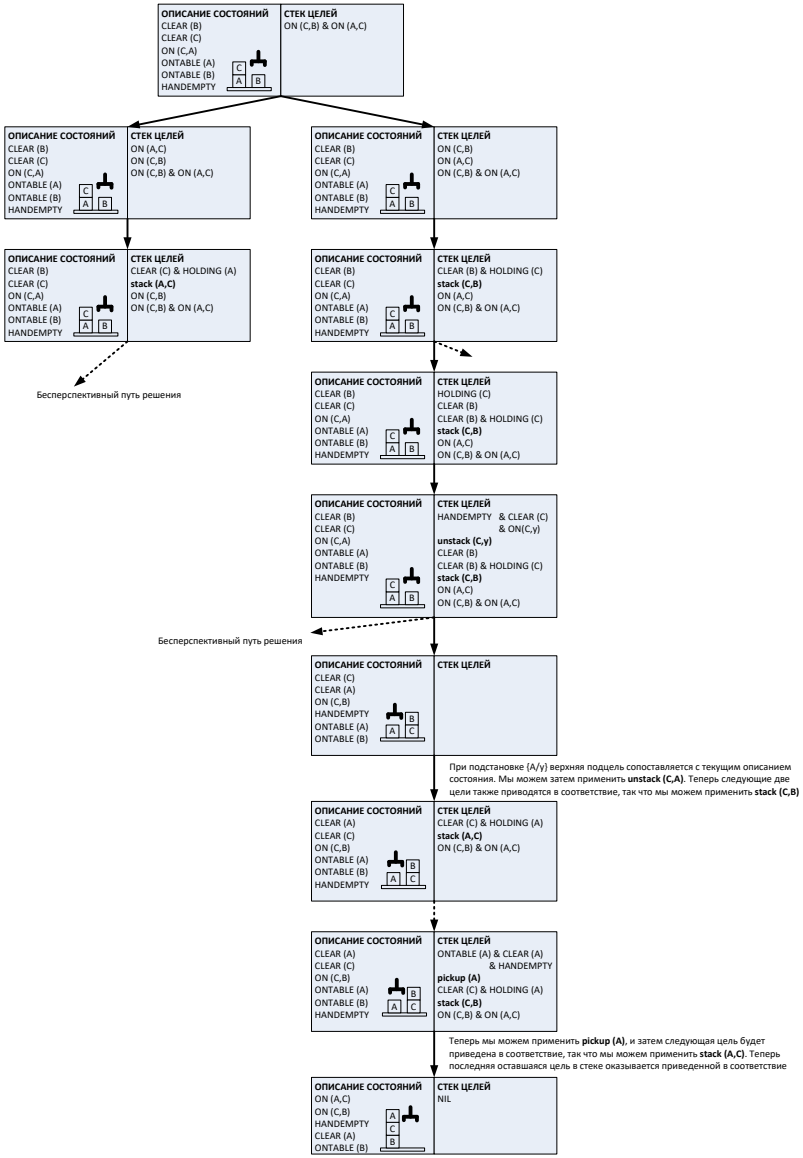
Прямая система продукций

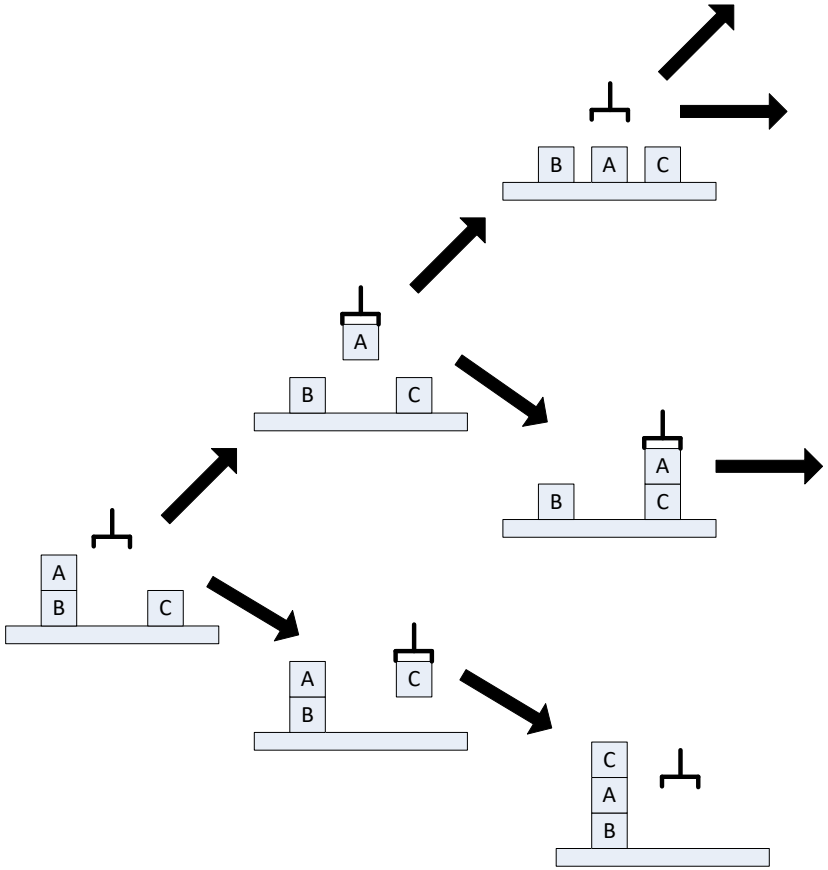
1) pickup(x)
   - P&D: ONTABLE(x), CLEAR (x), HANDEMPTY
   - A: HOLDING (x)
2) putdown(x)
   - P&D: HOLDING(x)
   - A: ONTABLE(x),CLEAR(x),HANDEMPT
3) stack(x,y)
   - P&D: HOLDING(x),CLEAR(y)
   - A: HANDEMPTY,ON(x,y),CLEAR(x)
4) unstack(x,y)
   - P&D: HANDEMPTY,CLEAR(x)ON(x,y)
   - A: HOLDING(x),CLEAR(y)





