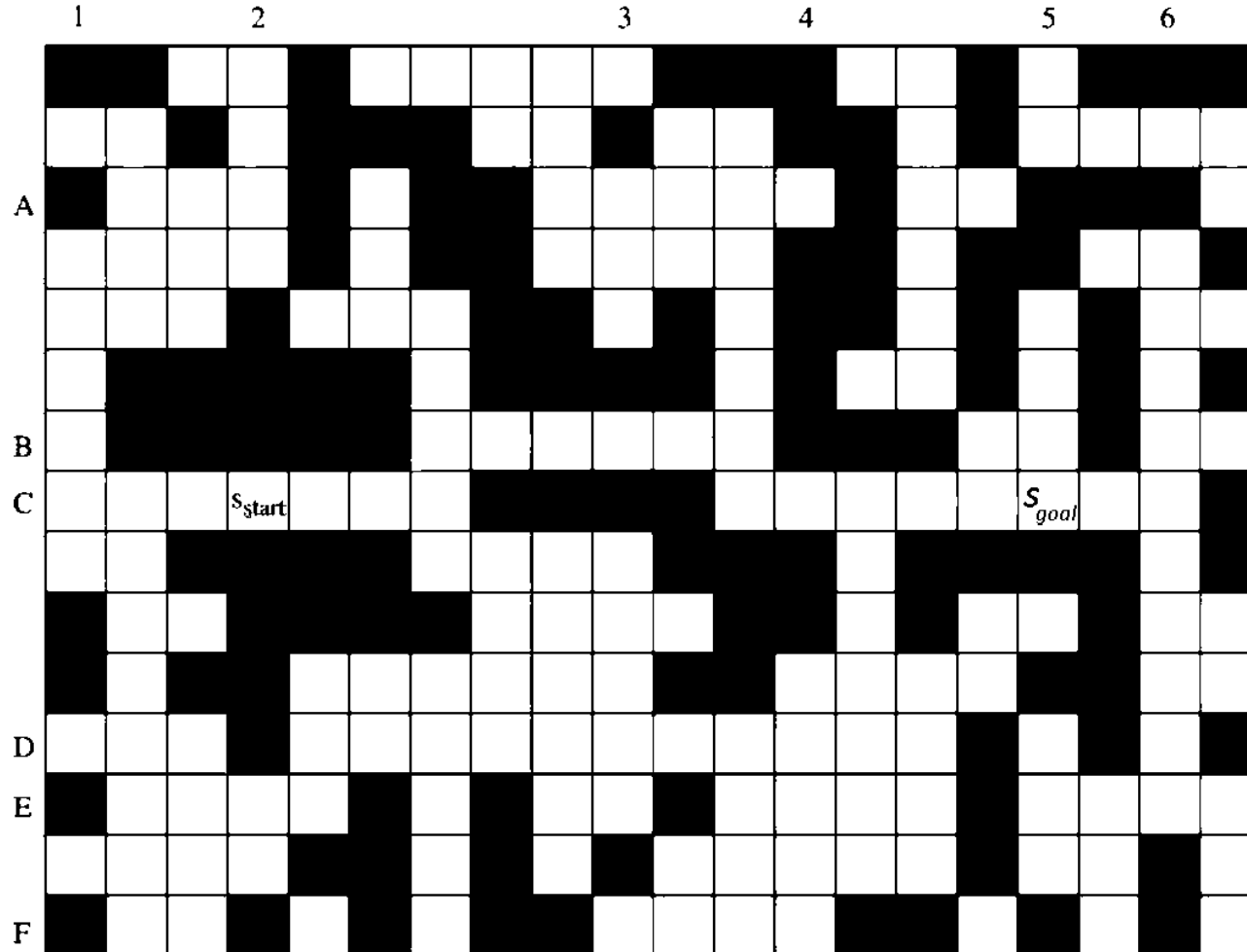


Развитие идей A^* -поиска

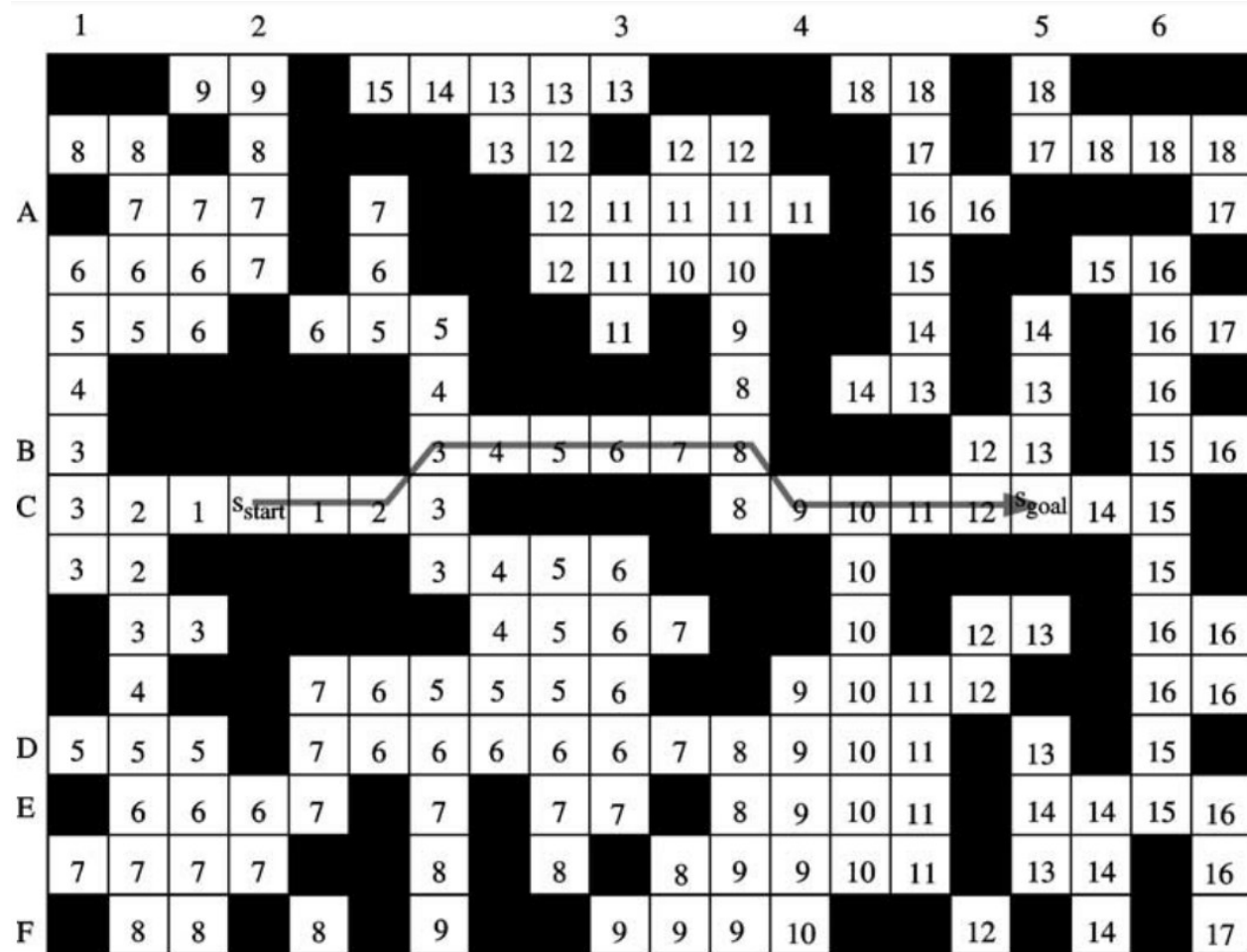
Стас Кикоть, плэннинг

Поиск пути на клетчатой бумаге

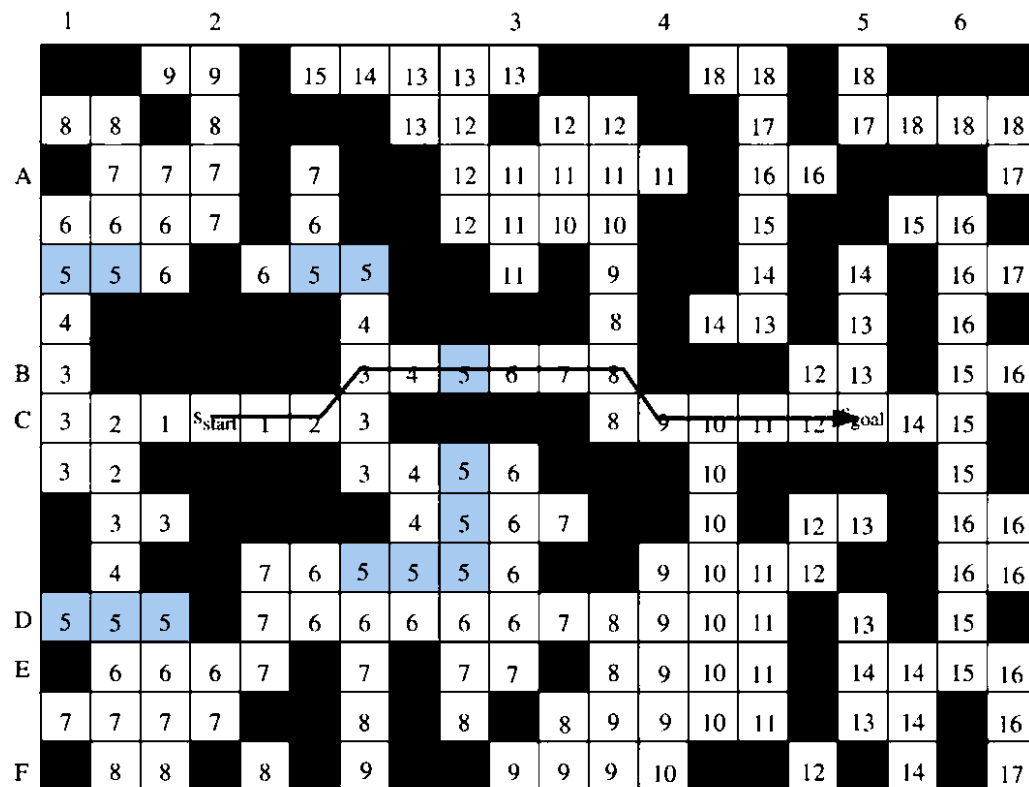


- нужно найти кратчайший путь от стартовой вершины s_{start} до целевой вершины s_{goal}
- двигаться можно только по белым клеткам
- можно ходить во все стороны и по диагонали

