Планирование движения

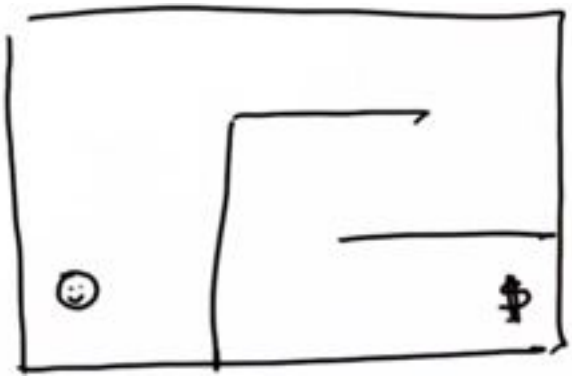
Содержание

Copyright

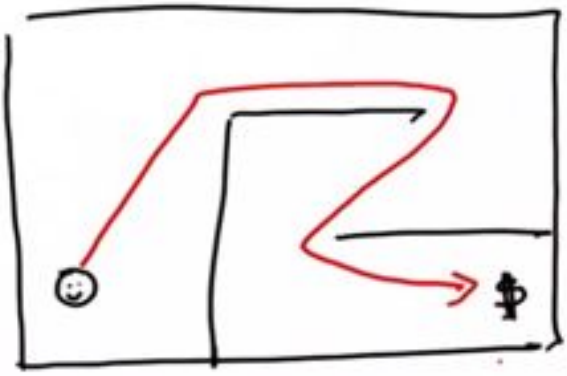
Задача планирования движения

Задача планирования движения (она же проблема навигации, англ. "motion planning", "navigation problem", "piano mover's problem") состоит в составлении детального плана движений для достижения какой-либо цели.

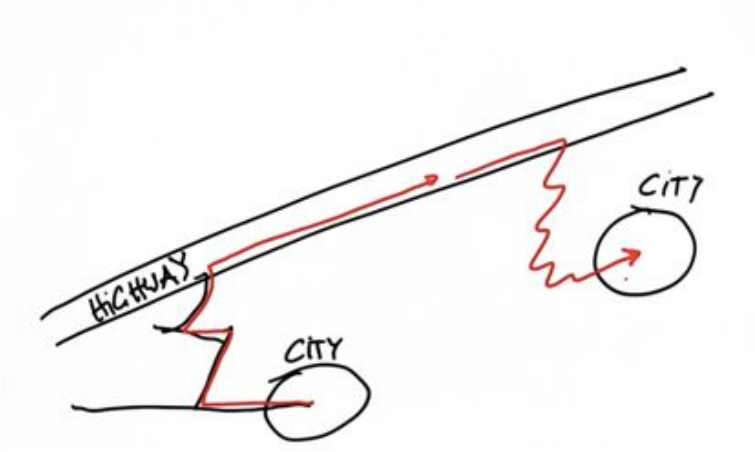
Пример
Робот функционирует в среде, показанной на рисунке, и нужно найти путь из стартовой позиции «☺» в целевую точку «$».



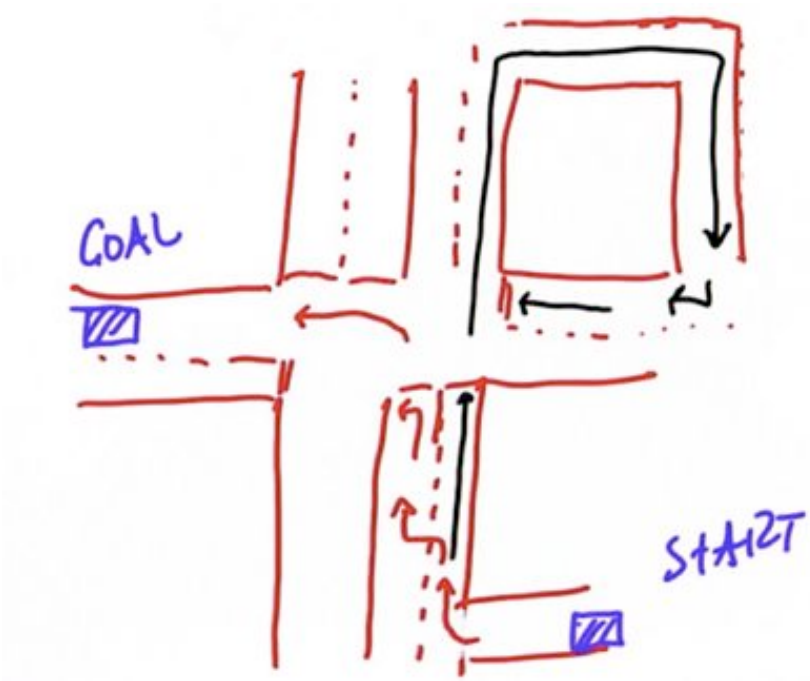
Результат – план движения может выглядеть как на рисунке:



Аналогичная проблема стоит, например, при движении автомобиля из одного города в другой по сети дорог.



Приблизимся и рассмотрим проблему навигации на перекрестке.



Автомобиль должен двигаться из точки «Start» к точке «Goal».
Для этого сначала нужно повернуть направо, затем сменить полосу движения, и затем сделать левый поворот на перекрестке, пересекая полосу встречного движения, как показано красными стрелками.
Альтернативный вариант, если, например, есть препятствие при смене полосы (длинный грузовик) – поехать в объезд, как показано черными стрелками.

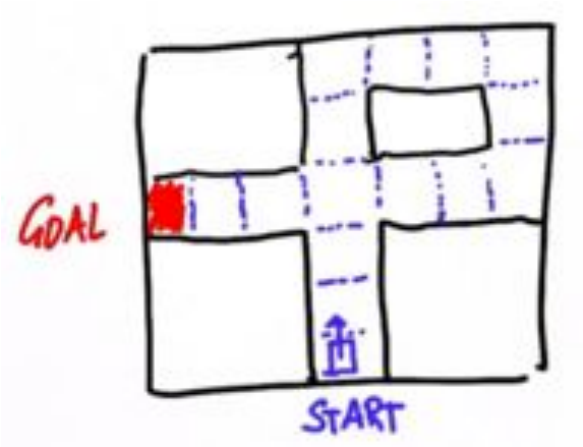
Итак, процесс приведения робота из начального состояния в целевое называем планированием движения робота.
Формально, нам дано:

- карта среды движения;
- стартовое положение;
- целевая точка;
- функция стоимости или целевая функция, например, время, которое займет движение.

Задача:
Найти путь из начального положения в целевое с минимальной целевой функцией.

Целевая функция

Рассмотрим примеры задания целевой функции.
Пусть мы работаем с дискретным миром, представленным на рисунке, т.е. мир разделен на маленькие ячейки сетки.



Начальное положение внизу, в направлении на север (вверх), обозначено «Start».
Пусть нужно добраться до положения слева, обозначенного «Goal».
Предположим, на каждом шаге робот может двигаться вперед или повернуть.
Назовем это возможными действиями (actions).
Пусть каждое действие имеет «стоимость» равную единице.
Тогда какова общая стоимость перемещения из начальной точки в заданную?

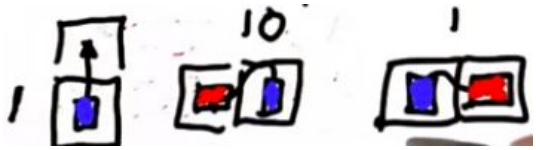
Ответ: 7. Движение по кратчайшему пути занимает 6 шагов (6 ячеек) и после третьего шага нужно повернуть налево, что также стоит единицу. Итого 6 движений вперед и один поворот дают 7.

Изменим модель движения. Пусть у нас есть 3 действия: движение вперед, поворот налево с движением вперед, поворот направо с движением вперед.

Пусть стоимость каждого действия по-прежнему равна единице. Какова будет стоимость оптимального пути к цели сейчас?

Ответ: 6. Здесь мы применяем последовательно: первое действие (движение вперед) 3 раза, затем второе действие (поворот налево с перемещение вперед) и снова движение вперед 2 раза. Таким образом, сумма равна 6, а не 7, как в предыдущем примере, когда мы рассматривали поворот на месте отдельно.

Следующий пример. Допустим, мы «наказываем» алгоритм за левые повороты, т.е. увеличиваем их стоимость.



Так поступают, например, при планировании оптимальных маршрутов в часы-пик службы доставки FedEx и UPS в США.
Зачем нам это делать? В реальных условиях левый поворот выполняется сложнее правого, т.к. зачастую нужно пропускать встречно идущий транспорт.

Пусть теперь, движение вперед «стоит» 1, левый поворот «стоит» 10, правый – 1. Каков будет оптимальный путь и его «стоимость»?

Ответ: Если вы попробовали ответить самостоятельно, то заметили, что это вопрос с подвохом, т.к. оптимальный путь не изменится, а его стоимость равна 15. Это значит, мы недостаточно «наказывали» левые повороты. Если мы совершаем действия движения вперед, вперед, вперед и влево, вперед, вперед, значение целевой функции («стоимость») равно 1 + 1 + 1 + 10 + 1 + 1 = 15. Альтернативный же путь в объезд имеет стоимость 16, в чем вы можете убедиться самостоятельно, и мы по-прежнему предпочитаем левый поворот.

Наконец, в последнем в этом параграфе примере, увеличим стоимость левого поворота. Пусть она будет равна 20.

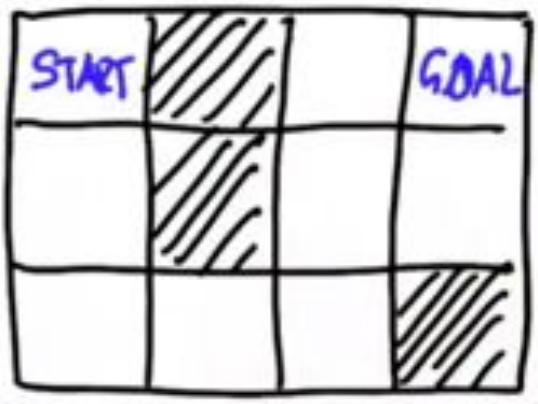
Какова сейчас общая «стоимость» движения к цели?

Ответ: 16. В этом случае, нужно двигаться по альтернативному пусти с петлей, избегая левые повороты.

