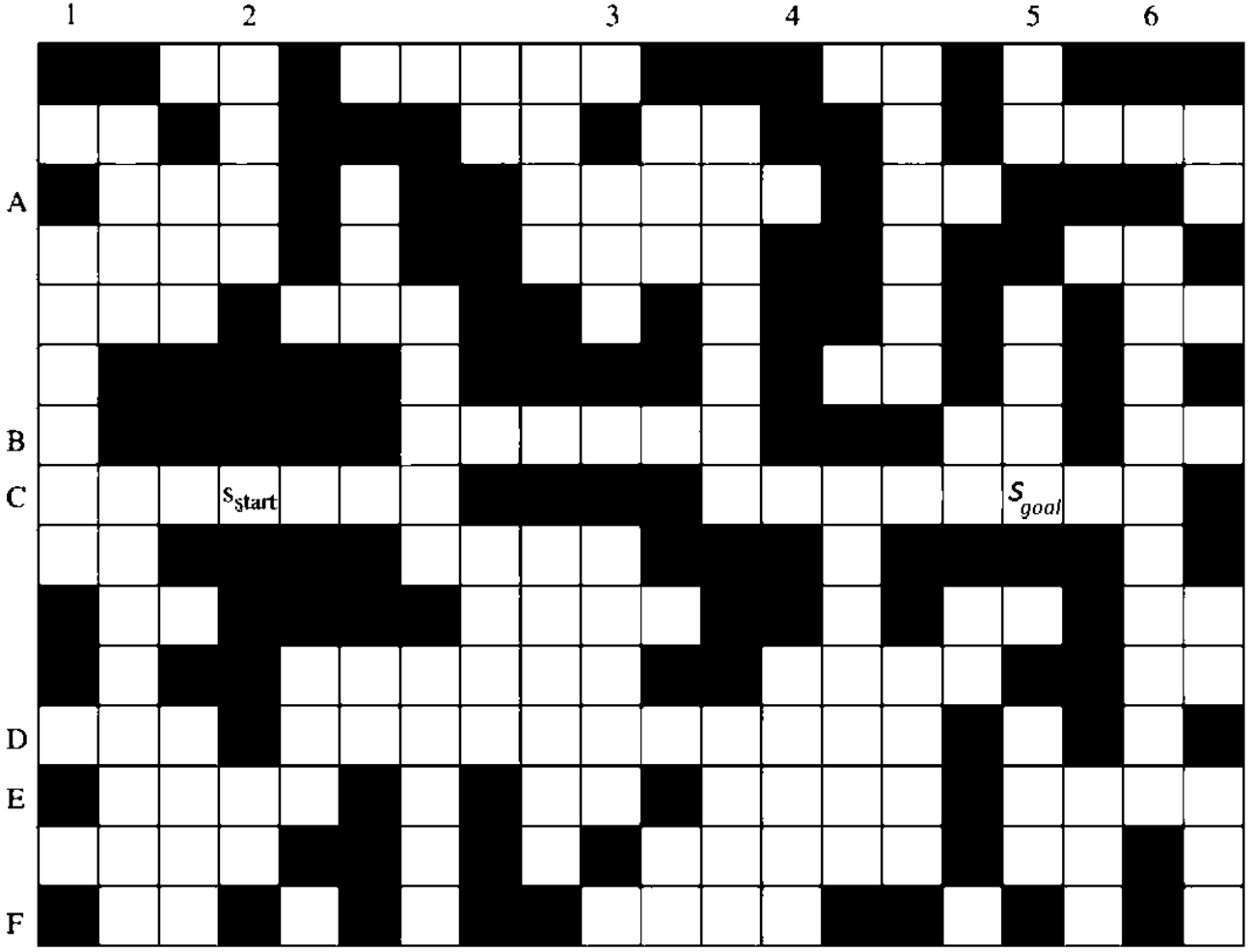
Развитие идей A*-поиска

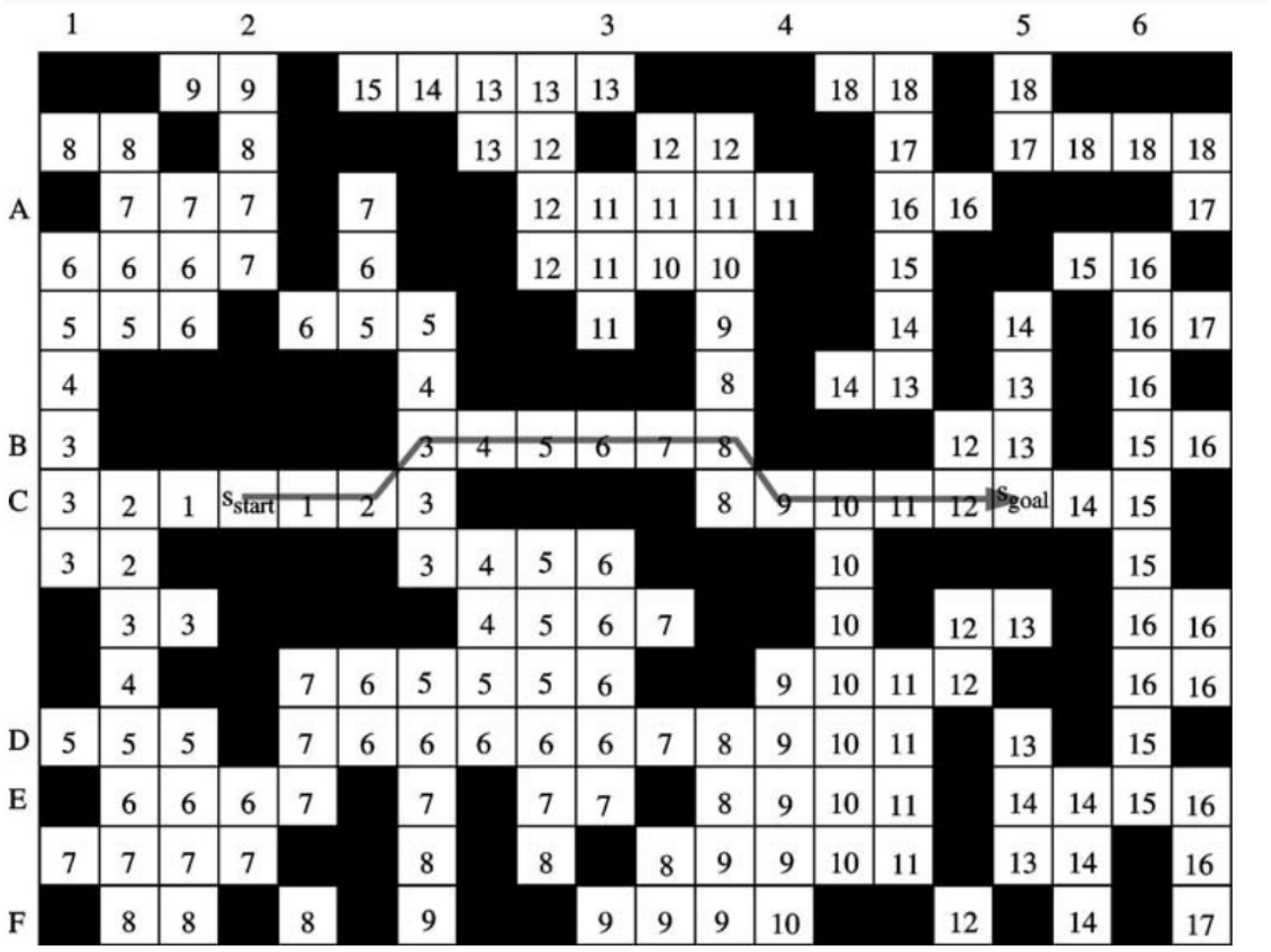
Кикоть Станислав

Поиск пути на клетчатой бумаге

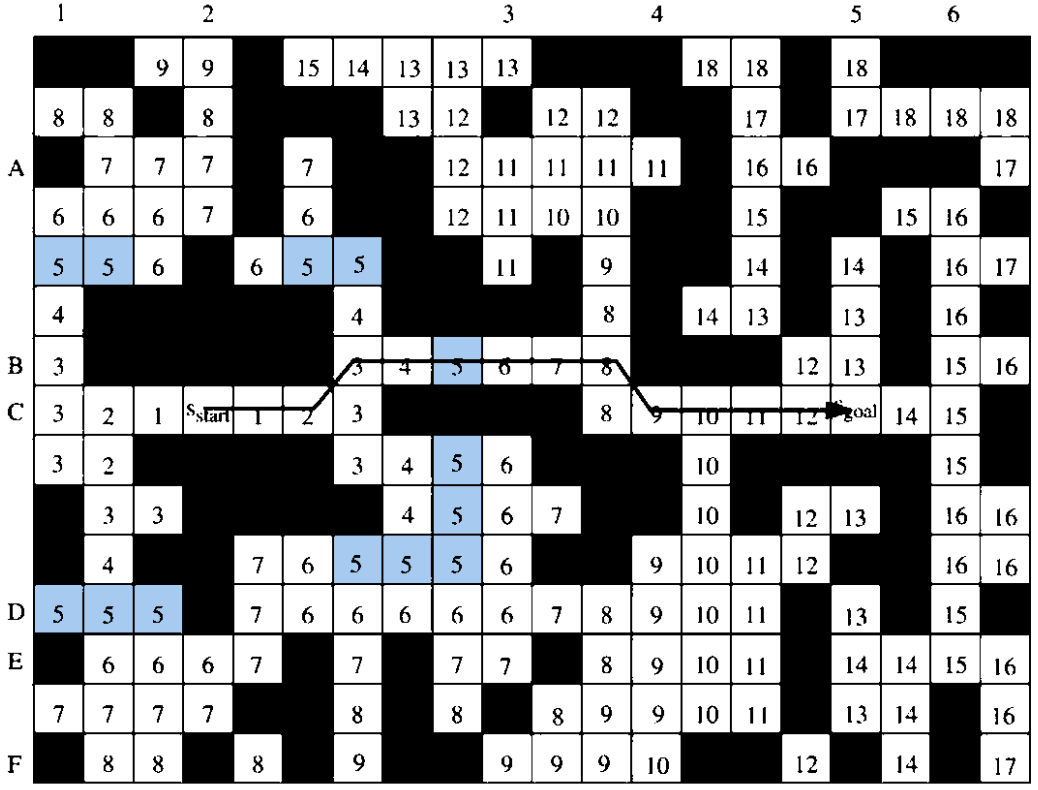


- нужно найти кратчайший путь от стартовой вершины s_{start} до целевой вершины s_{goal}
- двигаться можно только по белым клеткам