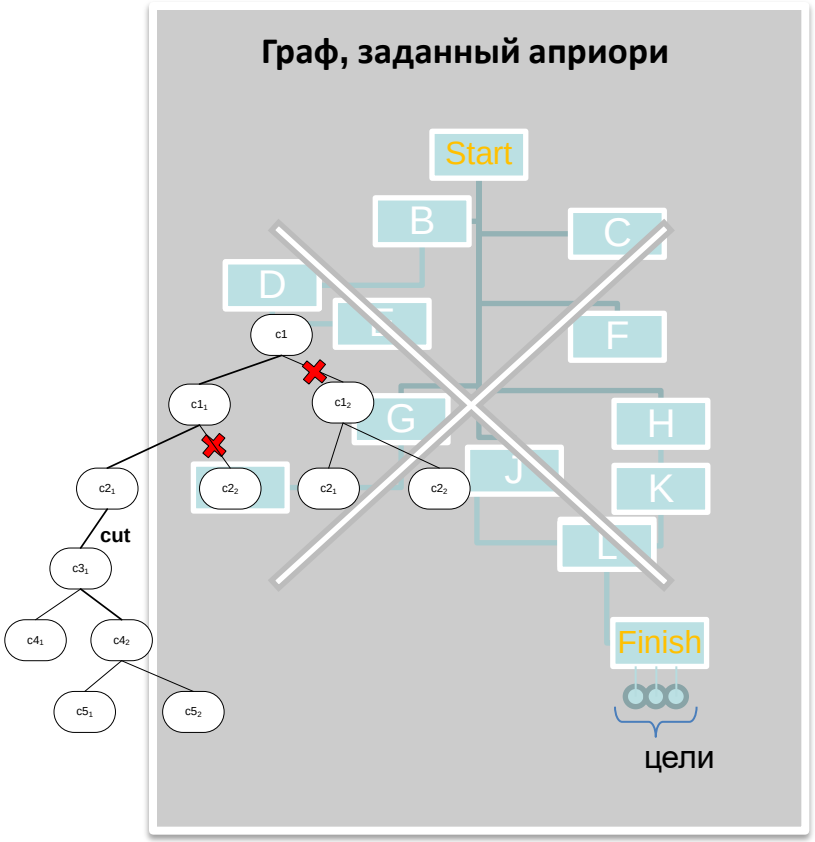
Граф поиска, созданный STRIPS

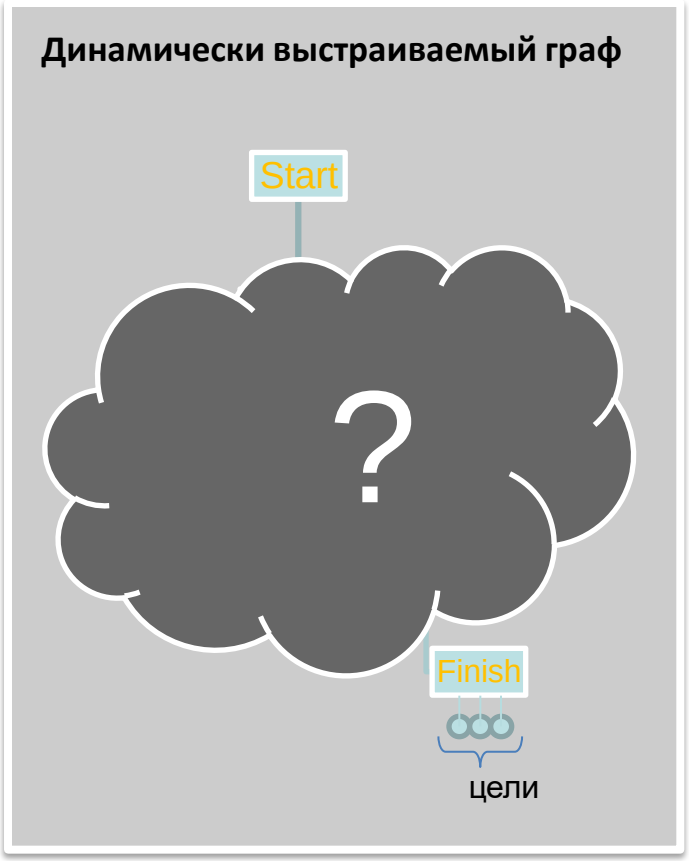




Общая структура поиска

- В начальный момент времени нет полностью построенного графа.
- Граф динамически формируется в процессе поиска.





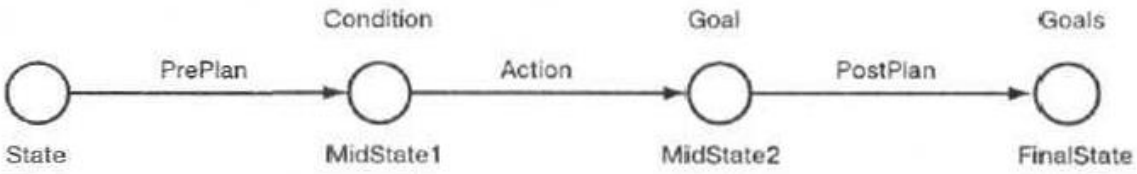
STRIPS + регрессия целей + защита целей

Поиск. Анализ целей и средств

Целеориентированный поиск «анализа целей и средств» дает заметное сужение пространства поиска.

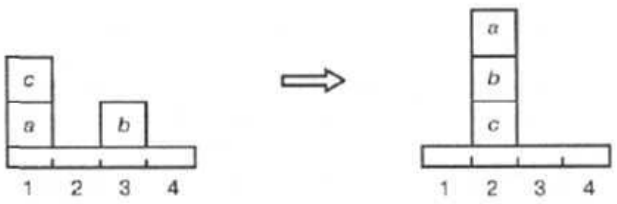
Если все цели Goals в состоянии State являются истинными, то FinalState = State. В противном случае выполнить следующие действия.

1. Выбрать в списке Goals цель Goal, для которой все еще не найдено решение.
2. Найти действие Action, которое добавляет цель Goal к текущему состоянию.
3. Обеспечить возможность выполнения действия Action, решив задачу создания предпосылок Condition действия Action, что приводит к созданию промежуточного состояния MidState1.
4. Применить действие Action к состоянию MidStatel и получить состояние MidState2 (в состоянии MidState2 цель Goal является истинной).
5. Найти решения для целей в списке Goals в состоянии MidState2, что приведет к конечному состоянию FinalState.