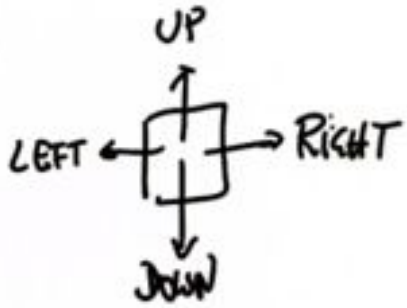
Задача планирования движения как поисковая проблема

Пример

Сначала рассмотрим пример.
Пусть мы имеем лабиринт, показанный на рисунке. Робот должен двигаться из точки «Start» к точке «Goal».

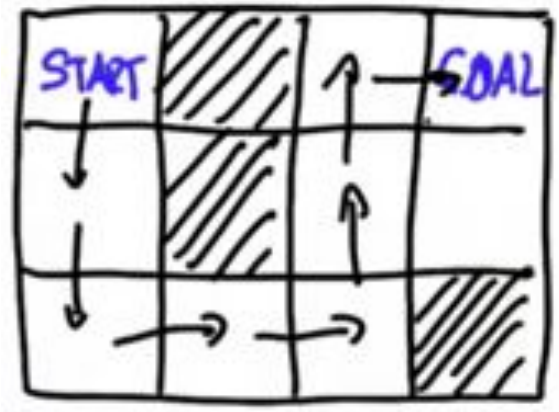


Есть множество препятствий, обозначенных штриховкой, которые робот встречает на пути.
Пусть наш робот может выполнять следующие действия: движения вверх, вниз, влево или вправо.



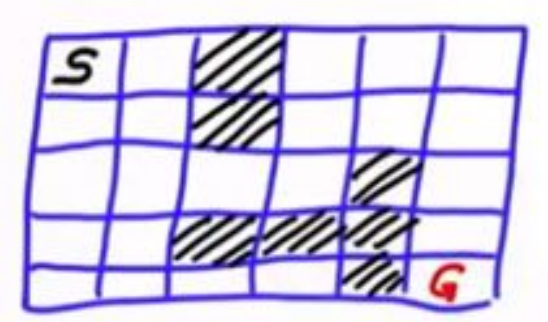
Сколько шагов займет движение от точки старта к цели?

Ответ: 7 шагов, как показано на рисунке.



Алгоритм поиска пути

Итак, рассмотрим задачу планирования движения как задачу поиска пути.
Берем для примера мир большего размера 6x5.

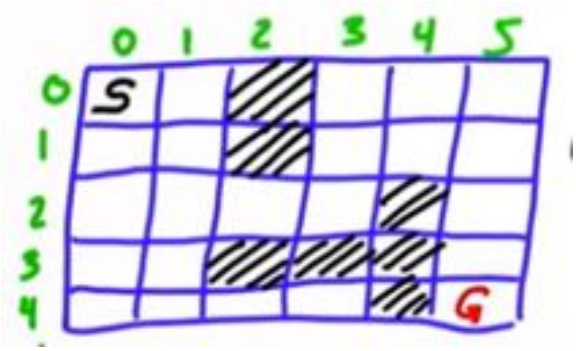


Стартовое положение «S» в верхнем левом углу, целевое положение «G» – в правом нижнем. Несколько заблокированных ячеек показаны штриховкой.
Это может быть упрощенной моделью для планирования движения в городском трафике, проезда через стоянку или лабиринт.
Робот, как и в предыдущем примере, может выполнять следующие действия: движения вверх, вниз, влево или вправо.
Задача состоит в поиске последовательности действий для перемещения робота из начального положения в заданное по кратчайшему пути.

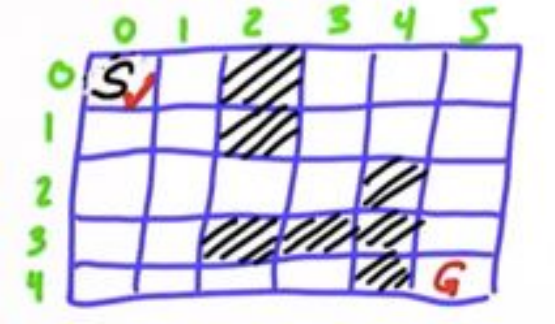
Сколько действий нужно для этого?

Ответ: 11: 2 вниз, 3 вправо, 1 вверх, 2 вправо и 3 вниз.

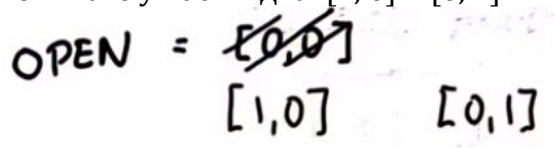
Пронумеруем столбцы и строки, как показано на рисунке.



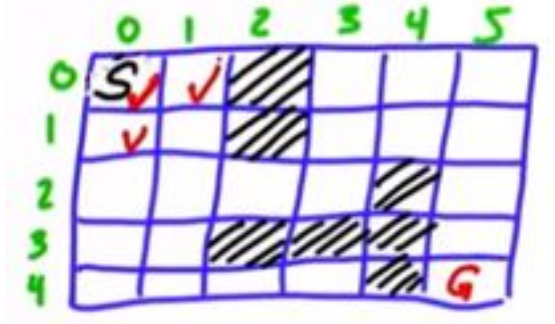
У нас 6 столбцов, пронумерованных от 0 до 5 и 5 строк от 0 до 4. Будем рассматривать каждую ячейку как узел графа.
Базовая идея состоит в создании списка узлов (ячеек), которые мы будем «исследовать» в будущем, или на которые алгоритм будет «распространяться». Назовем этот список open. Начальное состояние этого списка равно [0, 0].
Узлы, которые алгоритм уже обошел, будем помечать.



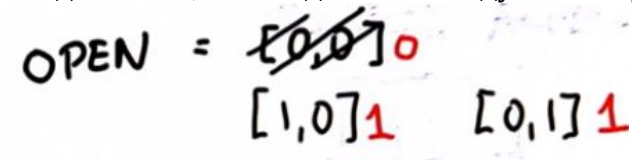
На каждом шаге алгоритма мы проверяем, является ли рассматриваемый узел целевым.
Очевидно, что на первом шаге это не так. И следующее действие алгоритма – «распространение» (expansion): берем узел из списка open, удаляем его из этого списка и рассматриваем его «наследников» – соседние узлы, в которые робот может перейти из него. На первом шаге у нас их два: [1, 0] и [0, 1].



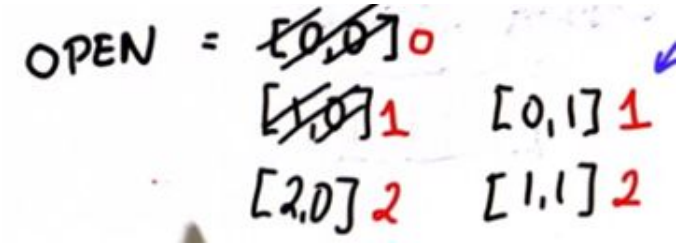
Отмечаем эти узлы как исследованные.



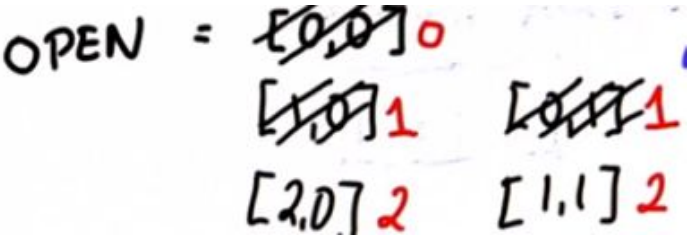
Еще одна вещь, которую должен делать алгоритм, это записать, сколько «распространений» понадобилось, чтобы дойти в каждую ячейку из списка open.



Назовем эту величину g. Когда алгоритм успешно завершится, это значение будет равно длине оптимального пути.
Движемся дальше и выполняем шаг алгоритма еще раз. Мы должны взять узел из списка open и «распространиеть» его. Чтобы путь был оптимальным, мы всегда должны брать узел с наименьшим значением g.
На данном шаге, оба узла в open имеют одинаковое значение g = 1, поэтому берем первый – [1, 0]. Он имеет 3 соседей – [0, 0], [1, 1] и [2, 0]. Но т.к. узел [0, 0] уже исследован, мы его больше не рассматриваем. И в список open добавляются узлы [1, 1] и [2, 0].



Снова выполняем шаг алгоритма. Из списка open выбираем узел с наименьшим g – это [0, 1]. Он имеет двух соседей – [0, 0] и [1, 1], но оба они уже помечены как исследованные. Поэтому «распространение» этого узла невозможно и мы его просто удаляем из списка open и на следующем шаге возьмем узел [2, 0] для «распространения».



Итак, алгоритм распространяется, пока не дойдет до целевого узла. Финальное значение g будет равно минимальному числу шагов, из начальной точки в конечную, потому что алгоритм всегда выбирает узел с наименьшим значением g для «распространения».

Программа поиска пути

Исходные данные

Реализуем описанное выше в виде программы.
Для этого зададим сетку ячеек дискретного мира, в котором функционирует робот:

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

Препятствия обозначены 1. Это та же сетка, которая была в примере.
Зададим стартовое положение:

```
init = [0, 0]
```

Зададим целевое положение (правый нижний угол):

```
goal = [len(grid)-1, len(grid[0])-1]
```

Зададим список возможных действий – движений робота:

```
delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right
```

Зададим «стоимость» каждого шага равной единице:

```
cost = 1
```

Полный код программы

```
# ----------
# User Instructions:
#
# Define a function, search() that takes no input
# and returns a list
# in the form of [optimal path length, x, y]. For
# the grid shown below, your function should output
# [11, 4, 5].
#
# If there is no valid path from the start point
# to the goal, your function should return the string
# 'fail'
# ----------

# Grid format:
#   0 = Navigable space
#   1 = Occupied space

grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1] # Make sure that the goal definition
stays in the function.

delta = [[-1, 0 ], # go up
        [ 0, -1], # go left
        [ 1, 0 ], # go down
        [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost = 1

def search():
    # two-dimensional list to check the cells once they are expanded
    closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))]
    # set the value of the starting cell to 1, since you are expanding it
    closed[init[0]][init[1]] = 1

    # Set the initial values for x, y and g and initialize open
    x = init[0]
    y = init[1]
    g = 0
    open = [[g, x, y]]
    print "initial open list:"
    print open

    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand

    # repeat the following code until you have either found the goal,
    # or found out that the goal can not be reached
    while not found and not resign:
        if len(open) == 0: # have not found the goal
            resign = True
```

```
            return "fail"
        else:
            open.sort()
            open.reverse()
            next = open.pop() # This gives us the smallest value of g
            print "take list item"
            print next
            x = next[1]
            y = next[2]
            g = next[0]
            if x == goal[0] and y == goal[1]:
                found = True
            else:
                #  for each possible move delta
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    # check if you are still within the borders of grid
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        # if the node has not yet been expanded and is
passable
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            open.append([g2, x2, y2]) # to be expanded later
                            closed[x2][y2] = 1 # not expanded again
            print "new open list:"
            print open
    return next

print search()
```