“Школьный” алгоритм поиска пути

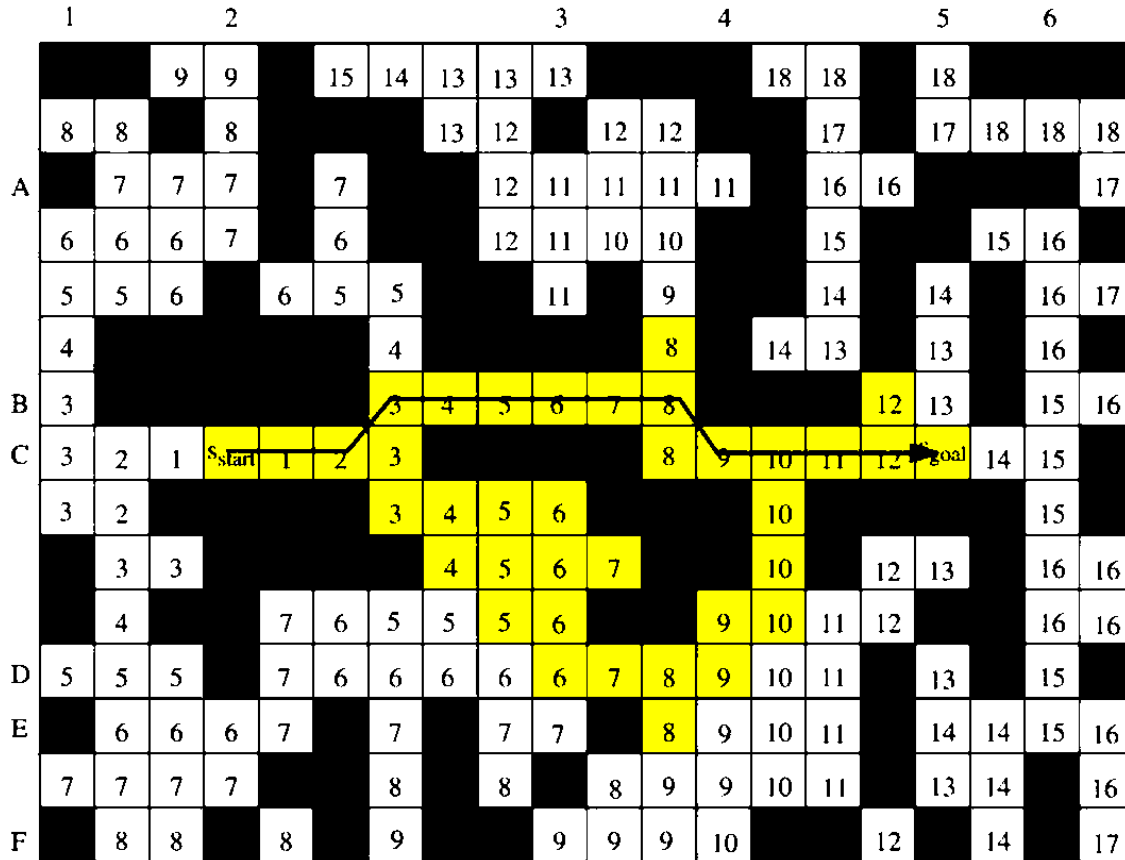


Поиск в ширину



- находит кратчайший путь
- граница поиска –
обычная очередь
- разворачивает много
лишних вершин,
так как не учитывает
направление к цели

A*-поиск



- продвигается в направлении цели
- граница поиска - очередь с приоритетом
- находит кратчайший путь
- разрешение споров (tie breaking) – предмет исследования

приоритет вершины в очереди = найденное расстояние до старта + эвристика
 $f(x) = g(x) + h(x)$ (оптимистичная оценка расстояния от вершины до цели)

Базовая модель

```
struct StateCell {  
    int x;  
    int y;  
    int speed;  
    int theta;  
    int steering;  
}
```

```
struct Control {  
    double longitudinal_acceleration;  
    // N дискретных значений  
    double angle_velocity_steering;  
    // M дискретных значений  
};
```

Плюсы:

- поиск учитывает модель движения автомобиля
- учитывается временной характер препятствий
(предсказания от prediction идут в виде букета возможных траекторий участников движения)

Минусы:

- чересчур большое пространство поиска
- может не найти решения из-за дискретизации управления

Отделить выбор траектории от выбора скорости движения ?

```
struct StateCell {  
    int x;  
    int y;  
    int theta; // азимут  
    int kappa; // кривизна  
}
```

```
struct Control {  
    double W;  
    // скорость изменения кривизны  
    // N дискретных значений (может быть N = 3 ?)  
};
```

