МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
Департамент математического и компьютерного моделирования

Реферат
о практическом задание по дисциплине АИСД
«Алгоритм поиска D*Lite»

направление подготовки 09.03.03 «Прикладная информатика»
профиль «Прикладная информатика в компьютерном дизайне»

Выполнил студент
гр. Б9121-09.03.03пикд
Масличенко Елизавета Андреевна

Доклад защищен:
С оценкой _____

Руководитель практики
Доцент ИМКТ А.С Кленин

г. Владивосток
2023

Аннотация

Методы инкрементного эвристического поиска используют эвристику для фокусировки поиска и повторного использования информации из предыдущих поисков, чтобы находить решения для серии похожих поисковых задач намного быстрее, чем это возможно при решении каждой поисковой задачи с нуля. D* Lite реализует то же поведение, что и сфокусированный динамический A * Стенца , но алгоритмически отличается. Мы доказываем свойства D * Lite и экспериментально демонстрируем преимущества сочетания инкрементного и эвристического поиска .

Постановка задачи

Рассмотрим целенаправленную задачу навигации робота в неизвестной местности, где робот всегда наблюдает, какие из его восьми соседних ячеек можно обойти, а затем перемещается с затратами от одной к одной из них. Робот стартует со стартовой ячейки и должен перейти к целевой ячейке. Он всегда вычисляет кратчайший путь от своей текущей ячейки до целевой ячейки в предположении , что ячейки с неизвестным статусом блокировки доступны для обхода. Затем он следует по этому пути до тех пор, пока не достигнет целевой ячейки, и в этом случае он успешно останавливается или наблюдает необратимое ячейку, в этом случае он вычисляет кратчайший путь от своей текущей ячейки до целевой ячейки.

На рисунке 1 показаны целевые расстояния всех проходимых ячеек и кратчайшие пути от его текущей ячейки до целевой ячейки как до, так и после перемещения робота по пути и обнаружения первой заблокированной ячейки, о которой он не знал.

Ячейки, расстояния до которых изменились, заштрихованы серым цветом. Расстояния до цели важны, потому что можно легко определить кратчайший путь от текущей ячейки робота до ячейки цели, жадно уменьшая расстояние до цели , как только расстояния до цели будут вычислены. Обратите внимание, что количество ячеек с измененными расстояниями до цели невелико, и большинство измененных расстояний до цели не имеют значения для повторного вычисления кратчайшего пути от текущей ячейки до цели ячейки. Таким образом, можно эффективно вычислить кратчайший путь от его текущей ячейки до ячейки цели, пересчитав только те расстояния до цели, которые изменились (или не были рассчитаны ранее) и имеют отношение к пересчету кратчайшего пути. Это то, что делает D * Lite. Задача состоит в том, чтобы эффективно идентифицировать эти клетки.

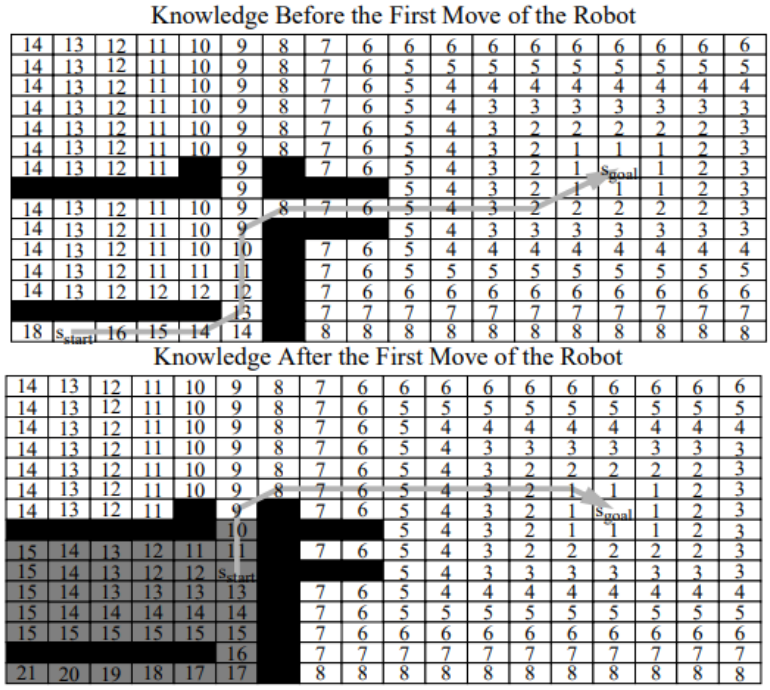


Figure 1: Simple Example.