Вывод последовательности действий алгоритма

```
initial open list:
[[0, 0, 0]]
take list item
[0, 0, 0]
new open list:
[[1, 1, 0], [1, 0, 1]]
take list item
[1, 0, 1]
new open list:
[[1, 1, 0], [2, 1, 1]]
take list item
[1, 1, 0]
new open list:
[[2, 1, 1], [2, 2, 0]]
take list item
[2, 1, 1]
new open list:
[[2, 2, 0], [3, 2, 1]]
take list item
[2, 2, 0]
new open list:
[[3, 2, 1], [3, 3, 0]]
take list item
[3, 2, 1]
new open list:
[[3, 3, 0], [4, 3, 1], [4,
2, 2]]
take list item
[3, 3, 0]
new open list:
[[4, 3, 1], [4, 2, 2], [4,
4, 0]]
take list item
[4, 2, 2]
new open list:
[[4, 4, 0], [4, 3, 1], [5,
2, 3]]
take list item
[4, 3, 1]
new open list:
[[5, 2, 3], [4, 4, 0], [5,
4, 1]]
take list item
[4, 4, 0]
new open list:
[[5, 4, 1], [5, 2, 3]]
take list item
[5, 2, 3]
new open list:
[[5, 4, 1], [6, 1, 3]]
take list item
[5, 4, 1]
new open list:
[[6, 1, 3], [6, 4, 2]]
take list item
[6, 1, 3]
new open list:
[[6, 4, 2], [7, 0, 3], [7,
1, 4]]
take list item
[6, 4, 2]
new open list:
[[7, 1, 4], [7, 0, 3], [7,
4, 3]]
take list item
[7, 0, 3]
new open list:
[[7, 4, 3], [7, 1, 4], [8,
0, 4]]
take list item
[7, 1, 4]
new open list:
[[8, 0, 4], [7, 4, 3], [8,
1, 5]]
take list item
[7, 4, 3]
new open list:
[[8, 1, 5], [8, 0, 4]]
take list item
[8, 0, 4]
new open list:
[[8, 1, 5], [9, 0, 5]]
take list item
[8, 1, 5]
new open list:
[[9, 0, 5], [9, 2, 5]]
take list item
[9, 0, 5]
new open list:
[[9, 2, 5]]
take list item
[9, 2, 5]
new open list:
[[10, 3, 5]]
take list item
[10, 3, 5]
new open list:
[[11, 4, 5]]
take list item
[11, 4, 5]
```

new open list:	[]	[11, 4, 5]

Результатом работы программы является тройка [11, 4, 5], которая содержит длину оптимального пути и координаты конечной точки.

Комментарии по реализации функции search

Чтобы отмечать, какие узлы были исследованы алгоритмом, создаем массив closed, имеющий размер равный размеру мира:

```
    closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))]
```

0 – узел открыт, не исследован, 1 – закрыт, исследован.

Закрываем стартовый узел:

```
    closed[init[0]][init[1]] = 1
```

Инициализируем координаты текущей ячейки и значение целевой функции g:

```
    x = init[0]
    y = init[1]
    g = 0
    open = [[g, x, y]]
```

Задаем два флага – закончен ли поиск и есть ли возможность распространения алгоритма соответственно.

```
    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand
```

Второй флаг нужен для завершения алгоритма в случае невозможности найти путь из начальной точки в целевую.

Далее в цикле повторяем последующий код, пока не достигнута целевая точка, либо пока не исследованы все возможные узлы.

```
    while not found and not resign:
```

Если список open пустой, это значит, что невозможно найти путь к целевой точке.

```
        if len(open) == 0: # have not found the goal
            resign = True
            return "fail"
```

Иначе, нужно найти узел с наименьшим значением g в списке open. Для этого сортируем его, переворачиваем и берем узел с наименьшим g с конца списка:

```
            open.sort()
            open.reverse()
            next = open.pop()
```

Важно, что в элементах списка open первым идет значение g, по нему и выполняется сортировка.

Присваиваем переменным x, y и g новые координаты и значение целевой функции рассматриваемого для «распространения» узла:

```
            x = next[1]
            y = next[2]
            g = next[0]
```

Если достигли цели, выставляем соответствующий флаг:

```
            if x == goal[0] and y == goal[1]:
                found = True
```

Иначе производим распространение: для каждого из возможных движений робота вычисляем новые координаты

```
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
```

Проверяем, не вышли ли мы за границы сетки:

```
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
```

Если не вышли, проверяем, не закрыт ли уже данный узел и не содержит ли он препятствие:

```
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
```

Если нет, рассчитываем «стоимость» и добавляем новый узел в список open:

```
                            g2 = g + cost
                            open.append([g2, x2, y2]) # to be expanded later
                            closed[x2][y2] = 1 # not expanded again
```

Доработка программы поиска пути

Сетка распространения

Следующая задача – построить таблицу, в ячейках которой записаны номера по порядку исследования ячеек алгоритмом, описанным выше. Все ячейки, содержащие препятствия или не исследованные, должны иметь значения -1.
Пример результата работы программы:
Для сетки вида

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0]]
```

Результат имеет вид

```
[0, 1, -1, 11, 15, 18]
[2, 3, 5, 8, 12, 16]
[4, 6, -1, 13, -1, 19]
[7, 9, -1, 17, -1, 21]
[10, 14, -1, 20, -1, 22]
```

Для сетки вида

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0]]
```

Результат имеет вид

```
[0, 1, -1, -1, -1, -1]
[2, 3, -1, -1, -1, -1]
[4, 5, -1, -1, -1, -1]
[6, 7, -1, -1, -1, -1]
[8, 9, -1, -1, -1, -1]
```

Полный код программы

```
# ----------
# User Instructions:
#
# Modify the function search() so that it returns
# a table of values called expand. This table
# will keep track of which step each node was
# expanded.
#
# For grading purposes, please leave the return
# statement at the bottom.
# ----------
# Grid format:
#   0 = Navigable space
#   1 = Occupied space

grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0]]
```

```
init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1] # Make sure that the goal definition
stays in the function.

delta = [[-1, 0 ], # go up
        [ 0, -1], # go left
        [ 1, 0 ], # go down
        [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost = 1

def search():
    # two-dimensional list to check the cells once they are expanded
    closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))]
    # set the value of the starting cell to 1, since you are expanding it
    closed[init[0]][init[1]] = 1
    expand = [[-1 for row in range(len(grid[0]))] for col in
range(len(grid))]

    # Set the initial values for x, y and g and initialize open
    x = init[0]
    y = init[1]
    g = 0
    open = [[g, x, y]]
#    print "initial open list:"
#    print open
    expand[x][y] = g
    count = 0

    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand

    # repeat the following code until you have either found the goal,
    # or found out that the goal can not be reached
    while not found and not resign:
        if len(open) == 0: # have not found the goal
            resign = True
#            return "fail"
        else:
            open.sort()
            open.reverse()
            next = open.pop() # This gives us the smallest value of g
#            print "take list item"
#            print next
            x = next[1]
            y = next[2]
            g = next[0]
            expand[x][y] = count
            count += 1
            if x == goal[0] and y == goal[1]:
                found = True
            else:
                #  for each possible move delta
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    # check if you are still within the borders of grid
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
```

```
                        # if the node has not yet been expanded and is
passable
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            open.append([g2, x2, y2]) # to be expanded later
                            closed[x2][y2] = 1 # not expanded again

#            print "new open list:"
#            print open
    return expand

s = search()
for i in range(len(s)):
    print s[i]
```

Для реализации поставленной задачи создан массив expand

```
    expand = [[-1 for row in range(len(grid[0]))] for col in
range(len(grid))]
```

счетчик count

```
    count = 0
```

Массив заполняется значениями count в цикле

```
            expand[x][y] = count
            count += 1
```

Вывод пути

Наконец, получим результат планирования движения – траекторию движения и набор набор действий, которые должен совершить робот для движения к цели.
Например, для сетки вида

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

план движения должен иметь вид:

```
['>', 'v', ' ', ' ', ' ', ' ']
[' ', 'v', ' ', '>', '>', 'v']
[' ', 'v', ' ', '^', ' ', 'v']
[' ', 'v', ' ', '^', ' ', 'v']
[' ', '>', '>', '^', ' ', '*']
```

Полный код программы:

```
# ----------
# User Instructions:
#
# Modify the the search function so that it returns
# a shortest path as follows:
#
# [['>', 'v', ' ', ' ', ' ', ' '],
#  [' ', '>', '>', '>', '>', 'v'],
#  [' ', ' ', ' ', ' ', ' ', 'v'],
#  [' ', ' ', ' ', ' ', ' ', 'v'],
#  [' ', ' ', ' ', ' ', ' ', '*']]
#
# Where '>', '<', '^', and 'v' refer to right, left,
# up, and down motions. NOTE: the 'v' should be
# lowercase.
#
```

```
# Your function should be able to do this for any
# provided grid, not just the sample grid below.
# ----------
# Grid format:
#   0 = Navigable space
#   1 = Occupied space

grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1] # Make sure that the goal definition
stays in the function.

delta = [[-1, 0 ], # go up
        [ 0, -1], # go left
        [ 1, 0 ], # go down
        [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost = 1


def search():
    closed = [[0] * len(grid[0]) for i in grid]
    closed[init[0]][init[1]] = 1
    action = [[-1] * len(grid[0]) for i in grid]
    x = init[0]
    y = init[1]
    g=0
    open = [[g, x, y]]
    found = False  # flag that is set when search is complet
    resign = False # flag set if we can't find expand
    while not found and not resign:
        if len(open) == 0:
            resign = True
            return 'fail'
        else:
            open.sort()
            open.reverse()
            next = open.pop()
            x = next[1]
            y = next[2]
            g = next[0]
            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            open.append([g2, x2, y2])
                            closed[x2][y2] = 1
                            action[x2][y2] = i
    policy =  [[' '] * len(grid[0]) for i in grid]
```

```
    x = goal[0]
    y = goal[1]
    policy[x][y] = '*'
    while x != init[0] or y != init[1]:
        x2 = x - delta[action[x][y]][0]
        y2 = y - delta[action[x][y]][1]
        policy[x2][y2] = delta_name[action[x][y]]
        x = x2
        y = y2

    for row in policy:
        print row
    return policy # make sure you return the shortest path.

print search()
```