- модель движения автомобиля **заменяется на ограничение на кривизну траектории**
- **не учитывается** временной характер препятствий (предсказания от prediction нужны в виде многоугольников, которые нужно избегать)
- все еще довольно большое пространство поиска

Еще проще ?

```
struct StateCell {  
    int x;  
    int y;  
    int theta; // азимут  
}
```

```
struct Control {  
    double kappa;  
    // кривизна  
    // N дискретных значений  
};
```

- могут получаться траектории, требующие вращение руля на месте
- все еще большое пространство поиска
- из-за дискретизации решение может быть не найдено

Гибридный подход (Стэнфорд, 2008)

- Каждой клетке (Cell) в очереди **жадным образом** приписывается некоторое **полное состояние** (State)

```
struct Cell {  
    int x;  
    int y;  
}
```

- Решает проблему потери точности при дискретизации
- Может выдать неоптимальный путь

- Работает относительно быстро

```
std::unordered_map<Cell, State> info;
```

```
struct State {  
    double x;  
    double y;  
    double speed;  
    double theta;  
    double steering;  
}
```

```
struct Control {  
    double longitudinal_acceleration;  
    // N дискретных значений  
    double angle_velocity_steering;  
    // M дискретных значений  
};
```

Гибридный подход (Торонто, 2018)

- Строим ломаную линию при помощи A*-поиска:

```
struct StateCell {  
    int x;  
    int y;  
}  
struct Control {  
    double phi;  
    // азимут  
    // принимает N  
    // дискретных значений  
};
```

- Сглаживаем ее с учетом расстояния до препятствий (минимизируем 'энергию' - интеграл квадрата кривизны)

Плюсы:

- сокращается пространство поиска

Минусы:

- модель движения автомобиля **заменяется на ограничение на кривизну траектории**
- **не учитывается** временной характер препятствий

(предсказания от **prediction** используются в виде **многоугольников**, которые нужно избегать)

Пауза перед последним рывком



POSE



PAUSE

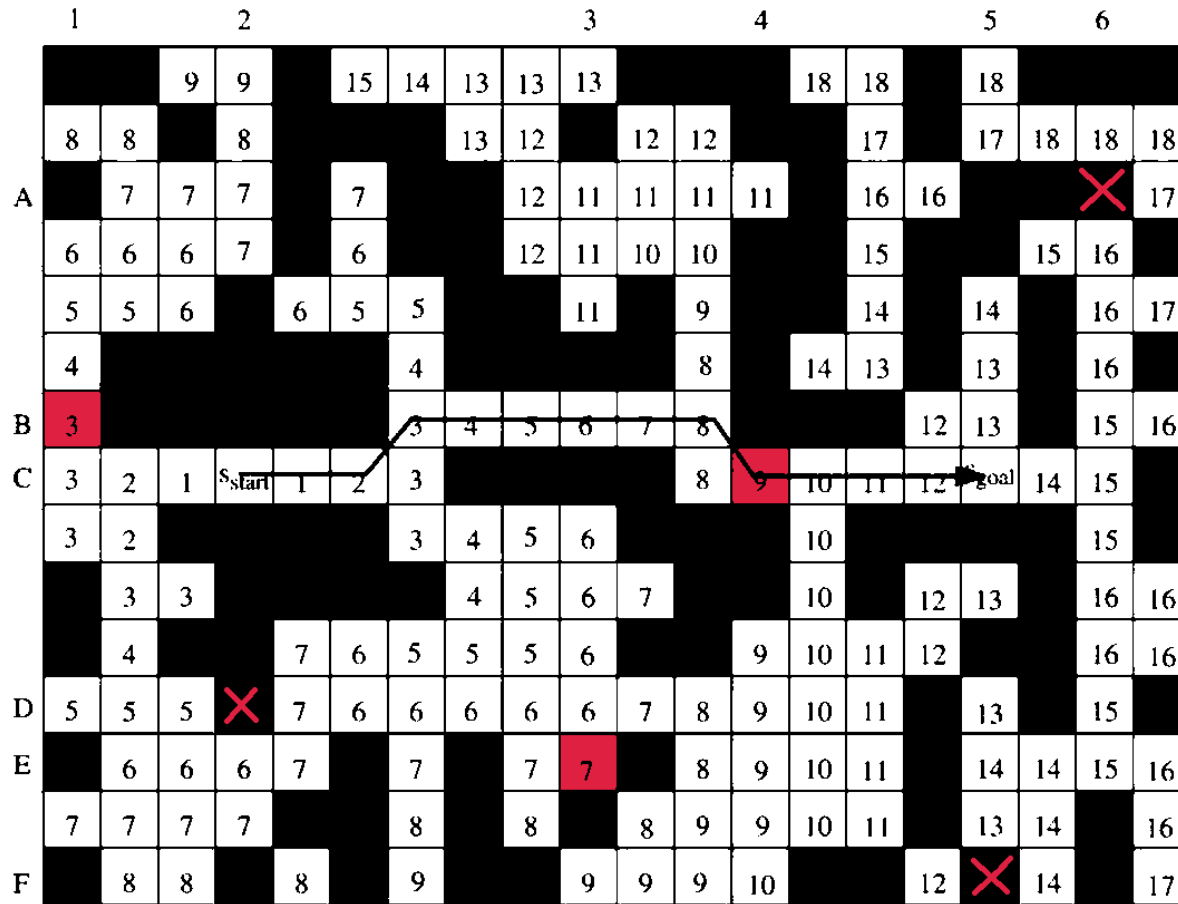


POUNCE



BOUNCE

Как быстро перерассчитать план?