- можно ходить во все стороны и по диагонали

“Школьный” алгоритм поиска пути

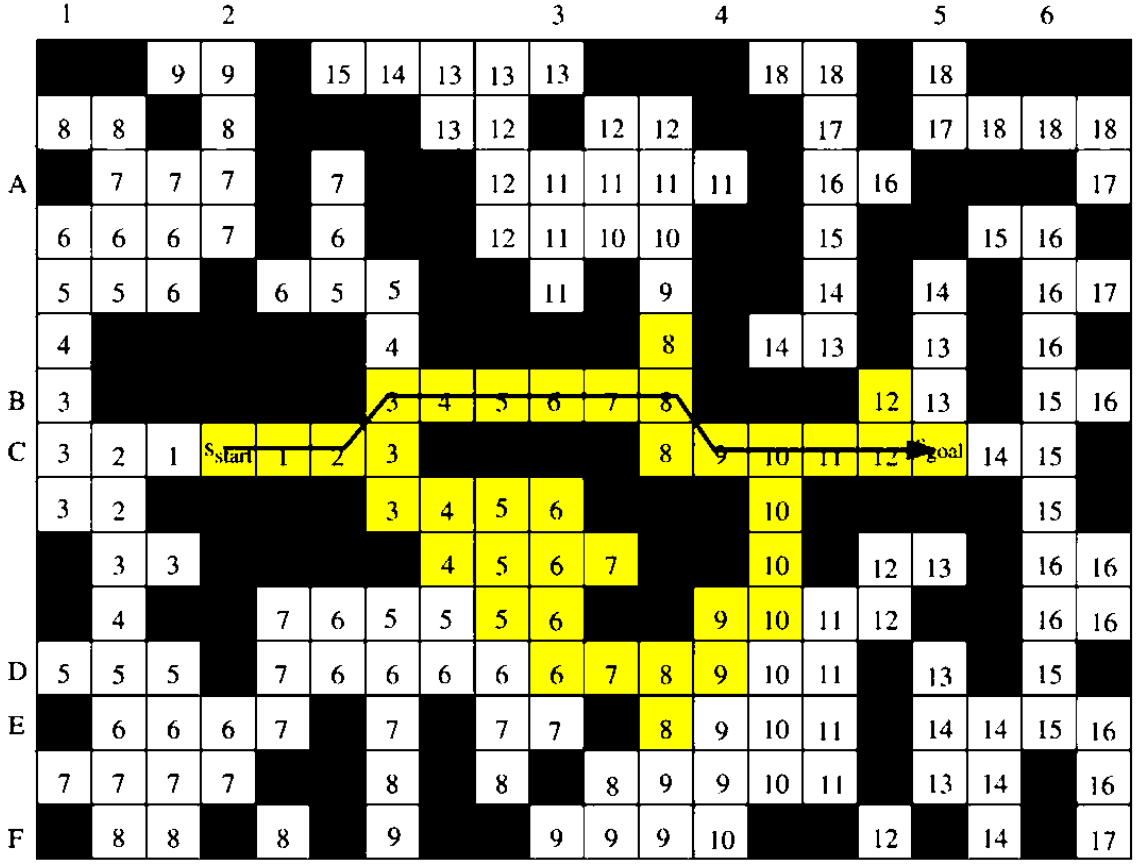


Поиск в ширину



- 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)

Зашить в A* модель движения ?

```
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)

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