Комментарии
Создаем массив action того же размера, что и сетка мира. В каждой ячейке будем записывать действие, которое привело нас в нее. Например, для целевой ячейки в примере выше это действие – движение вниз. Для этого после строки closed[x2][y2] = 1 допишем

```
action[x2][y2] = i
```

В результате в массиве action будут содержаться действия для всех исследованных узлов. Если заменить индексы действий их символическими обозначениями, мы получим что-то похожее на:

```
[' ', '>', ' ', ' ', '>', '>']
['v', 'v', ' ', ' ', '>', '>']
['v', 'v', ' ', ' ', ' ', 'v']
['v', 'v', ' ', ' ', ' ', 'v']
['v', 'v', '>', '>', ' ', 'v']
```

Однако, это еще не совсем то, что нужно.
Однако, глядя на этот массив, можно заметить, что для каждого исследованного узла известно действие, благодаря которому алгоритм попал в него. Поэтому легко получить путь, двигаясь от целевой точки в соответствии с действиями в массиве action в обратном направлении.
Для хранения итогового плана создадим массив policy, обозначим целевую точку звездочкой:

```
    policy =  [[' '] * len(grid[0]) for i in grid]
    x = goal[0]
    y = goal[1]
    policy[x][y] = '*'
```

И заполним его, двигаясь в обратном направлении от цели, пока не достигнем стартовой точки:

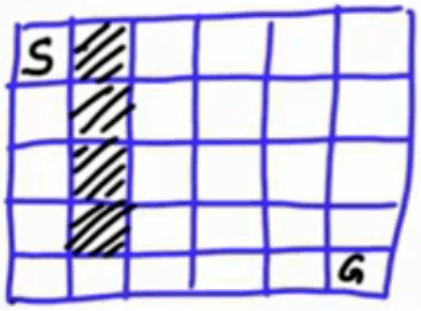
```
    while x != init[0] or y != init[1]:
        x2 = x - delta[action[x][y]][0]
        y2 = y - delta[action[x][y]][1]
        policy[x2][y2] = delta_name[action[x][y]]
        x = x2
        y = y2
```

Алгоритм A*

Отличие алгоритма

Поиск A* (произносится «А звезда» или «А стар», от англ. A star) является более эффективной модификацией рассмотренного ранее алгоритма.

Для иллюстрации разницы рассмотрим пример.
Сетка ячеек имеет вид



Или в коде

```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0]]
```

Если запустить код программы вывода сетки распространения выше, мы получим

```
  [0, -1, -1, -1, -1, -1]
  [1, -1, 12, -1, -1, -1]
  [2, -1,  9, 13, -1, -1]
  [3, -1,  7, 10, 14, -1]
  [4,  5,  6,  8, 11, 15]
```

Алгоритм распространяется вниз к узлу 4, а затем – в открытом пространстве равномерно, пока не достигнет цели. Это занимает 16 операций распространения (включая начальное).
Если мы будем использовать алгоритм A*, распространение будет иметь вид:

```
    [0, -1, -1, -1, -1, -1]
    [1, -1, -1, -1, -1, -1]
    [2, -1, -1, -1, -1, -1]
    [3, -1, -1, -1, -1, -1]
    [4,  5,  6,  7,  8,  9]
```

Заметим, что в этом случае результат достигнут за 10 шагов. Алгоритм распространяется вниз, а затем движется строго к цели.
Усложним мир – поместим препятствие перед целью:

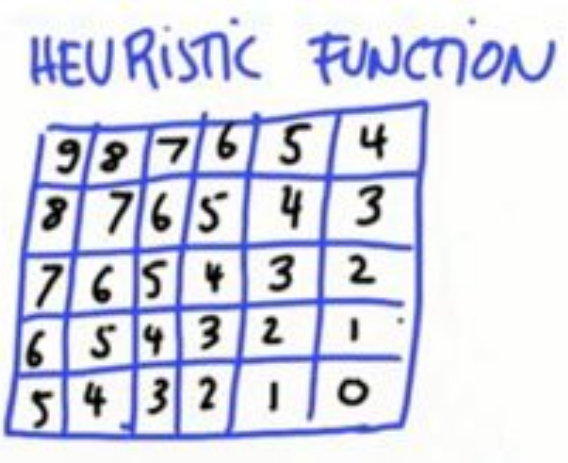
```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]
```

Запустим алгоритм A* снова:

```
    [0, -1, -1, -1, -1, -1]
    [1, -1, -1, -1, -1, -1]
    [2, -1, -1, -1, -1, -1]
    [3, -1,  8,  9, 10, 11]
    [4,  5,  6,  7, -1, 12]
```

Алгоритм, встречая препятствие распространяется вверх, а затем вправо и вниз, достигая цель.
Таким образом, он неким образом минимизируется объем работы для достижения цели максимально быстро.
Для этого алгоритм использует эвристическую функцию, это его базовый принцип.
Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других:

функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x)) и эвристической оценкой расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).
Эвристическую функцию h нужно задать. Если задать ее значения для каждой клетки нулевыми, алгоритм A* превратится в рассмотренный ранее.
Однако, если значения h будут равны числу шагов от текущей ячейки до целевой в отсутствии препятствий, алгоритм будет работать с большей эффективностью как в примерах выше.



Эвристическая функция должна быть оптимистическим предположением о числе шагов для достижения цели:

```
h(x,y) ≤ actual goal distance from x,y
```

Красота подхода состоит в том, что эвристическая функция не должна быть точной. Она должна быть функцией, которая используется для выбора направления распространения быстрее приближающего к цели.

Программа A*

Внесем модификацию в ранее созданный код вывода таблицы распространения, для реализации алгоритма A*.

Полный код:

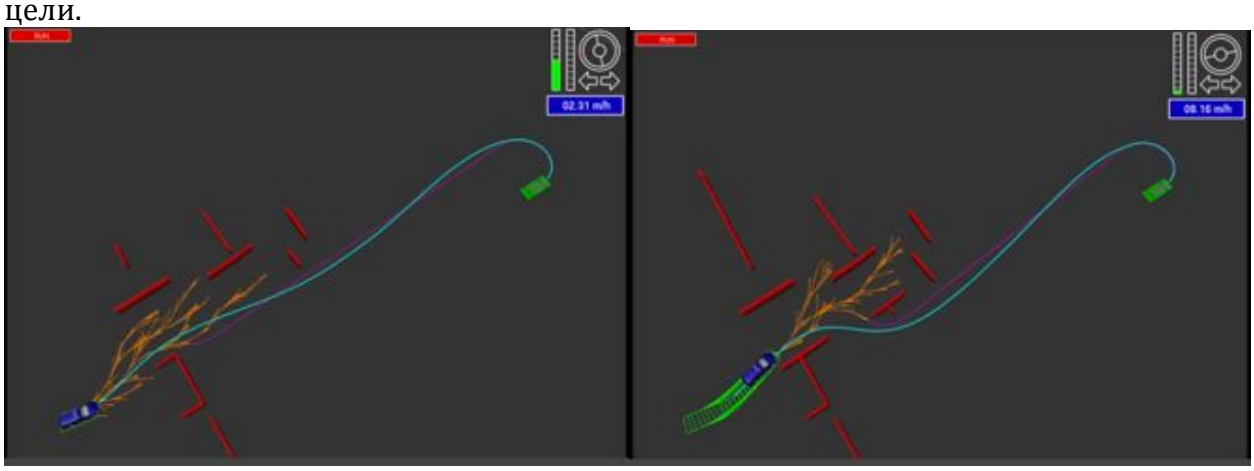
```
# ----------
# User Instructions:
#
# Modify the the search function so that it becomes
# an A* search algorithm as defined in the previous
# lectures.
#
# Your function should return the expanded grid
# which shows, for each element, the count when
# it was expanded or -1 if the element was never expanded.
# In case the obstacles prevent reaching the goal,
# the function should return "Fail"
#
# You do not need to modify the heuristic.
# ----------

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]

heuristic = [[9, 8, 7, 6, 5, 4],
```