Авторы и история

Алгоритм D* — алгоритм поиска кратчайшего пути во взвешенном ориентированном графе, где структура графа неизвестна заранее или постоянно подвергается изменению. Разработан Свеном Кёнигом и Максимом Лихачевым в 2002 году.

До того, как в 2002 году был разработан D* Lite, в области поиска пути доминировали другие алгоритмы, такие как A* и алгоритм Дейкстры. D* Lite отличается от этих алгоритмов оптимизации пути двумя основными способами.
Во-первых, Дейкстры и A* требуют от пользователя полного знания окружающей среды, поскольку они не предоставляют встроенных функций перепланировки. Это различие означает, что если робот сталкивается с чем-то неизвестным, алгоритмы A* и Дейкстры должны быть полностью перезапущены, чтобы был создан новый путь. С другой стороны, D* Lite быстро пересчитывает только пораженные участки, тем самым значительно сокращая необходимое время обработки. Когда узел обнаруживается как препятствие, стоимость прибытия туда становится бесконечной, и поэтому оценка RHS обновляется до бесконечности.
Во-вторых, алгоритмы A* и Дейкстры начинаются с генерации пути от начала до конца. D* Lite, с другой стороны, генерирует путь от цели до старта. Это различие полезно для перепланирования, потому что, когда путь необходимо восстановить, требуется меньше изменений при перемещении назад.

Описание алгоритма и реализации

Алгоритм D*Lite

Постановка задачи

Дан взвешенный ориентированный граф ***G(V,E)***.

Даны вершины: стартовая вершина ***f*** и конечная вершина ***t***.Требуется после каждого изменения графа ***G*** обновлять значения функции ***g(s)*** при поступлении новой информации о графе ***G***

Описание

Функция ***g(s)*** хранит минимальное известное расстояние ***t*** до ***s***.

Будем поддерживать для каждой вершины два вида смежных с ней вершин:

- Обозначим множество ***Succ(s)*** $\subseteq$ ***V*** как множество вершин, исходящих из вершины ***s***.
- Обозначим множество ***Pred(s)*** $\subseteq$ ***V*** как множество вершин,входящих в вершину ***s***.

Возвращает список состояний-преемников для состояния ***u*** , поскольку это 8-полосный график этот список содержит все соседние ячейки. Если ячейка не занята, и в этом случае у нее нет преемников.

```
1  void Dstar::getSucc(state u,list<state> &s) {
2
3      s.clear();
4      u.k.first  = -1;
5      u.k.second = -1;
6
7      if (occupied(u)) return;
8      u.x += 1;
9      s.push_front(u);
10     u.y += 1;
11     s.push_front(u);
12     u.x -= 1;
13     s.push_front(u);
14     u.x -= 1;
15     s.push_front(u);
16     u.y -= 1;
17     s.push_front(u);
18     u.y -= 1;
19     s.push_front(u);
20     u.x += 1;
21     s.push_front(u);
22     u.x += 1;
23     s.push_front(u);
24 }
```

Возвращает список всех состояний-предшественников для состояния ***u***. Поскольку это для 8-полосного связного графа, список содержит все соседи для состояния ***u***. Занятые соседи не добавляются в список.

```
1  void Dstar::getPred(state u,list<state> &s) {
2
3      s.clear();
4      u.k.first  = -1;
5      u.k.second = -1;
6
7      u.x += 1;
8      if (!occupied(u)) s.push_front(u);
9      u.y += 1;
10     if (!occupied(u)) s.push_front(u);
11     u.x -= 1;
12     if (!occupied(u)) s.push_front(u);
13     u.x -= 1;
14     if (!occupied(u)) s.push_front(u);
15     u.y -= 1;
16     if (!occupied(u)) s.push_front(u);
17     u.y -= 1;
18     if (!occupied(u)) s.push_front(u);
19     u.x += 1;
20     if (!occupied(u)) s.push_front(u);
21     u.x += 1;
22     if (!occupied(u)) s.push_front(u);
23
24 }
```