Защита целей

- Взаимодействующие цели могут либо неоправданно увеличить план:

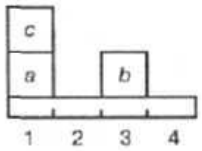


Цели: [on(b,c) ,on(a,b)]

move(b,3,c), чтобы достичь цели on(b,c)
move(b,c,3), чтобы достичь цели clear(c)
для обеспечения дальнейшего перемещения
move(c,a,2), чтобы достичь цели clear(a)
для обеспечения возможности выполнения действия move(a,1,b)
move(a,1,b), чтобы достичь цели on(a,b)
move(a,b,1), чтобы достичь цели clear(b) для обеспечения возможности выполнения действия move(b,3,c)
move(b,3,c), чтобы достичь цели on(b,c) **(повторно)**
move(a,1,b), чтобы достичь цели on(a,b) **(повторно)**

- Либо могут привести к тому, что план не будет найден вообще:

Цели: [clear(3),clear(2)]

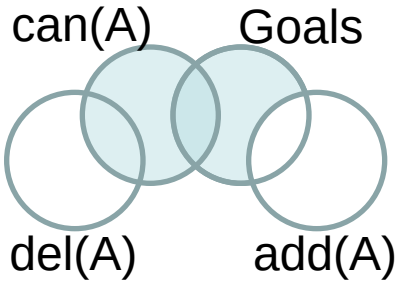


move(b,3,2), чтобы достичь цели clear(3)
move(b,2,3), чтобы достичь цели clear(2)
move(b,3,2), чтобы достичь цели clear(3)
move(b,2,3), чтобы достичь цели clear(2)
…

Планировщик должен всегда сохранять уже достигнутые цели в отдельный список и в дальнейшем избегать таких действий, которые уничтожают цели в этом списке

Регрессия целей

Регрессия помогает находить действия, которые наиболее эффективно приближают планировщика к цели (добавляют больше целей за раз).



В простом анализе целей и средств цель выбирается выбирали в отрыве от других целей.

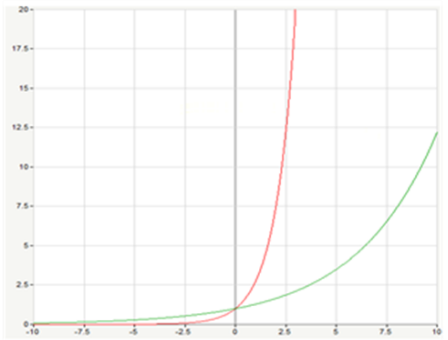
Теперь цели рассматриваются коллективно.

Чтобы достичь списка целей Goals из некоторого начального состояния StartState, необходимо выполнить следующие действия.

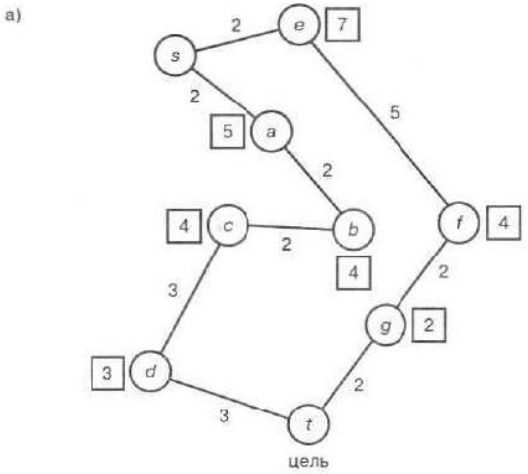
- Если в состоянии StartState все цели в списке Goals являются истинными, то достаточно применить пустой план.
- В противном случае выбрать цель G из списка Goals и некоторое действие A, в результате которого в этот список добавляется цель G. Затем выполнить регрессию целей в списке Goals с помощью действия A, получив список NewGoals, и найти план для достижения целей списка NewGoals из состояния StartState.