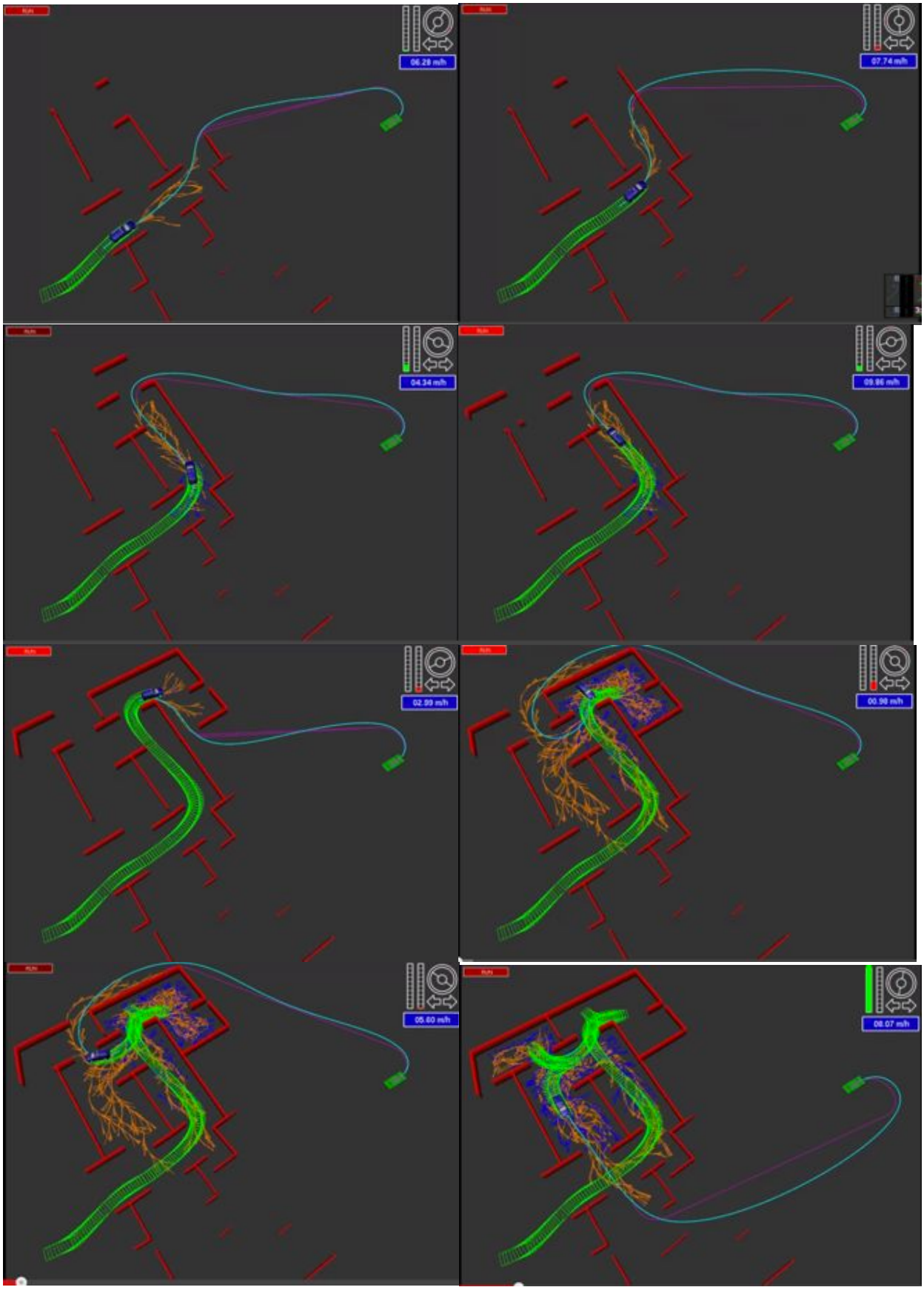
```
              [8, 7, 6, 5, 4, 3],
              [7, 6, 5, 4, 3, 2],
              [6, 5, 4, 3, 2, 1],
              [5, 4, 3, 2, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]

delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost = 1

# ----------------------------------------
# modify code below
# ----------------------------------------

def search():
    closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))]
    closed[init[0]][init[1]] = 1

    expand = [[-1 for row in range(len(grid[0]))] for col in
range(len(grid))]
    action = [[-1 for row in range(len(grid[0]))] for col in
range(len(grid))]


    x = init[0]
    y = init[1]
    g = 0

    h = heuristic[x][y]
    f=g+h
    open = [[f, g, h, x, y]]

    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand
    count = 0

    while not found and not resign:
        if len(open) == 0:
            resign = True
            return "Fail"
        else:
            open.sort()
            open.reverse()
            next = open.pop()
            x = next[3]
            y = next[4]
            g = next[1]
            expand[x][y] = count
            count += 1

            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
```

```
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            h2 = heuristic[x2][y2]
                            f2 = g2 + h2
                            open.append([f2, g2, h2, x2, y2])
                            closed[x2][y2] = 1
    for i in range(len(expand)):
        print expand[i]
    return expand #Leave this line for grading purposes!

search()
```

A* в реальном мире

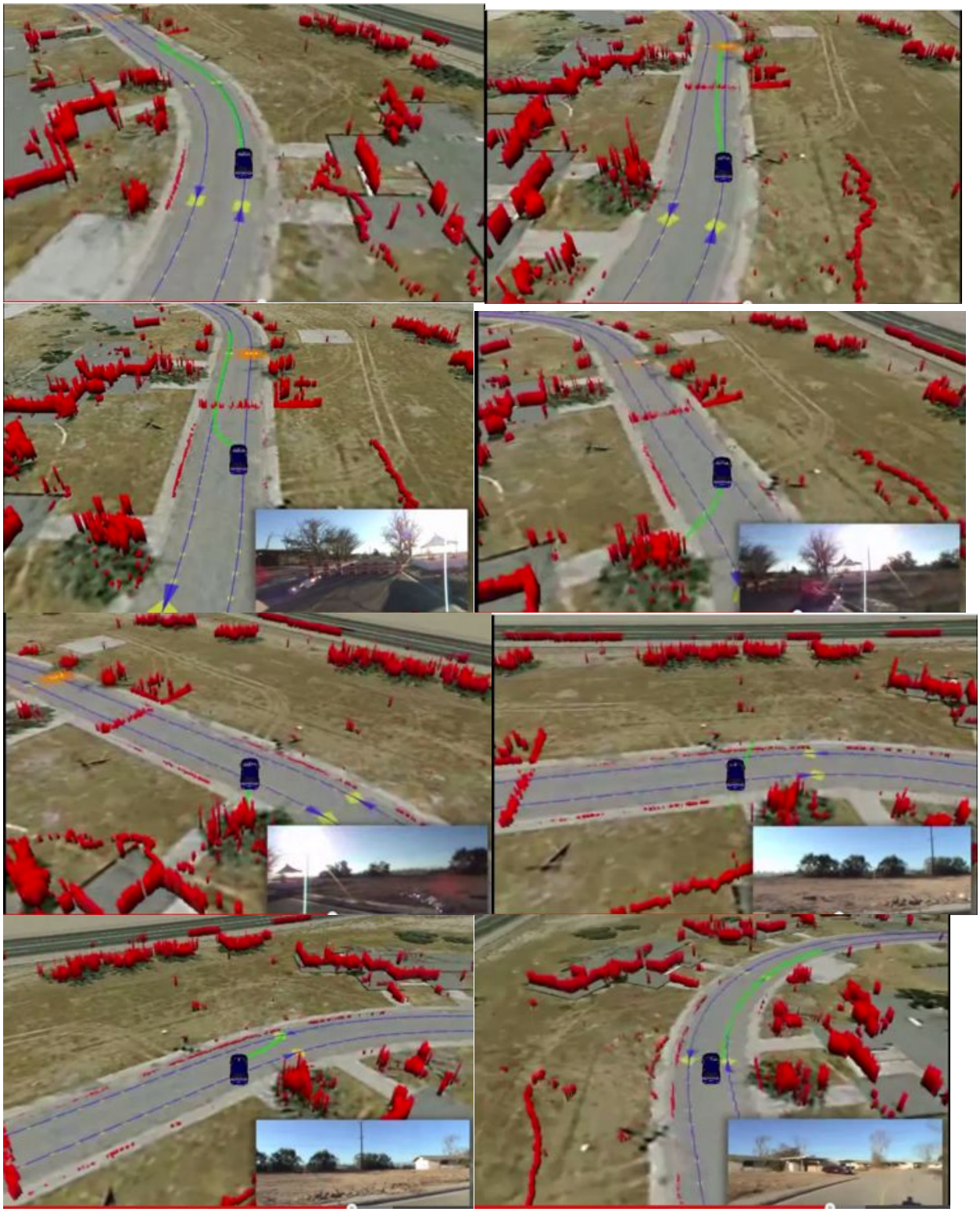
На рисунках вы можете увидеть фактическую реализацию движения робота-автомобиля Стэндфордского университета на соревнованиях DARPA Urban Challenge. Автомобиль пытается найти путь через лабиринт. Лабиринт постоянно изменяется, так как автомобиль движется и использует датчики , чтобы увидеть препятствия, когда они находятся неподалеку. Замечательно то что автомобиль в состоянии планировать очень сложные маневры движения к цели. Оранжевые деревья – визуализация дерева поиска A*.
Эта реализация настолько быстра, что робот может планировать траектории менее чем за 10 миллисекунд для любого места в этом лабиринте. Это было быстрее, чем любая другая команда на DARPA Grand или Urban Challenge.
Перепланирование происходит каждый раз, когда робот отменяет предыдущий план. На изображениях можно увидеть, что A* планирует с простой эвристической функцией на основе евклидового расстояния до цели.
Разница между этой реализацией и примером выше заключается в модели движения. Робота-автомобиль способен поворачивать, двигаться назад и вы должны написать всю математику. Но кроме этого это по сути тот же алгоритм A*.
Итак, если вы хотите построить систему управления роботом, вы теперь понимаете , как сделать действительно сложный алгоритм планирования , чтобы найти путь к цели.





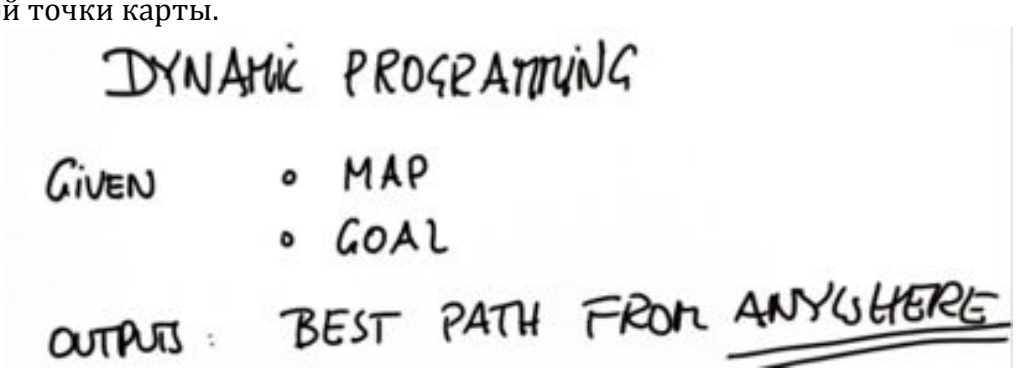
Сцена ниже демонстрирует пример самостоятельного вождения автомобиля при барьере на дороге.
Единственный способ для автомобиля для продолжения навигации это сделать несколько разворотов , и он должен был запланировать все это сам с помощью A*.

Машина подъезжает к барьеру , понимает, что нет правильного пути и вызывает планировщик A*, чтобы спланировать маневр. Автомобиль был в состоянии развернуться сам с использованием A* , найти оптимальный план , чтобы сделать это, и затем двигаться дальше. В противном случае он бы навсегда застрял перед препятствием.