Функция $0 \leq$ ***c(s,s')*** $\leq +\infty$ будет возвращать стоимость ребра ***(s,s')***. При этом ***(s,s')***$=+\infty$ будет тогда и только тогда, когда ребра ***(s,s')*** не существует.
Возвращает стоимость перехода из состояния a в состояние b. Это может быть либо стоимость переезда из состояния a в состояние b,мы выбрали первое. Это также стоимость в 8 сторон

```
1  double Dstar::cost(state a, state b) {
2
3      int xd = fabs(a.x-b.x);
4      int yd = fabs(a.y-b.y);
5      double scale = 1;
6
7      if (xd + yd > 1) scale = M_SQRT2;
8
9      if (cellHash.count(a) == 0) return scale*C1;
10     return scale*cellHash[a].cost;
11 }
```

Определение Будем называть ***rhs***-значением (значение с правой стороны) такую функцию ***rhs(s)***, которая будет возвращать потенциальное минимальное расстояние от ***f*** до ***s*** по следующим правилам:

$$rhs(s) = \begin{cases} 0 & \text{если } s = f \\ min(g(s) + c(s, s'))_{s' \in Pred(s)} & \text{иначе} \end{cases}$$

```
1
2  double Dstar::getRHS(state u) {
3
4      if (u == s_goal) return 0;
5      if (cellHash.find(u) == cellHash.end())
6          return heuristic(u,s_goal);
7      return cellHash[u].rhs;
8  }
9  double Dstar::setRHS(state u, double rhs) {
10
11     makeNewCell(u);
12     cellHash[u].rhs = rhs;
13
14 }
```

Так как ***rhs***-значение использует минимальное значение из минимальных расстояний от ***f*** до вершин, входящих в данную вершину ***s***, это будет нам давать информацию об оценочном расстоянии от ***f*** до ***s***.

Определение: Вершина ***s*** называется ***насыщенной***, если $g(s) = rhs(s)$

Определение: Вершина ***s*** называется ***переполненной***, если $g(s) > rhs(s)$

Определение: Вершина ***s*** называется ***ненасыщенной***, если $g(s) < rhs(s)$

```
1  //Возвращает значение G длясостояния  u.
2  double Dstar::getG(state u) {
3      if (cellHash.find(u) == cellHash.end())
4          return heuristic(u,s_goal);
5      return cellHash[u].g;
6  }
7  void Dstar::setG(state u, double g) {
8
9      makeNewCell(u);
10     cellHash[u].g = g;
11 }
```

Очевидно, что если все вершины насыщены, то мы можем найти расстояние от стартовой вершины до любой. Такой граф будем называть устойчивым (насыщенным).

Определение: Эвристическая функция ***h(s,s')*** должна поддерживать неравенство треугольника для всех вершин ***s,s',s''*** $\in V$, т.е. $h(s, s'') \leq h(s, s') + h(s', s'')$ Так же должно выполняться свойство $(s, s') \leq c*(s, s'), c*(s, s')$, где $c*(s, s')$ - стоимость

перехода по кратчайшему пути из ***s в s'***, при этом ***s и s'*** не должны быть обязательно смежными.
Возвращает 8-полосное расстояние между состоянием a и состоянием b.

```
1  double Dstar::eightCondist(state a, state b) {
2      double temp;
3      double min = fabs(a.x - b.x);
4      double max = fabs(a.y - b.y);
5      if (min > max) {
6          double temp = min;
7          min = max;
8          max = temp;
9      }
10     return ((M_SQRT2-1.0)*min + max);
11 }
```

Само собой разумеется, что эвристика, которую мы используем, - это расстояние в 8 направлениях масштабируется на константу C1 (должно быть установлено значение больше <= min cost).

```
1  double Dstar::heuristic(state a, state b) {
2      return eightCondist(a,b)*C1;
3  }
```

Определение: Будем называть ключом вершины такую функцию ***key(s)***, которая возвращает вектор из 2-ух значений ***k1(s)***, ***k2(s)***.

- $k1(s) = min(g(s), rhs(s)) + h(s,t)$
- $k2(s) = min(g(s), rhs(s))$
- где ***s*** - вершина из множества ***V***

```
1  state Dstar::calculateKey(state u) {
2
3      double val = fmin(getRHS(u),getG(u));
4      u.k.first  = val + heuristic(u,s_start) + k_m;
5      u.k.second = val;
6      return u;
7  }
```

Если в конце поиска пути $g(t) = +\infty$, путь не найден на данной итерации. Иначе путь найден и ”робот” может проследовать по нему.