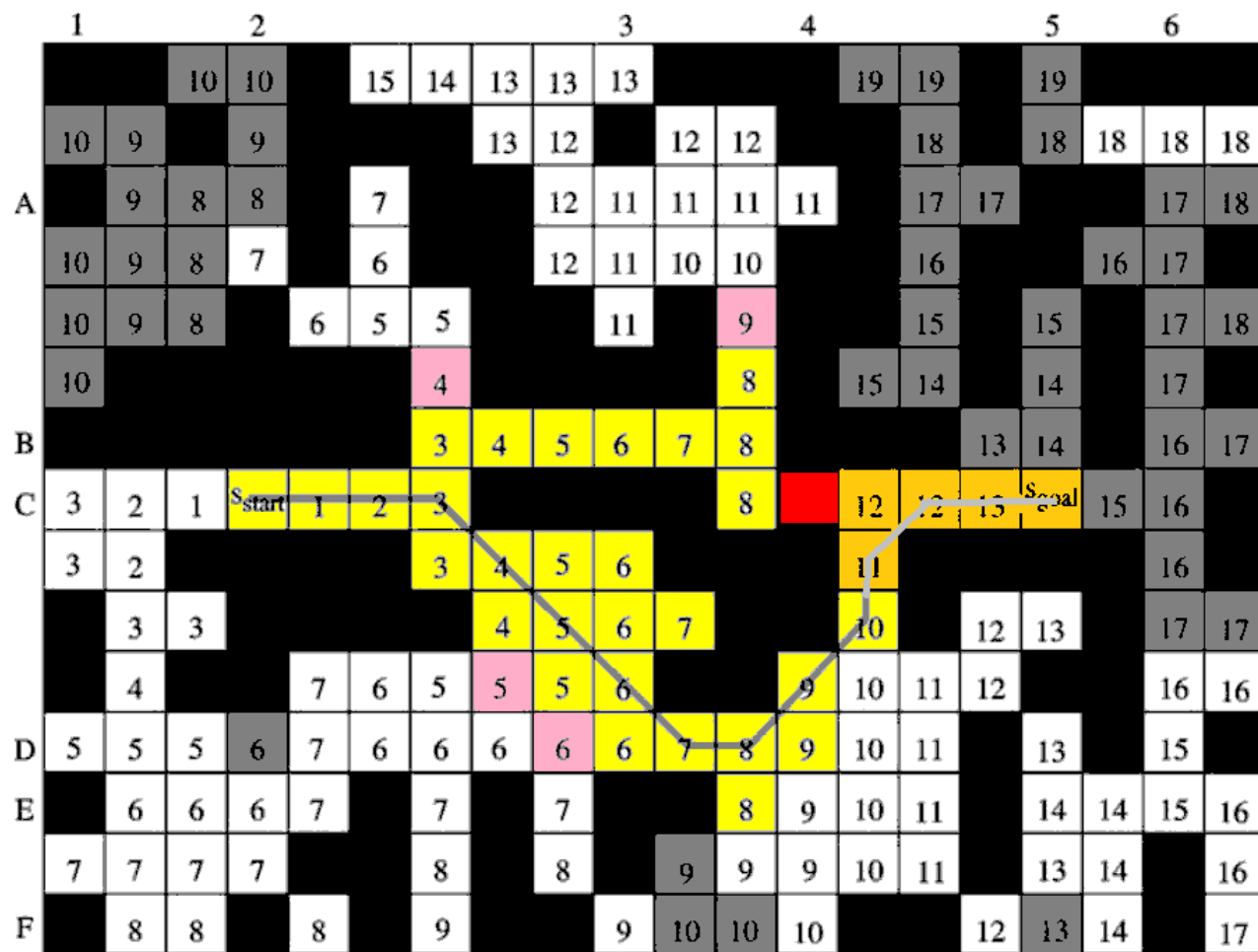


при небольших
изменениях
лабиринта

чтобы новый путь
был продолжением
построенного

сохранив
неизменившиеся
цифры

Задача о перепостроении плана



- решается эффективно алгоритмом LPA*, который при перепланировании **отталкивается от модифицированных клеток**

procedure Initialize()

{02} $U = \emptyset$;

{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;

{04} $rhs(s_{start}) = 0$;

{05} $U.Insert(s_{start}, [h(s_{start}); 0])$;

procedure CalculateKey(s)

{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;

procedure UpdateVertex(u)

{06} if ($u \neq s_{start}$) $rhs(u) = \min_{s' \in pred(u)} (g(s') + c(s', u))$;

{07} if ($u \in U$) $U.Remove(u)$;