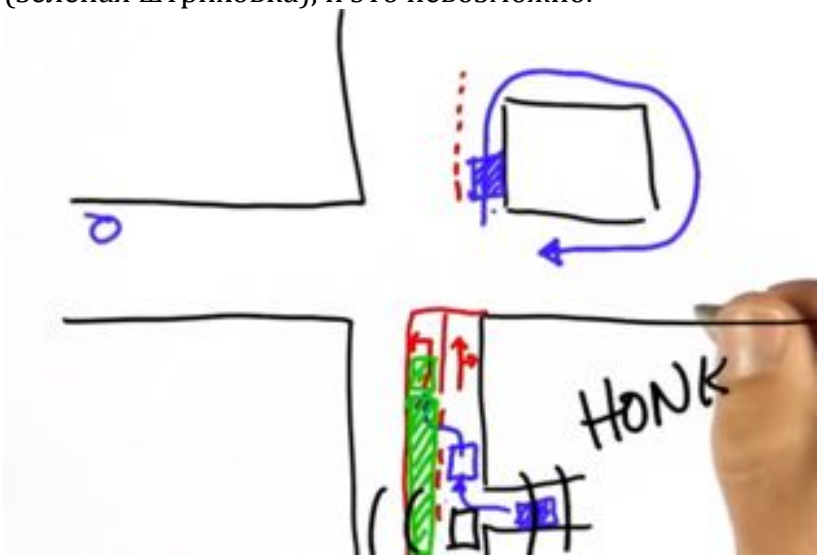


Динамическое программирование

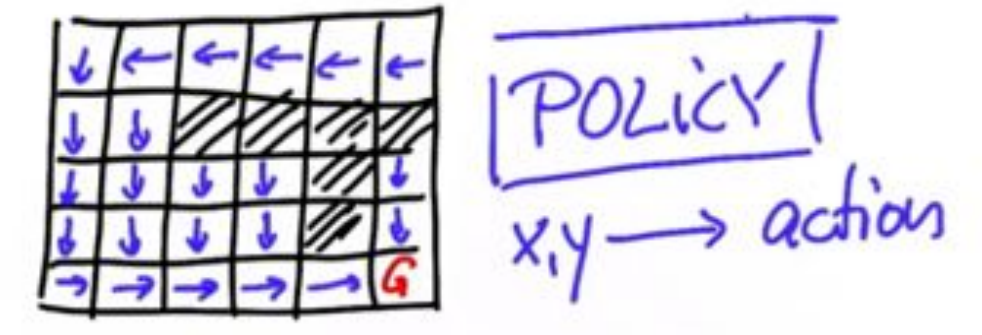
Рассмотрим альтернативный метод планирования движения – динамическое программирование.
Этот метод имеет ряд преимуществ и недостатков.
Так же как для A* дано: карта, цель, результат (в отличии от A*): оптимальный путь из любой точки карты.



Зачем это может быть нужно? Пример. Допустим робот-автомобиль находится на перекрестке, как показано на рисунке. Имеются две полосы движения – для левого поворота и движения вперед. Допустим, робот пытается сменить полосу для поворота налево. Однако, представим, что там находится длинный грузовик (зеленая штриховка), и это невозможно.

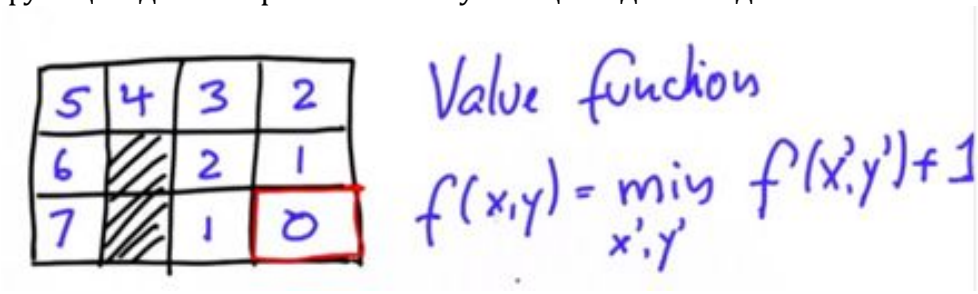


Таким образом, среда функционирования робота является стохастической и он может оказаться за перекрестком и должен иметь план не только для наиболее вероятных положений, но и для остальных. Динамическое программирование позволяет получить такой план.



Для каждой ячейки задано действие.

Начнем реализацию алгоритма динамического программирования с расчета функции длины кратчайшего пути к цели для каждой ячейки.



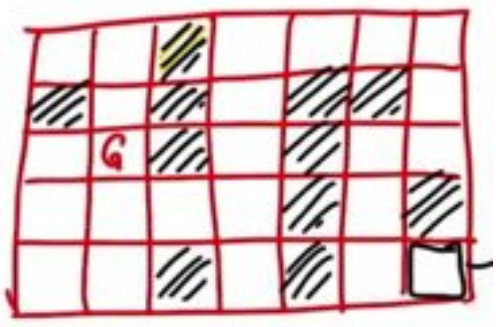
Очевидно, для целевой точки значение этой функции = 0. Для каждой соседней клетки оно равно 1. Далее – 3, 4, 5, 6, 7. Эти значения функции рассчитываются рекурсивно, путем выбора оптимального соседа x', y' (той ячейки, которая имеет минимальное значение функции длины кратчайшего пути к цели) и добавлением «стоимости» перемещения в эту ячейку. Применяя эту формулу рекурсивно, мы можем рассчитать функцию длины кратчайшего пути к цели для каждой ячейки. Имея эту функцию, мы можем рассчитать оптимальные действия (движения), минимизируя ее.

Динамическое программирование в теории управления и теории вычислительных систем — способ решения сложных задач путём разбиения их на более простые подзадачи.

Ключевая идея в динамическом программировании достаточно проста. Как правило, чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы.

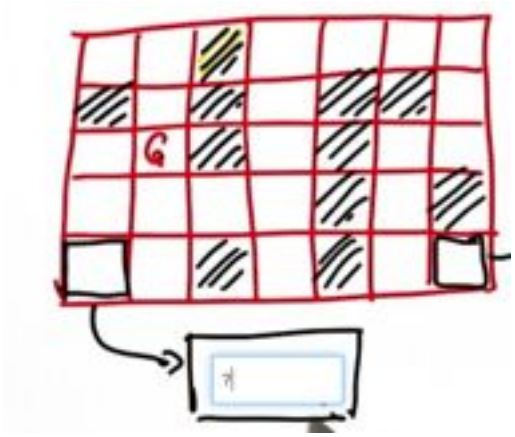
> *Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать. (с) А. Кумок.*

Пример. Каково значение функцию длины кратчайшего пути к цели для обведенной ячейки?

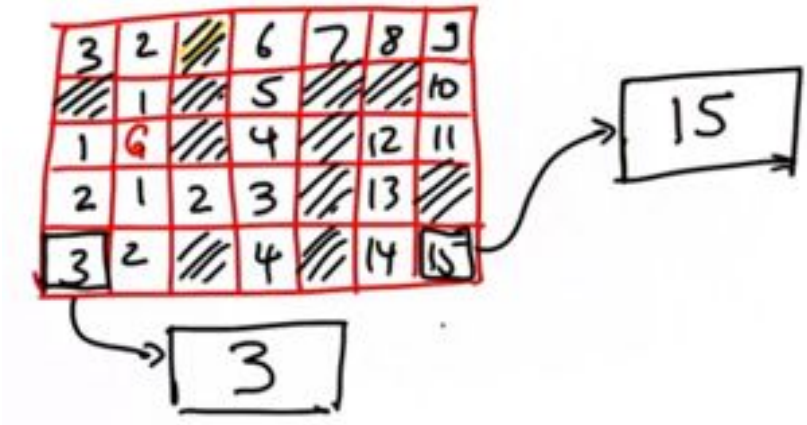


Ответ: 15

Пример. Каково значение функцию длины кратчайшего пути к цели для обведенной ячейки?



Ответ: 3



Программа планирования движения с использованием динамического программирования

Функция расчета длины кратчайшего пути к цели для всех ячеек

Реализуем функцию расчета длины кратчайшего пути к цели для всех ячеек.
Для сетки

```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]
```

Целевая точка – правый нижний угол.
Результатом должно быть

```
[11, 99, 7, 6, 5, 4]
[10, 99, 6, 5, 4, 3]
[9, 99, 5, 4, 3, 2]
[8, 99, 4, 3, 2, 1]
[7, 6, 5, 4, 99, 0]
```

Значения 99 соответствуют препятствиям.

Полный текст программы:

```
# ----------
# User Instructions:
#
# Create a function compute_value() which returns
# a grid of values. Value is defined as the minimum
# number of moves required to get from a cell to the
# goal.
```

```
#
# If it is impossible to reach the goal from a cell
# you should assign that cell a value of 99.

# ----------

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]

delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost_step = 1 # the cost associated with moving from a cell to an adjacent
one.

# ----------------------------------------
# insert code below
# ----------------------------------------

def compute_value():
    value = [[99 for row in range(len(grid[0]))] for col in range(len(grid))]
    change = True
    while change:
        change = False
        for x in range(len(grid)):
            for y in range(len(grid[0])):
                if goal[0] == x and goal[1] == y:
                    if value[x][y] > 0:
                        value[x][y] = 0
                        change = True
                elif grid[x][y] == 0:
                    for a in range(len(delta)):
                        x2 = x + delta[a][0]
                        y2 = y + delta[a][1]
                        if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                            v2 = value[x2][y2] + cost_step
                            if v2 < value[x][y]:
                                change = True
                                value[x][y] =  v2


    return value #make sure your function returns a grid of values as
demonstrated in the previous video.


v = compute_value()
for row in v:
    print row
```

Комментарии

Инициализируем массив, в который будем помещать рассчитанные значения

```
    value = [[99 for row in range(len(grid[0]))] for col in range(len(grid))]
```

Значения, которыми мы инициализируем массив, должны быть достаточно большими (больше максимальной длины пути в сетке).

Далее мы будем рекурсивно выполнять расчет значений элементов массива value. Мы не знаем, сколько раз, поэтому мы создаем флаг change, который будем устанавливать в True, когда мы рассчитаем новое значение в массиве value. Как только нам не останется, что рассчитывать, цикл завершится.

```
    change = True
    while change:
        change = False
```

Внутри цикла while проходим по всем ячейкам

```
        for x in range(len(grid)):
            for y in range(len(grid[0])):
```

Сначала проверяется, является ли рассматриваемая ячейка целевой. Если да, ее значение устанавливается в 0 (если оно еще не ноль) и устанавливается флаг изменения change.

```
                if goal[0] == x and goal[1] == y:
                    if value[x][y] > 0:
                        value[x][y] = 0
                        change = True
```

Если рассматриваемая ячейка не является целевой, запускается цикл по всем возможным движениям робота, перечисленным в списке delta.

```
                elif grid[x][y] == 0:
                    for a in range(len(delta)):
```

Рассчитываются координаты положения робота после выполнения рассматриваемого действия

```
                        x2 = x + delta[a][0]
                        y2 = y + delta[a][1]
```

Проверяется, являются ли координаты x2 и y2 допустимыми для робота – не выходят за границы сетки и не содержат препятствие

```
                        if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
```

Если новые координаты удовлетворяют условию выше, рассчитывается значение функции длин путей до цели в новой ячейке.

```
                            v2 = value[x2][y2] + cost_step
```

Если это рассчитанное значение длины пути к цели лучше (меньше) уже записанного в ячейку с координатами x2, y2, старое значение заменяется новым и устанавливается флаг внесения изменений.

```
                            v2 = value[x2][y2] + cost_step
                            if v2 < value[x][y]:
                                change = True
                                value[x][y] =  v2
```

Программа расчета плана для каждой ячейки

Доработаем программы для вывода матрицы движений робота для каждой ячейки для движения к цели.