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

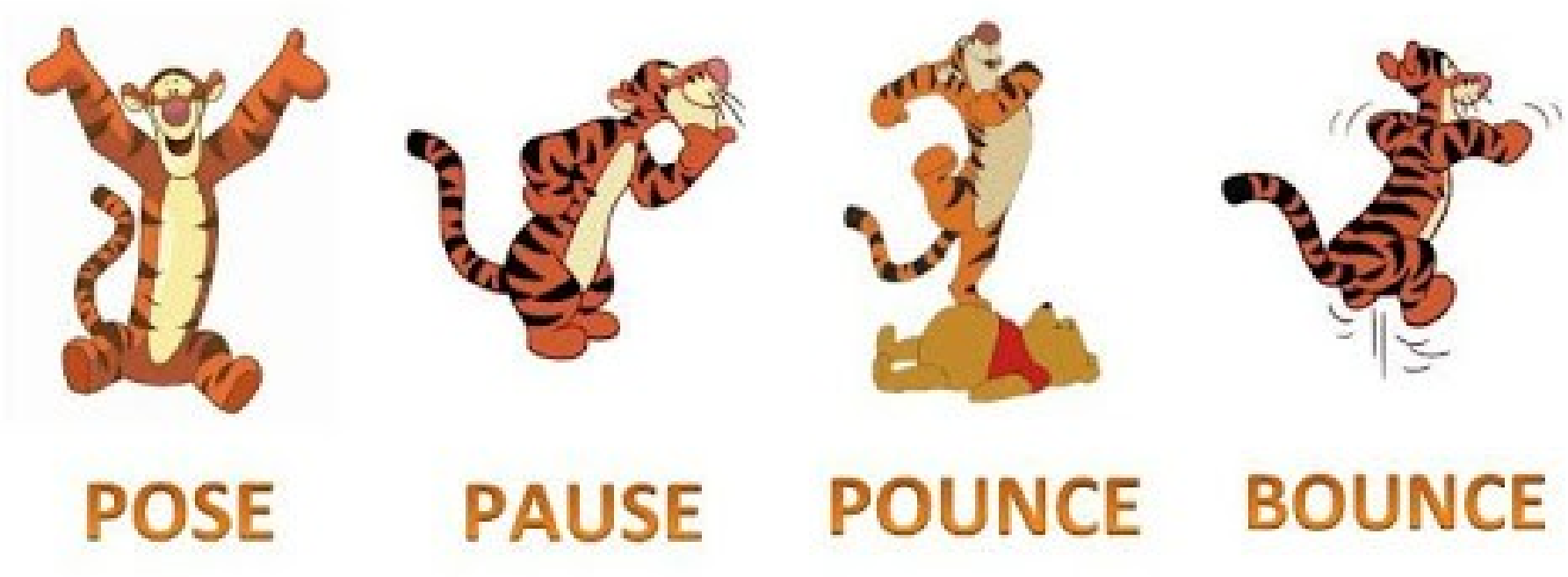
- **сокращается пространство поиска**

Минусы:

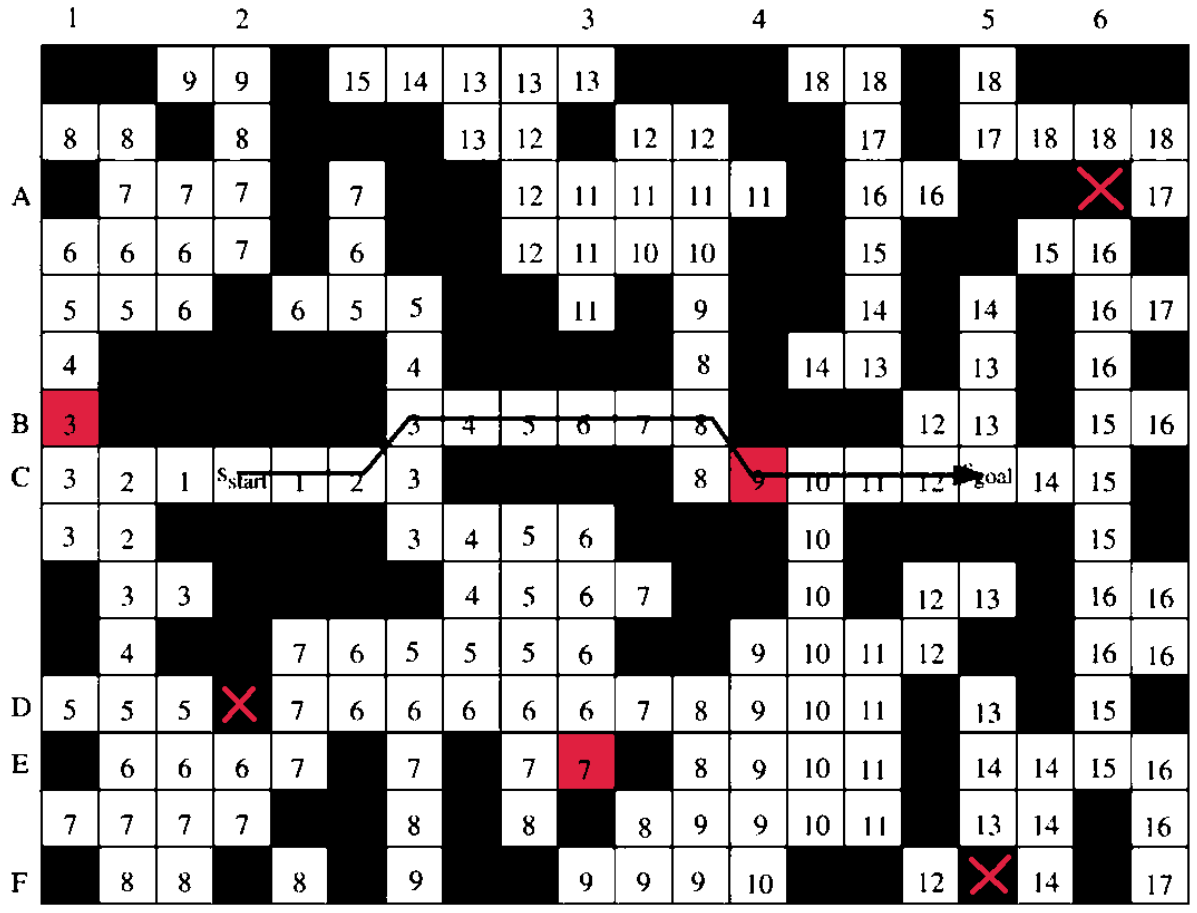
- **модель движения автомобиля заменяется на ограничение на кривизну траектории**
- **не учитывается временной характер препятствий**