{08} if ($g(u) \neq rhs(u)$) $U.Insert(u, CalculateKey(u))$;

LPA*

procedure Main()

{17} Initialize();

{18} forever

{19} ComputeShortestPath();

{20} Wait for changes in edge costs;

{21} for all directed edges (u, v) with changed edge costs

{22} Update the edge cost $c(u, v)$;

{23} UpdateVertex(v);

procedure ComputeShortestPath()

{09} while ($U.TopKey() \prec CalculateKey(s_{goal})$ OR $rhs(s_{goal}) \neq g(s_{goal})$)

{10} $u = U.Pop()$;

{11} if ($g(u) > rhs(u)$)

{12} $g(u) = rhs(u)$;

{13} for all $s \in succ(u)$ UpdateVertex(s);

{14} else

{15} $g(u) = \infty$;

{16} for all $s \in succ(u) \cup \{u\}$ UpdateVertex(s);

Пример работы LPA*

Start distances / heuristics

| | 0 | 1 | 2 | 3 |
|---|-----|-----|-----|-----|
| A | 3/5 | 2/5 | 1/5 | 0/5 |
| B | | 2/4 | | 1/4 |
| C | | 3/3 | | 2/3 |
| D | | 4/2 | | 3/3 |
| E | | 5/1 | | 4/3 |
| F | 6/0 | 6/1 | 5/2 | 5/3 |

Iteration #1

| | 0 | 1 | 2 | 3 |
|---|----------|----------|----------|-------------------|
| A | ∞ | ∞ | ∞ | ∞ [5;0] |
| B | | ∞ | | ∞ |
| C | | ∞ | | ∞ |
| D | | ∞ | | ∞ |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #2

| | 0 | 1 | 2 | 3 |
|---|----------|----------|-------------------|-------------------|
| A | ∞ | ∞ | ∞ [6;1] | 0 |
| B | | ∞ | | ∞ [5;1] |
| C | | ∞ | | ∞ |
| D | | ∞ | | ∞ |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Пример работы LPA*

Iteration #3

| | 0 | 1 | 2 | 3 |
|---|----------|----------|-------------------|-------------------|
| A | ∞ | ∞ | ∞ [6;1] | 0 |
| B | | ∞ | | 1 |
| C | | ∞ | | ∞ [5;2] |
| D | | ∞ | | ∞ |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #4

| | 0 | 1 | 2 | 3 |
|---|----------|----------|-------------------|-------------------|
| A | ∞ | ∞ | ∞ [6;1] | 0 |
| B | | ∞ | | 1 |
| C | | ∞ | | 2 |
| D | | ∞ | | ∞ [6;3] |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #5

| | 0 | 1 | 2 | 3 |
|---|----------|-------------------|----------|-------------------|
| A | ∞ | ∞ [7;2] | 1 | 0 |
| B | | ∞ [6;2] | | 1 |
| C | | ∞ | | 2 |
| D | | ∞ | | ∞ [6;3] |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Пример работы LPA*

Iteration #6

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|----------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | ∞ [6;3] | | 2 |
| D | | ∞ | | ∞ [6;3] |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #7

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|----------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | ∞ [6;4] | | ∞ [6;3] |
| E | | ∞ | | ∞ |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #8

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|----------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | ∞ [6;4] | | 3 |
| E | | ∞ | | ∞ [7;4] |
| F | ∞ | ∞ | ∞ | ∞ |

Пример работы LPA*

Iteration #9

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|----------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | 4 | | 3 |
| E | | ∞ [6;5] | | ∞ [7;4] |
| F | ∞ | ∞ | ∞ | ∞ |

Iteration #10

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|-------------------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | 4 | | 3 |
| E | | 5 | | ∞ [7;4] |
| F | ∞ [6;6] | ∞ [7;6] | ∞ [8;6] | ∞ |

Shortest path

| | 0 | 1 | 2 | 3 |
|---|-------------------|-------------------|-------------------|-------------------|
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | 4 | | 3 |
| E | | 5 | | ∞ [7;4] |
| F | 6 | ∞ [7;6] | ∞ [8;6] | ∞ |

Второй Поиск

Предыдущий поиск

| Iteration #9 | | | | | Iteration #10 | | | | | Shortest path | | | | |
|--------------|-------------------|-------------------|-------------------|-------------------|---------------|-------------------|-------------------|-------------------|-------------------|---------------|-------------------|-------------------|---|-------------------|
| | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 | A | ∞ [8;3] | ∞ [7;2] | 1 | 0 | A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 | B | | 2 | | 1 | B | | 2 | | 1 |
| C | | | 3 | | C | | 3 | | 2 | C | | 3 | | 2 |
| D | | | 4 | | D | | 4 | | 3 | D | | 4 | | 3 |
| E | | | ∞ [6;5] | ∞ [7;4] | E | | 5 | | ∞ [7;4] | E | | 5 | | ∞ [7;4] |
| F | ∞ | ∞ | ∞ | ∞ | F | ∞ [6;6] | ∞ [7;6] | ∞ [8;6] | ∞ | F | | 6 | | ∞ [7;6] |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| A | 3 | 2 | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | 4 | | 3 |
| E | | 5 | | 4 |
| F | 6 | 6 | 5 | 5 |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| A | 3 | 2 | 1 | 0 |
| B | | 2 | | 1 |
| C | | 3 | | 2 |
| D | | 4 | | 3 |
| E | | 5 | | 4 |
| F | 6 | 6 | 5 | 5 |