Для сетки вида

```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]
```

результатом должно быть

```
['v', ' ', 'v', 'v', 'v', 'v']
['v', ' ', 'v', 'v', 'v', 'v']
['v', ' ', 'v', 'v', 'v', 'v']
```

```
['v', ' ', '>', '>', '>', 'v']
['>', '>', '^', '^', ' ', '*']
```

Полный текст программы:

```
# ----------
# User Instructions:
#
# Create a function optimum_policy() that returns
# a grid which shows the optimum policy for robot
# motion. This means there should be an optimum
# direction associated with each navigable cell.
#
# un-navigable cells must contain an empty string
# WITH a space, as shown in the previous video.
# Don't forget to mark the goal with a '*'

# ----------

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]

delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost_step = 1 # the cost associated with moving from a cell to an adjacent
one.

# ----------------------------------------
# modify code below
# ----------------------------------------

def optimum_policy():
    value = [[99 for row in range(len(grid[0]))] for col in range(len(grid))]
    change = True
    policy = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]

    while change:
        change = False

        for x in range(len(grid)):
            for y in range(len(grid[0])):
                if goal[0] == x and goal[1] == y:
                    if value[x][y] > 0:
                        value[x][y] = 0
                        policy[x][y] = '*'
                        change = True

                elif grid[x][y] == 0:
                    for a in range(len(delta)):
                        x2 = x + delta[a][0]
```

```
                        y2 = y + delta[a][1]

                        if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 <
len(grid[0]) and grid[x2][y2] == 0:
                            v2 = value[x2][y2] + cost_step

                            if v2 < value[x][y]:
                                change = True
                                value[x][y] = v2
                                policy[x][y] = delta_name[a]
    for i in range(len(value)):
        print policy[i]

    return policy # Make sure your function returns the expected grid.

optimum_policy()
```

Комментарии

Объявим массив, в который запишем план движения.

```
    policy = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]
```

Если рассматриваем целевую точку, запишем “*”

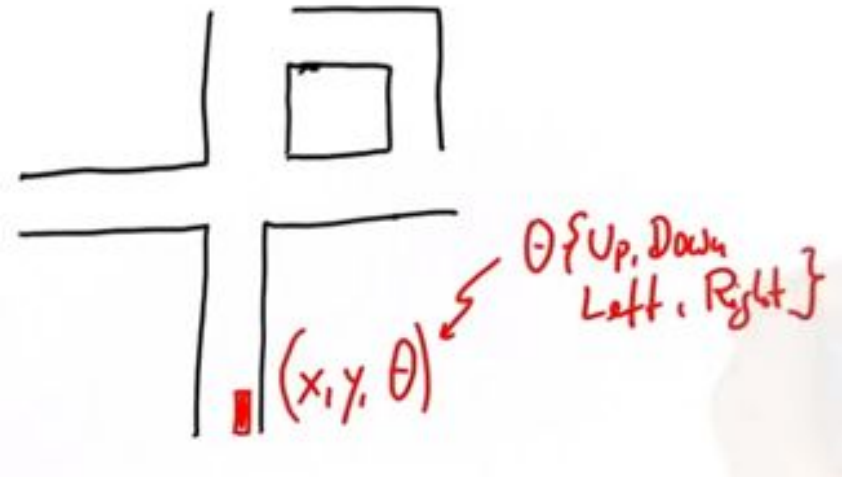
```
                    if value[x][y] > 0:
                        value[x][y] = 0
                        policy[x][y] = '*'
                        change = True
```

Когда нашли меньшее значение функции длин путей к цели для новой ячейки, записываем движение робота, которое его уменьшило

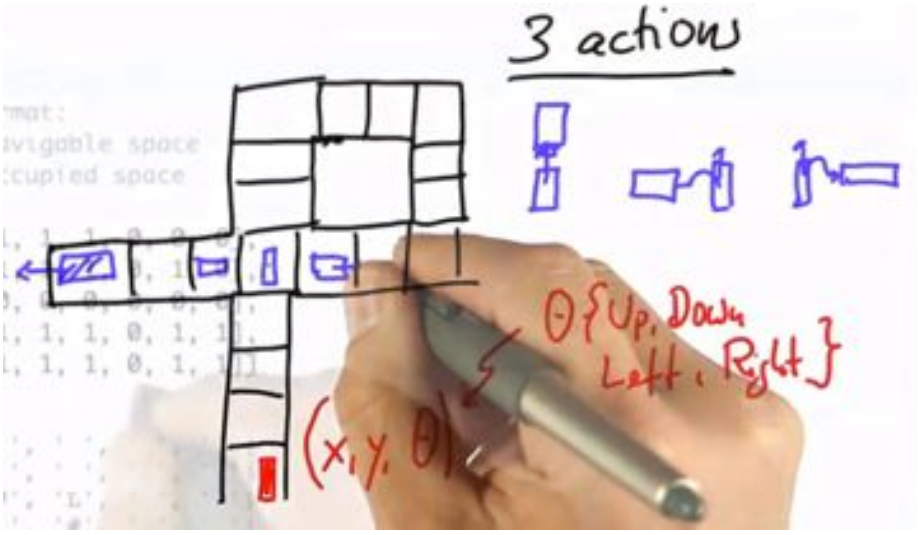
```
                            if v2 < value[x][y]:
                                change = True
                                value[x][y] = v2
                                policy[x][y] = delta_name[a]
```

Программа планирования движения на перекрестке

Вернемся к примеру в начале. Имеем автомобиль с координатами на перекрестке (x, y, θ). Для простоты пусть ориентация робота θ принимает одно из значений: вверх, вниз, влево, вправо.



Пусть задача – движение в область показанную синим. Робот может двигаться прямо, налево и прямо, направо и прямо.



В коде сетка имеет вид:

```
grid = [[1, 1, 1, 0, 0, 0],
        [1, 1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0],
        [1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1]]
```

Цель движения

```
goal = [2, 0] # final position
```

Начальное состояние

```
init = [4, 3, 0] # first 2 elements are coordinates, third is
direction
```

Обратите внимание, размерность состояния робота теперь 3.

Варианты ориентации робота перечислены в списке forward, а их имена в forward_name

```
forward = [[-1,  0], # go up
           [ 0, -1], # go left
           [ 1,  0], # go down
           [ 0,  1]] # do right
forward_name = ['up', 'left', 'down', 'right']
```

У нас есть 3 возможных действия робота и их имена:

```
action = [-1, 0, 1]
action_name = ['R', '#', 'L']
```

Мы можем прибавить к индексу варианта ориентации робота в списке forward, показанном выше,-1, 0 или 1. Если мы добавим -1 к индексу варианта ориентации робота это эквивалентно повороту направо, если добавим 1 – влево, 0 – ориентация не изменяется. Так специально составлен список forward.

«Стоимости» возможных действий робота:

```
cost = [2, 1, 20] # the cost field has 3 values: right turn, no turn,
left turn
```

Здесь правый поворот «стоит» 2, движение прямо – 1, левый – 20.

Результат работы программы должен выглядеть следующим образом:

```
[' ', ' ', ' ', 'R', '#', 'R']
[' ', ' ', ' ', '#', ' ', '#']
['*', '#', '#', '#', '#', 'R']
[' ', ' ', ' ', '#', ' ', ' ']
[' ', ' ', ' ', '#', ' ', ' ']
```

Если изменить стоимость левого поворота,

```
cost = [2, 1, 3] # the cost field has 3 values: right turn, no turn,
left turn
```

результат изменится:

```
[' ', ' ', ' ', ' ', ' ', ' ']
```

```
[' ', ' ', ' ', ' ', ' ', ' ']
['*', '#', '#', 'L', ' ', ' ']
[' ', ' ', ' ', '#', ' ', ' ']
[' ', ' ', ' ', '#', ' ', ' ']
```

Полный код программы:

```
# ----------
# User Instructions:
#
# Implement the function optimum_policy2D() below.
#
# You are given a car in a grid with initial state
# init = [x-position, y-position, orientation]
# where x/y-position is its position in a given
# grid and orientation is 0-3 corresponding to 'up',
# 'left', 'down' or 'right'.
#
# Your task is to compute and return the car's optimal
# path to the position specified in `goal'; where
# the costs for each motion are as defined in `cost'.

# EXAMPLE INPUT:

# grid format:
#     0 = navigable space
#     1 = occupied space
grid = [[1, 1, 1, 0, 0, 0],
        [1, 1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0],
        [1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1]]
goal = [2, 0] # final position
init = [4, 3, 0] # first 2 elements are coordinates, third is direction
cost = [2, 1, 20] # the cost field has 3 values: right turn, no turn, left
turn

# EXAMPLE OUTPUT:
# calling optimum_policy2D() should return the array
#
# [[' ', ' ', ' ', 'R', '#', 'R'],
#  [' ', ' ', ' ', '#', ' ', '#'],
#  ['*', '#', '#', '#', '#', 'R'],
#  [' ', ' ', ' ', '#', ' ', ' '],
#  [' ', ' ', ' ', '#', ' ', ' ']]
#
# ----------


# there are four motion directions: up/left/down/right
# increasing the index in this array corresponds to
# a left turn. Decreasing is is a right turn.

forward = [[-1,  0], # go up
           [ 0, -1], # go left
           [ 1,  0], # go down
           [ 0,  1]] # do right
forward_name = ['up', 'left', 'down', 'right']

# the cost field has 3 values: right turn, no turn, left turn
action = [-1, 0, 1]
action_name = ['R', '#', 'L']
```

```
# ---------------------------------------
# modify code below
# ---------------------------------------

def optimum_policy2D():

    value = [[[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))]]
    policy = [[[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]]
    policy2D = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]
    change = True
    while change:
        change = False
        for x in range(len(grid)):
            for y in range(len(grid[0])):
                for orientation in range(4):
                    if goal[0] == x and goal[1] == y:
                        if value[orientation][x][y] > 0:
                            value[orientation][x][y] = 0
                            policy[orientation][x][y] = '*'
                            change = True
                    elif grid[x][y] == 0:
                        for i in range(3):
                            o2 = (orientation + action[i]) % 4
                            x2 = x + forward[o2][0]
                            y2 = y + forward[o2][1]
                            if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2
< len(grid[0]) and grid[x2][y2] == 0:
                                v2 = value[o2][x2][y2] + cost[i]
                                if v2 < value[orientation][x][y]:
                                    change = True
                                    value[orientation][x][y] = v2
                                    policy[orientation][x][y] =
action_name[i]

    x = init[0]
    y = init[1]
    orientation = init[2]
    policy2D[x][y] = policy[orientation][x][y]
    while policy[orientation][x][y] != '*':
        if policy[orientation][x][y] == '#':
            o2 =  orientation
        elif policy[orientation][x][y] == 'R':
            o2 = (orientation - 1) % 4
        elif policy[orientation][x][y] == 'L':
            o2 = (orientation + 1) % 4
```

```
        x = x + forward[o2][0]
        y = y + forward[o2][1]
        orientation = o2
        policy2D[x][y] = policy[orientation][x][y]

    return policy2D # Make sure your function returns the expected grid.

policy = optimum_policy2D()
for i in range(len(policy)):
    print policy[i]
```