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

Пауза перед последним рывком



Как быстро перерассчитать план?

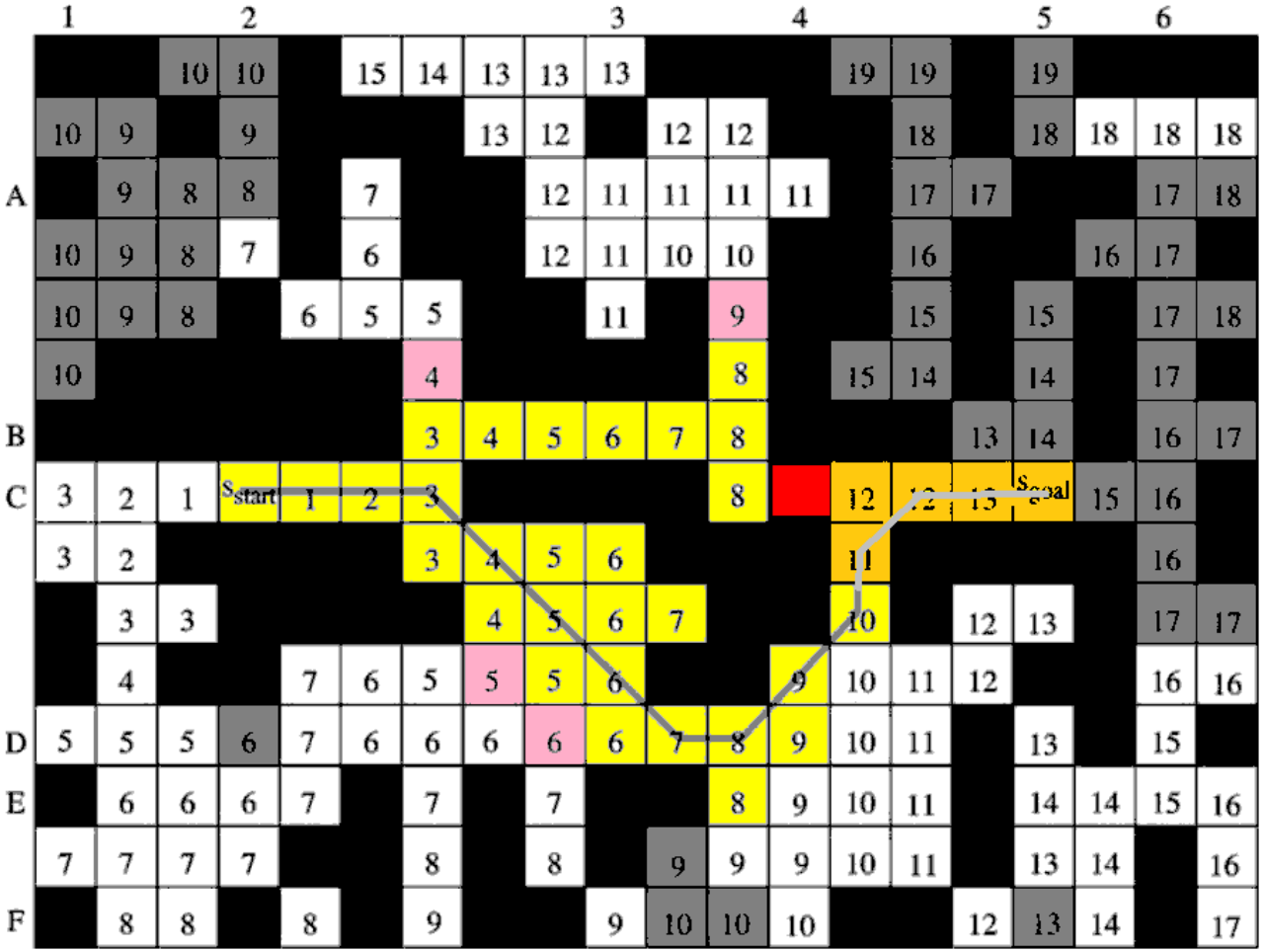


при небольших изменениях лабиринта

чтобы новый путь был продолжением построенного

сохранив неизменившиеся цифры

Задача о перепостроении плана