Примечание: Так же следует отметить, что функция $init$ не обязана инициализировать абсолютно все вершины перед стартом алгоритма. Это важно, так как на практике число вершин может быть огромным, и только немногие будут пройдены роботом в процессе движения. Так же это дает возможность добавления/удаления ребер без потери устойчивости всех подграфов данного графа.
Инициализация d star начинается с координат начала и цели, остальное в соответствии с документацией [S. Koenig, 2002]

```
1  void Dstar::init(int sX, int sY, int gX, int gY) {
2
3      cellHash.clear();
4      path.clear();
5      openHash.clear();
6      while(!openList.empty()) openList.pop();
7
8      k_m = 0;
9
10     s_start.x = sX;
11     s_start.y = sY;
12     s_goal.x  = gX;
13     s_goal.y  = gY;
14
15     cellInfo tmp;
16     tmp.g = tmp.rhs =  0;
17     tmp.cost = C1;
18
19     cellHash[s_goal] = tmp;
20
21     tmp.g = tmp.rhs = heuristic(s_start,s_goal);
22     tmp.cost = C1;
23     cellHash[s_start] = tmp;
24     s_start = calculateKey(s_start);
25
26     s_last = s_start;
27 }
```

Функция (updateVertex): Если некоторые затраты на ребро изменились,вызывается $UpdateVertex()$, чтобы обновить значения rhs и ключи вершин, потенциально затронутых измененными затратами на ребро, а также их членство в очереди приоритетов, если они становятся ***насыщенными*** или ***ненасыщенными***

```
1  void Dstar::updateVertex(state u) {
2      list<state> s;
3      list<state>::iterator i;
4
5      if (u != s_goal) {
6          getSucc(u,s);
7          double tmp = INFINITY;
8          double tmp2;
9
10         for (i=s.begin();i != s.end(); i++) {
11             tmp2 = getG(*i) + cost(u,*i);
12             if (tmp2 < tmp) tmp = tmp2;
13         }
14         if (!close(getRHS(u),tmp)) setRHS(u,tmp);
15     }
16     if (!close(getG(u),getRHS(u))) insert(u);
17 }
```

Обновить позицию робота, это не приведет к принудительному воспроизведению.

```
1  void Dstar::updateStart(int x, int y) {
2
3      s_start.x = x;
4      s_start.y = y;
5
6      k_m += heuristic(s_last, s_start);
7
8      s_start = calculateKey(s_start);
9      s_last  = s_start;
10 }
```

```
1  void Dstar::updateCell(int x, int y, double val) {
2
3      state u;
4
5      u.x = x;
6      u.y = y;
7
8      if ((u == s_start) || (u == s_goal)) return;
9
10     makeNewCell(u);
11     cellHash[u].cost = val;
12
13     updateVertex(u);
14 }
```

Теорема (О монотонности изменения ключей): В течение выполнения функции $ComputeShortestPath$ вершины, взятые из очереди, монотонно не убывают.

Теорема (О необратимой насыщенности): Если в функциибыла $ComputeShortestPath$ взята переполненная вершина, то на следующей итерации она станет насыщенной.

Теорема После выполнения функции $ComputeShortestPath$ можно восстановить путь из ft. Для этого, начиная с вершины t, нужно постоянно передвигаться к такой вершине s', входящей в t, чтобы $g(s')+c(s',s)$ было минимальным, до тех пора, пока не будет достигнута вершина f.

Теорема (Свен Кёниг, Об устойчивой насыщенности вершин): Функция $ComputeShortestPath$ в данной версии алгоритма расширяет вершину максимум 2 раза, а именно 1 раз, если вершина ненасыщена, и максимум 1 раз, если она переполнена.