Дальнейшие задачи

- ОБРАТНАЯ СИСТЕМА ПРОДУКЦИЙ
- Эвристики
- Защита целей
  - планировщик иногда уничтожает цели, которые уже были достигнуты.
- Эвристики
- Поиск по заданному критерию

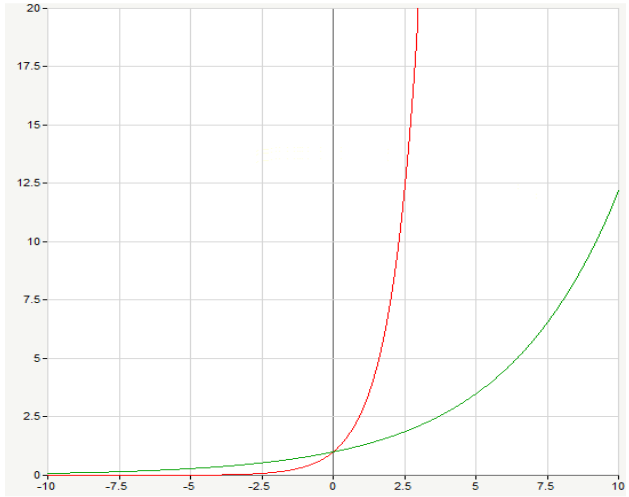


Уменьшение наклона экспоненты за счет эвристик



Эвристики

Экспоненциальная зависимость ресурсов от длины плана.
Возможен комбинаторный взрыв.
Необходимо использовать эвристики.



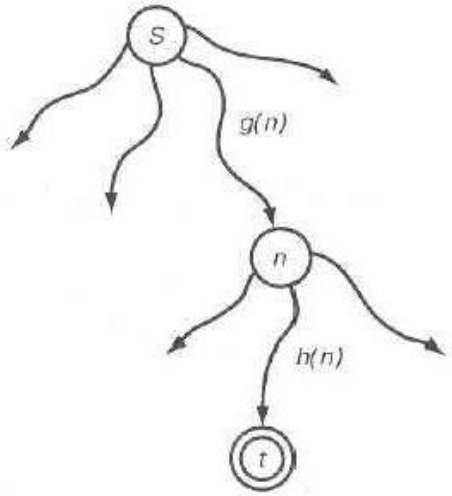


Иллюстрация эвристической функции для очередного узла

Уменьшение наклона экспоненты за счет эвристик

Эвристическая функция F:

F(n) = G(n) + H(n),

где
G(n) - пройденный путь от s до n,
H(n) - прогноз оставшегося пути от n до t.

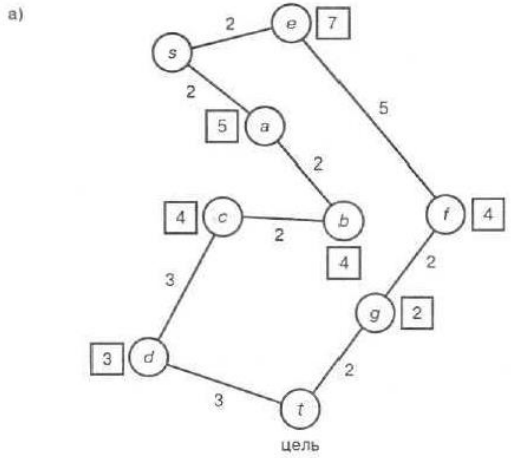
Это – эвристика алгоритма **A***

Эвристический алгоритм A*

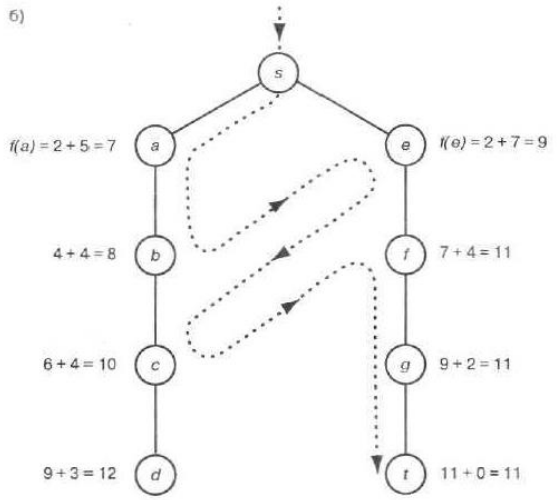
1. По графу осуществляется множественный поиск.
2. Множество процессов конкурируют за право продвигаться по графу.