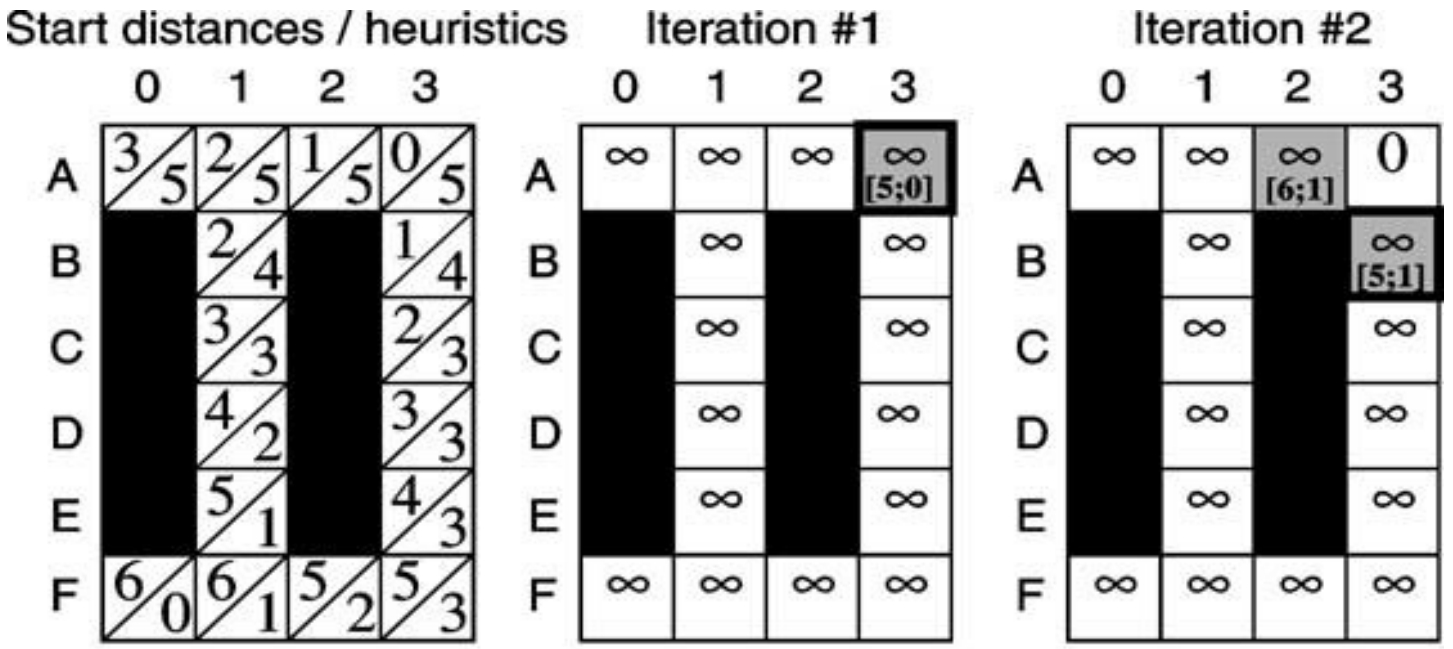


- решается эффективно алгоритмом LPA*, который при перепланировании **отталкивается от модифицированных клеток**

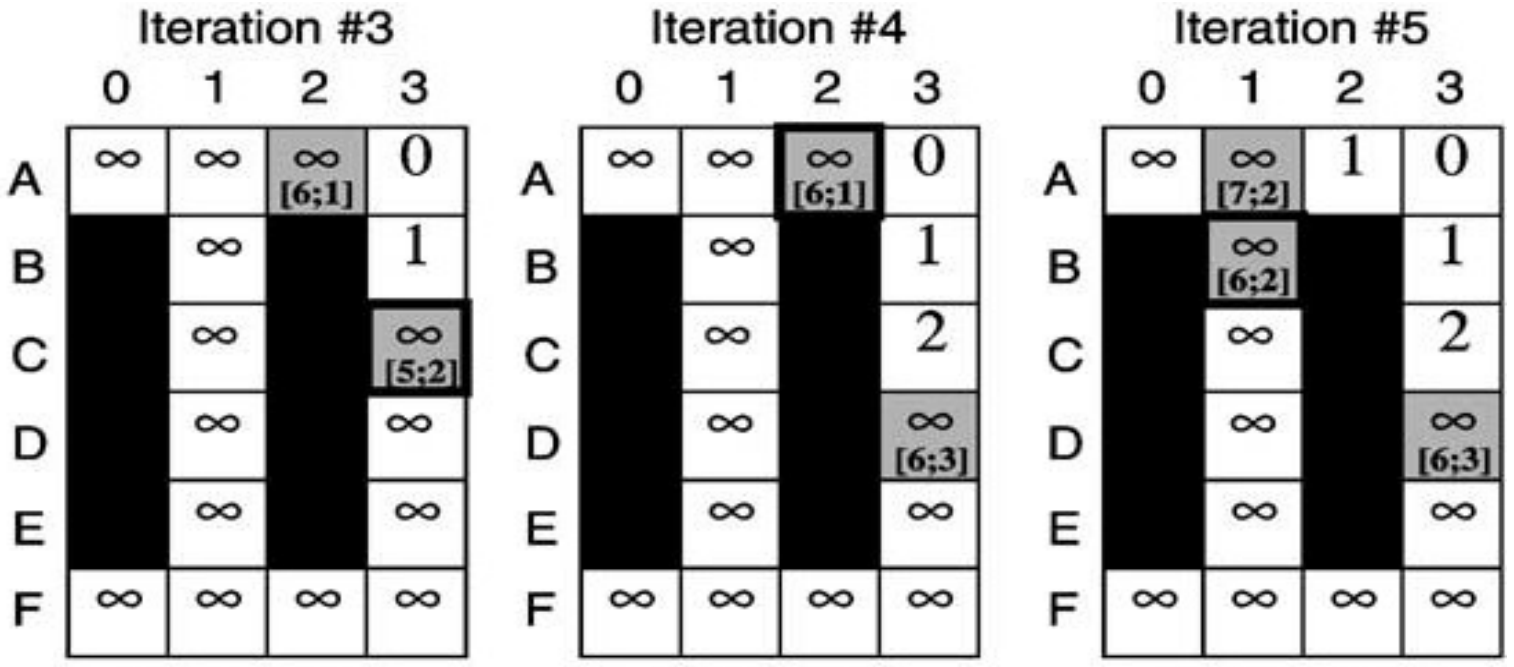
LPA*

```
procedure Initialize()
{02} U = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{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 ≠ s_start) rhs(u) = min_{s' ∈ pred(u)} (g(s') + c(s', u));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
{09} while (U.TopKey() ≺ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
{10}     u = U.Pop();
{11}     if (g(u) > rhs(u))
{12}         g(u) = rhs(u);
{13}         for all s ∈ succ(u) UpdateVertex(s);
{14}     else
{15}         g(u) = ∞;
{16}         for all s ∈ succ(u) ∪ {u} UpdateVertex(s);

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);
```

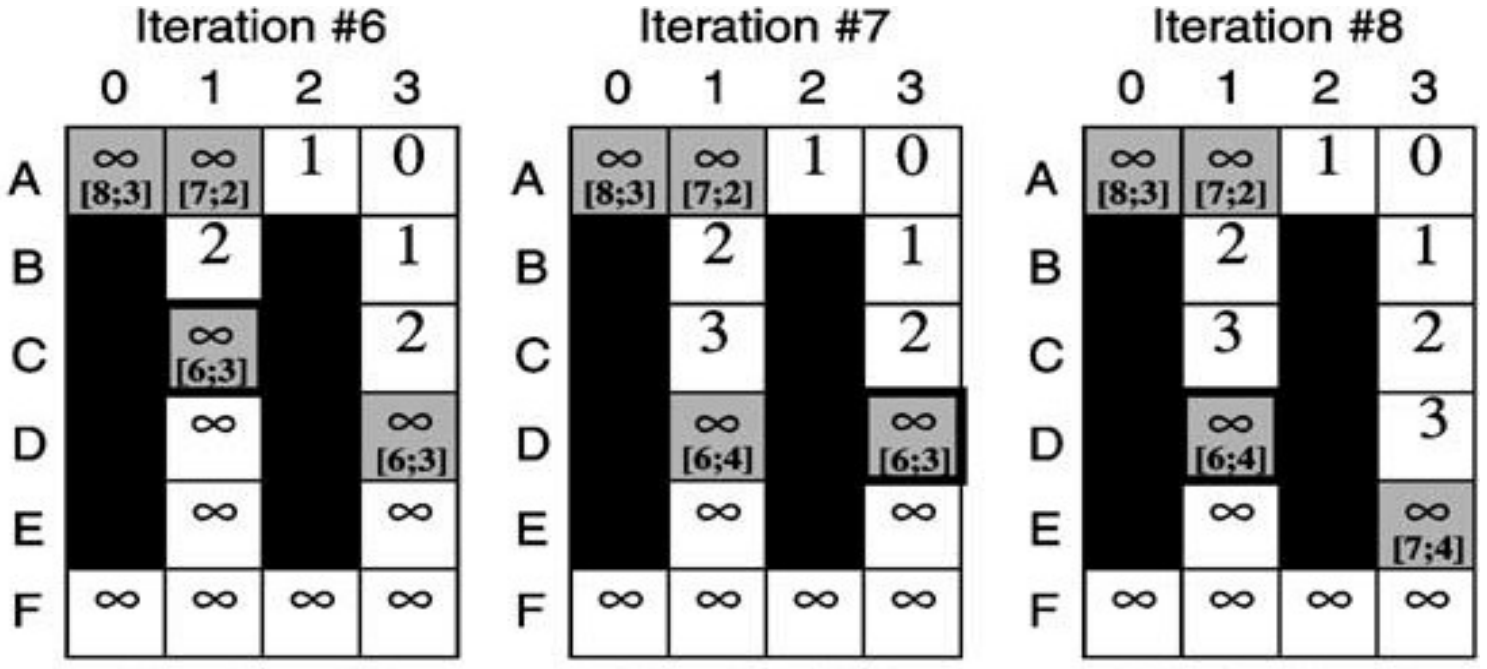
Пример работы LPA*



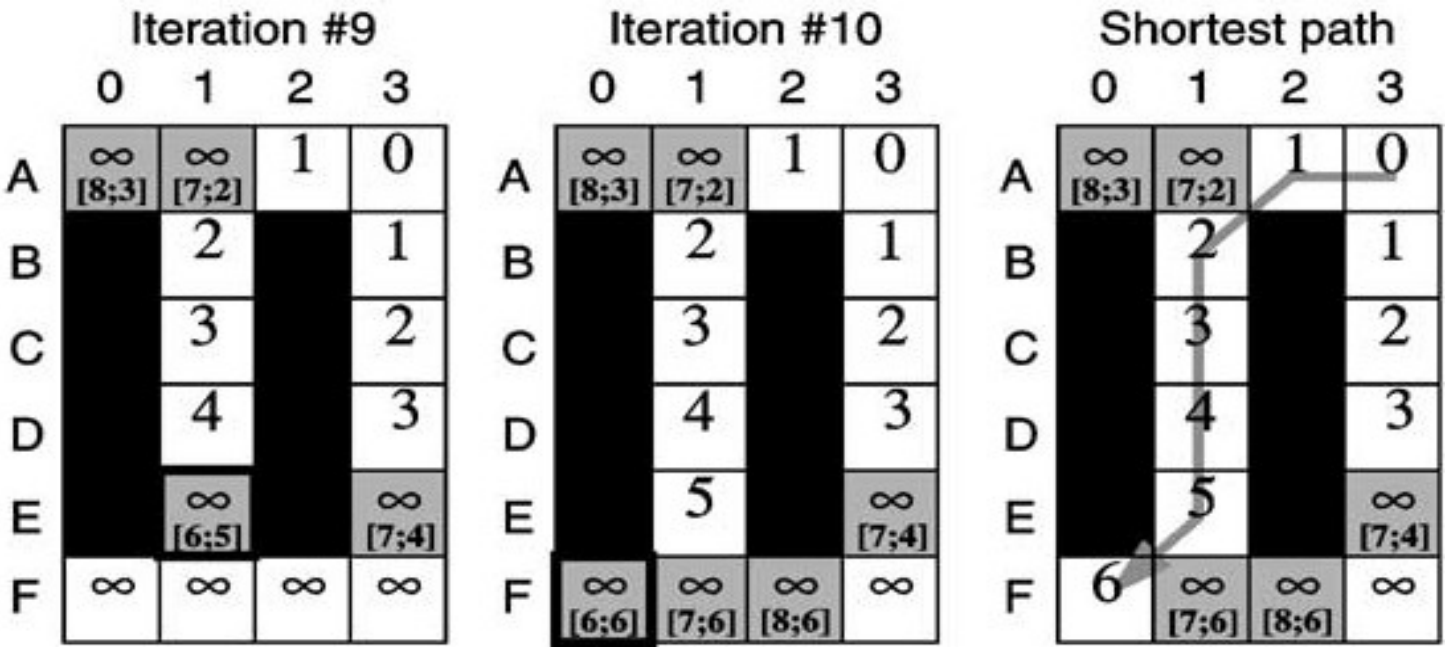
Пример работы LPA*



Пример работы LPA*

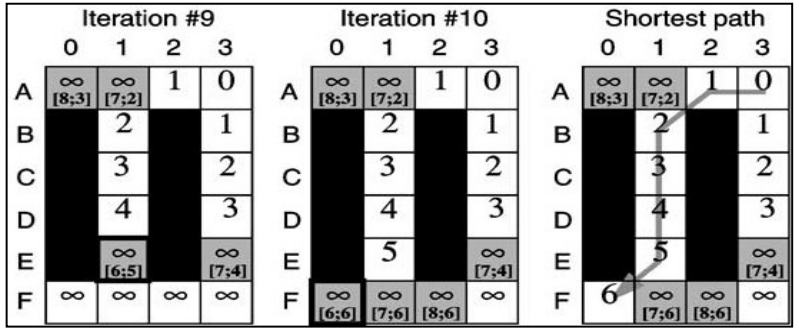