Изменение лабиринта

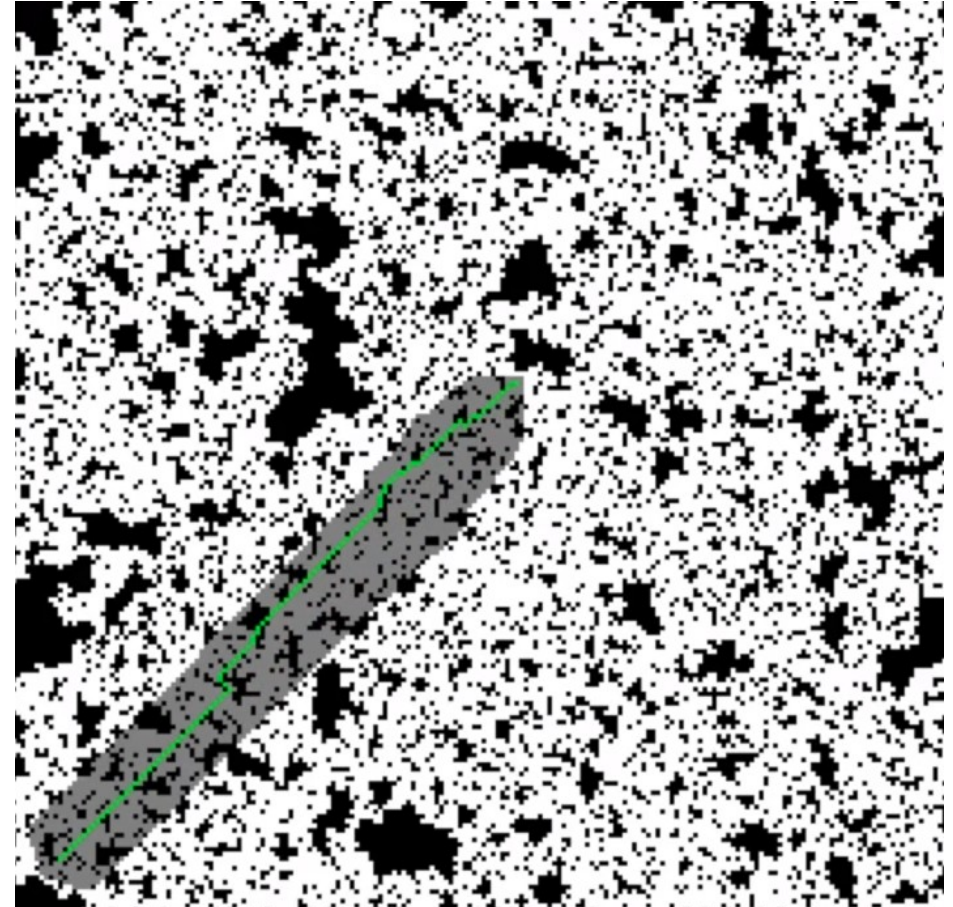
| Iteration #1 | | | | | Iteration #2 | | | | | Iteration #3 | | | | |
|--------------|-------------------|-------------------|-------------------|-------------------|--------------|-------------------|-------------------|----------|-------------------|--------------|-------------------|-------------------|----------|-------------------|
| | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |
| A | ∞ [8;3] | ∞ [7;2] | 1 | 0 | A | ∞ [8;3] | ∞ [7;2] | 1 | 0 | A | ∞ [8;3] | ∞ [7;2] | 1 | 0 |
| B | | 2 | | 1 | B | | 2 | | 1 | B | | 2 | | 1 |
| C | | | 3 | | C | | 3 | | 2 | C | | 3 | | 2 |
| D | | | | 3 | D | | | | 3 | D | | | | 3 |
| E | | | 5 [6;5] | ∞ [7;4] | E | | ∞ [8;7] | | ∞ [7;4] | E | | ∞ | | ∞ [7;4] |
| F | 6 | ∞ [7;6] | ∞ [8;6] | ∞ | F | 6 [6;6] | ∞ [8;7] | ∞ | ∞ | F | ∞ | ∞ | ∞ | ∞ |

| Iteration #4 | | | | | Iteration #5 | | | | | Iteration #6 | | | | |
|--------------|-------------------|----------|----------|-------------------|--------------|-------------------|----------|-------------------|-------------------|--------------|-------------------|-------------------|---|-------------------|
| | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |
| A | ∞ [8;3] | 2 | 1 | 0 | A | ∞ [8;3] | 2 | 1 | 0 | A | ∞ [8;3] | 2 | 1 | 0 |
| B | | 2 | | 1 | B | | 2 | | 1 | B | | 2 | | 1 |
| C | | | 3 | | C | | 3 | | 2 | C | | 3 | | 2 |
| D | | | | 3 | D | | | | 3 | D | | | | 3 |
| E | | | ∞ | ∞ [7;4] | E | | ∞ | | 4 | E | | ∞ [7;6] | | 4 |
| F | ∞ | ∞ | ∞ | ∞ | F | ∞ | ∞ | ∞ [7;5] | ∞ [8;5] | F | ∞ | ∞ [7;6] | 5 | ∞ [8;5] |