Комментарии

Матрица value (стоимостей движения в целевую точку из данного состояния) для трехмерного случая:

```
    value = [[[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))]]
```

Аналогично, план policy:

```
    policy = [[[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
              [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]]
    policy2D = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]
```

Также есть двумерное представление этого плана – массив policy2D:

```
    policy2D = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]
```

Изменение функции расчета value состоит в том, что мы проходим не только по всем возможным координатам, но и по всем ориентациям (которых 4). Так что это более глубокий цикл:

```
        for x in range(len(grid)):
            for y in range(len(grid[0])):
                for orientation in range(4):
```

Как и раньше, если нашли рассматриваемая ячейка – целевая, задаем соответствующее value в 0 (если еще не 0):

```
                    if goal[0] == x and goal[1] == y:
                        if value[orientation][x][y] > 0:
                            value[orientation][x][y] = 0
                            policy[orientation][x][y] = '*'
                            change = True
```

Если рассматриваем не целевое положение, проверяем является ли оно допустимым (нет препятствия)

```
                    elif grid[x][y] == 0:
```

и пробуем выполнить 3 допустимых действия робота (движение вперед либо поворот влево или вправо с движением), рассчитываем при этом новое состояние (ориентацию o2 и координаты x2 и y2):

```
                        for i in range(3):
                            o2 = (orientation + action[i]) % 4
                            x2 = x + forward[o2][0]
                            y2 = y + forward[o2][1]
```

Здесь трюк, связанный с тем, как задается и индексируется список forward. Когда мы имеем дело с i-м действием, мы прибавляем к ориентации ее изменение, соответствующее индексу i: action[i]. Напомним, action задается так:

```
action = [-1, 0, 1]
```

Мы берем остаток от деления на 4 (% 4), чтоб делать изменение ориентации циклическим. Затем мы применяем изменения координат, беря их из списка forward в соответствии с новой ориентацией o2. Здесь считается, что робот сначала поворачивает, а затем движется вперед.

Проверяем допустимость нового положения – не за пределами ли сетки и не в препятствии:

```
                                if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2
< len(grid[0]) and grid[x2][y2] == 0:
```

И как раньше рассчитываем стоимость перехода в эту новую ячейку и обновляем эту стоимость, если новая стоимость меньше старой, а также заполняем матрицу с действиями плана движения policy:

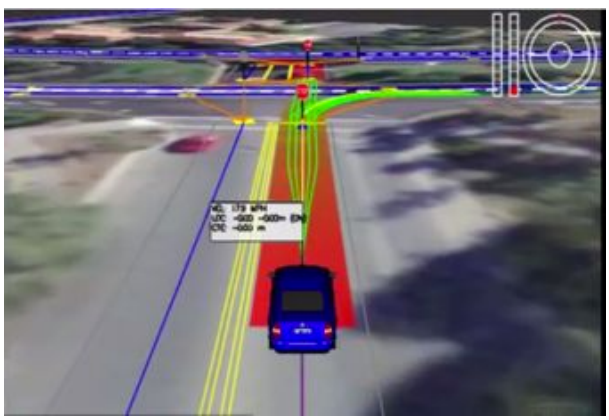
```
                                    v2 = value[o2][x2][y2] + cost[i]
                                    if v2 < value[orientation][x][y]:
                                        change = True
                                        value[orientation][x][y] = v2
                                        policy[orientation][x][y] =
action_name[i]
```

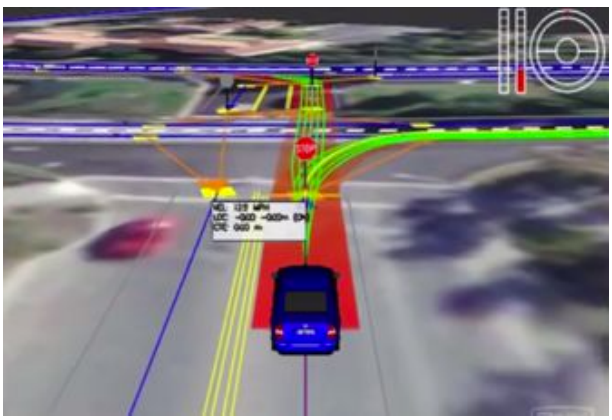
Чтобы вывести план движения policy на экран, преобразуем его к двумерному виду. Это можно сделать для частных случаев – зная начальное положение. Начиная от начального состояния

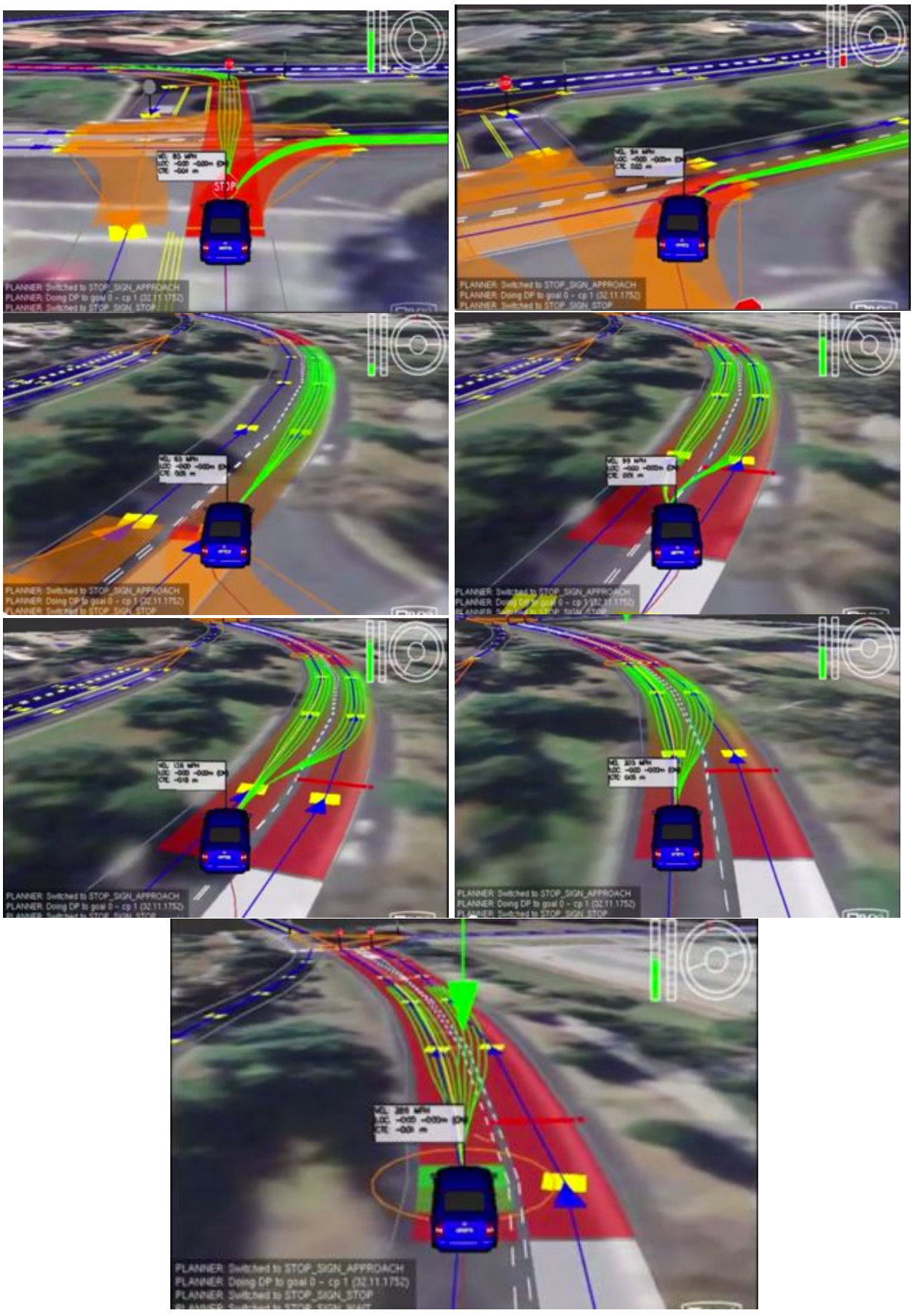
```
    x = init[0]
    y = init[1]
    orientation = init[2]
    policy2D[x][y] = policy[orientation][x][y]
```

копируем последовательность действий в двумерное представление, пока не достигнем целевой точки:

```
     while policy[orientation][x][y] != '*':
        if policy[orientation][x][y] == '#':
            o2 =  orientation
        elif policy[orientation][x][y] == 'R':
            o2 = (orientation - 1) % 4
        elif policy[orientation][x][y] == 'L':
            o2 = (orientation + 1) % 4
        x = x + forward[o2][0]
        y = y + forward[o2][1]
        orientation = o2
        policy2D[x][y] = policy[orientation][x][y]
```







Поздравления! Вы изучили 2 подхода к планированию движения – A* и динамическое программирование.

PLANNING

DISCRETE

- A* → PATH
- DP → POLICY 3D