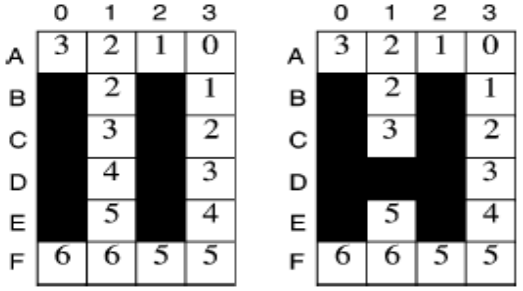
Пример работы LPA*



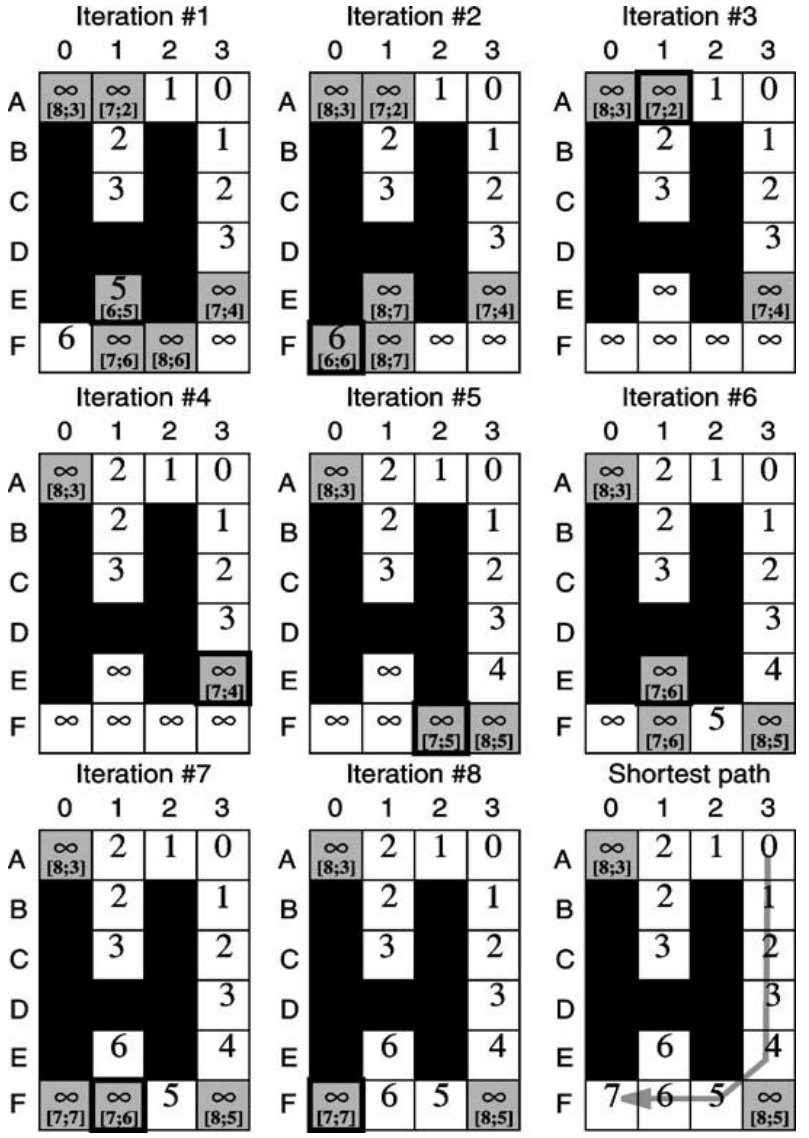
Второй Поиск

Предыдущий поиск



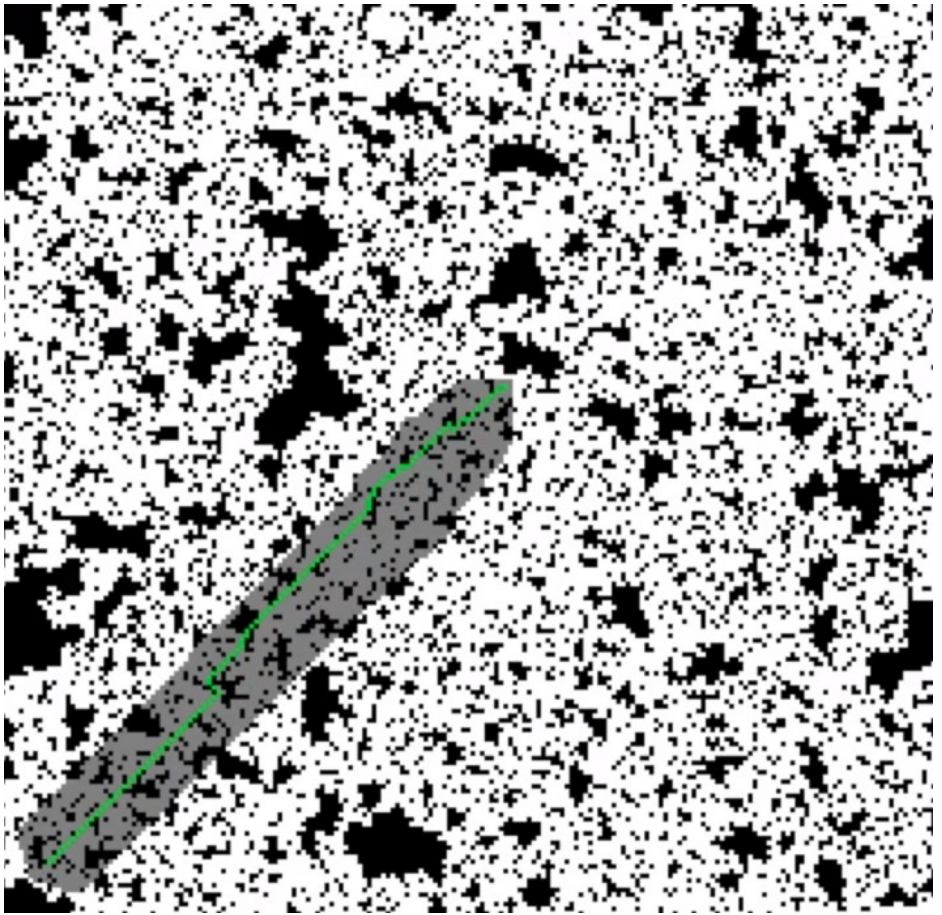


Изменение лабиринта

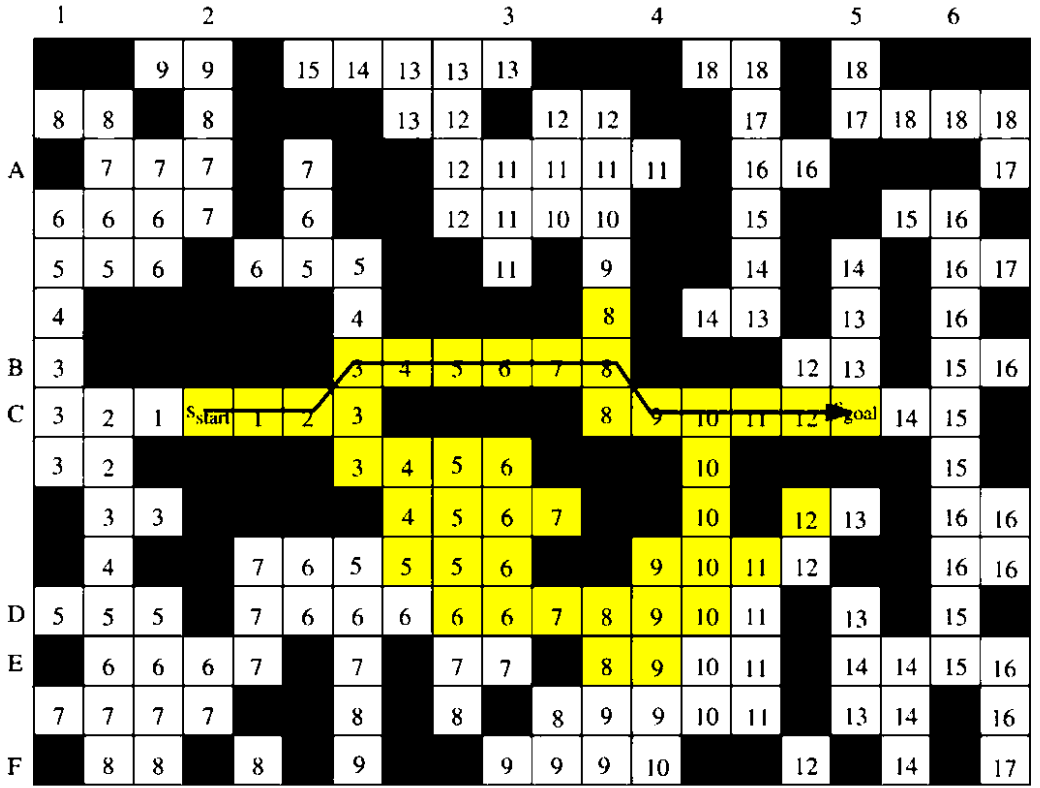


Алгоритм D*-Lite

- 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)