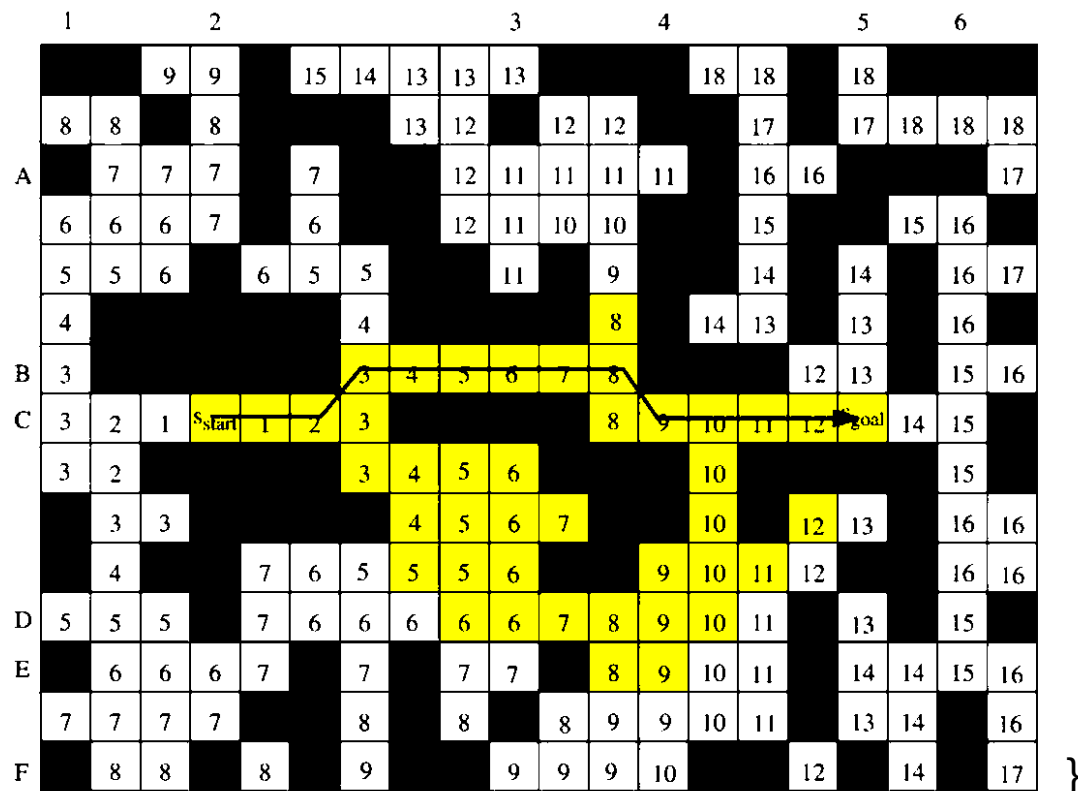
| Iteration #7 | | | | | Iteration #8 | | | | | Shortest path | | | | |
|--------------|-------------------|-------------------|---|-------------------|--------------|-------------------|---|---|-------------------|---------------|-------------------|---|---|-------------------|
| | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |
| A | ∞ [8;3] | 2 | 1 | 0 | A | ∞ [8;3] | 2 | 1 | 0 | A | ∞ [8;3] | 2 | 1 | 0 |
| B | | 2 | | 1 | B | | 2 | | 1 | B | | 2 | | 1 |
| C | | | 3 | | C | | 3 | | 2 | C | | 3 | | 2 |
| D | | | | 3 | D | | | | 3 | D | | | | 3 |
| E | | | 6 | 4 | E | | 6 | | 4 | E | | 6 | | 4 |
| F | ∞ [7;7] | ∞ [7;6] | 5 | ∞ [8;5] | F | ∞ [7;7] | 6 | 5 | ∞ [8;5] | F | 7 | 6 | 5 | ∞ [8;5] |

Алгоритм D*-Lite

- используется в задаче о навигации с неточной картой
- ищет путь от цели к позиции робота
- в определенных сценариях работает на два порядка быстрее A*



A*-поиск



```

function A*(start, goal, f) {
    % множество пройденных вершин
    var closed := the empty set
    % граница поиска
    var open := make_queue(f)
    enqueue(open, path(start))
    while open is not empty
        var p := remove_first(open)
        var x := the last node of p
        if x in closed
            continue
        if x = goal
            return p
        add(closed, x)
        % добавляем смежные вершины
        foreach y in successors(x)
            enqueue(open, add_to_path(p, y))
    return failure
}
    
```

приоритет вершины в очереди = найденное расстояние до старта + эвристика
 $f(x) = g(x) + h(x)$ (оптимистичная оценка расстояния от вершины до цели)