3. В каждый момент времени активен только 1 процесс.

4. Активен тот процесс, у которого F(x) = G(x) + H(x) меньше.



Граф состояний



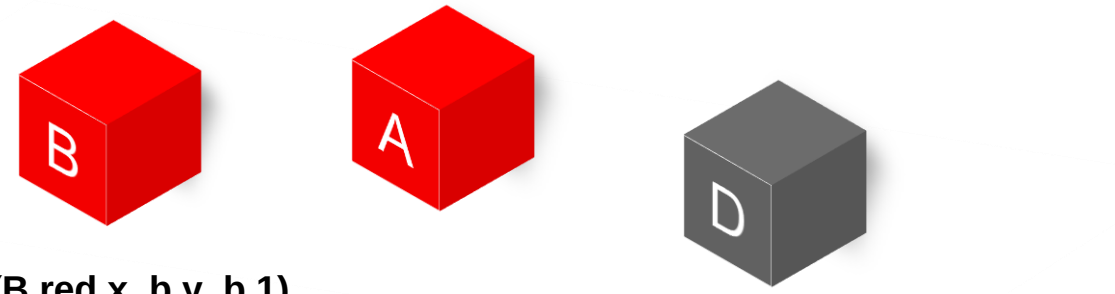
Поиск на графе состояний

Архитектура РТК

Решение задачи для реального объекта

I. Атрибуция и идентификация

В реальном мире нет априорной идентификации объектов (мы не можем оперировать просто именами объектов как это классически полагается). Необходимо производить идентификацию по группе физических атрибутов **(цвет, размер, положение в пространстве и т.д.).**



block(B,red,x_b,y_b,1)

block(A,red,x_a,y_a,1)

block(D,blue,x_d,y_d,1)

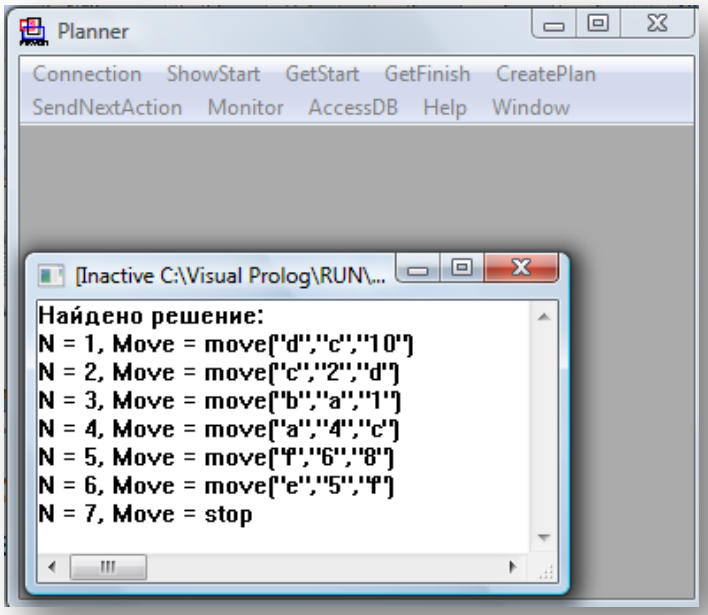
II. Устранение аномалий

Необходим механизм перестройки плана из-за возможных погрешностей в аппаратной части.

Постоянный мониторинг выполнения плана.

Реализация и работа

Планировщик написана в среде разработки **Visual Prolog**.



Особенности:

- Необходимо корректно задавать состояния и цели.
- Допустимая размерность задачи:
  8 кубиков, 12 мест, 6 целей.
- Время выполнения ~ 2 сек.

Эксперименты

Робот может достигать целей не тем способом, которым ожидается.

GOALS = (Finish) – (Start)

- Удаляется информация из **Finish** о фактах, которые совпадают с теми, которые есть в **Start.**
- Для робота это означает уменьшение ограничений на конечное состояние.
- Следовательно, робот может не следить за выполнением этих фактов в конечном состоянии.