- Прекращаем планирование после нескольких шагов, ”maxsteps” мы делаем это потому что этот алгоритм может планировать вечно, если начало окруженны препятствиями.
- Лениво удаляем состояния из открытого списка, чтобы нам никогда не приходилось перебирать его.

```
1  Dstar::Dstar() {
2      maxSteps = 80000;   // расширенияузлов
3      C1       = 1;       // стоимостьневидимойячейки
4  }
```

```
1  int Dstar::computeShortestPath() {
2
3      list<state> s;
4      list<state>::iterator i;
5
6      if (openList.empty()) return 1;
7
8      int k=0;
9      while ((!openList.empty()) &&
10            (openList.top() < (s_start = calculateKey(
                 s_start))) ||
11            (getRHS(s_start) != getG(s_start))) {
12
13         if (k++ > maxSteps) {
14             fprintf(stderr, "At maxsteps\n");
15             return -1;
16         }
17
18         state u;
19
20         bool test = (getRHS(s_start) != getG(s_start));
21
22         // ленивоеудаление
23         while(1) {
24             if (openList.empty()) return 1;
25             u = openList.top();
26             openList.pop();
27
28             if (!isValid(u)) continue;
29             if (!(u < s_start) && (!test)) return 2;
30             break;
31         }
32         ds_oh::iterator cur = openHash.find(u);
33         openHash.erase(cur);
34
35         state k_old = u;
36
37         if (k_old < calculateKey(u)) { // u устарел
38             insert(u);
39         } else if (getG(u) > getRHS(u)) { //
               нуждаетсявобновлениистало  ( лучше)
40             setG(u,getRHS(u));
41             getPred(u,s);
42             for (i=s.begin();i != s.end(); i++) {
43                 updateVertex(*i);
44             }
```

```
45          } else {    // g <= rhs, состояниеухудшилось
46              setG(u,INFINITY);
47              getPred(u,s);
48              for (i=s.begin();i != s.end(); i++) {
49                  updateVertex(*i);
50              }
51              updateVertex(u);
52          }
53      }
54      return 0;
55  }
```

Открытый хэш и отложенное удаление Чтобы ускорить время, необходимое для добавления / удаления / поиска в открытом списке, мы использовали как $stl :: priority_queue$, так и $stl :: hash_map$ для хранения состояний.
Очередь сохраняет состояния в отсортированном порядке, поэтому легко найти следующее лучшее состояние, в то время как хэш используется для быстрого определения того, какие состояния находятся в очереди. Когда ячейка вставляется в $openlist$, она помещается в очередь и помещается в хеш-таблицу. Чтобы проверить, есть ли ячейка в открытом списке, можно просто проверить, есть ли она в хеш-таблице.
В хэш-таблице также хранится хэш ключа ячеек, поэтому ячейки, которые устарели в очереди, все еще могут быть удалены. Каждый раз, когда ячейка извлекается из очереди, мы проверяем, есть ли она в хэше, если нет, она отбрасывается и выбирается новая.

Возвращает хэш-код ключа для состояния u, который используется для сравнения состояние, которое было обновлено

```
1  float Dstar::keyHashCode(state u) {
2      return (float)(u.k.first + 1193*u.k.second);
3  }
```

Возвращает значение true, если состояние u находится в открытом списке или нет, проверяя, есть ли оно в хэш-таблице.

```
1  bool Dstar::isValid(state u) {
2      ds_oh::iterator cur = openHash.find(u);
3      if (cur == openHash.end()) return false;
4      if (!close(keyHashCode(u), cur->second)) return false;
5      return true;
6  }
```

Проверяет, есть ли ячейка в хэш-таблице, если нет, то добавляет ее в нее

```
1  void Dstar::makeNewCell(state u) {
2      if (cellHash.find(u) != cellHash.end()) return;
3      cellInfo tmp;
4      tmp.g       = tmp.rhs = heuristic(u,s_goal);
5      tmp.cost    = C1;
6      cellHash[u] = tmp;
7  }
```

Возвращает значение true, если ячейка занята (недоступна для обхода),в противном случае значение false. непроходимые отмечены стоимостью < 0.

```
1  bool Dstar::occupied(state u) {
2
3      ds_ch::iterator cur = cellHash.find(u);
4
5      if (cur == cellHash.end()) return false;
6      return (cur->second.cost < 0);
7  }
```

Оптимизация евклидова пути Получение пути из сгенерированной D* карты затрат обычно выполняется путем запуска с начального узла и выполнения жадного поиска путем расширения последующих узлов с наименьшими затратами до цели.

Этот подход может генерировать траекторию, которая начинает двигаться под углом 45 градусов к цели, а не прямо к ней. Это происходит потому, что 8-полосное связное расстояние является приблизительным, и нет никакой разницы между тем, чтобы сначала выполнять все угловые ходы и выполнять все прямые ходы. Чтобы сгенерировать путь, который ближе к истинной наименьшей стоимости, мы добавили простую модификацию в жадный поиск. Когда мы сравниваем затраты со всеми последующими ячейками, мы выбираем ту, которая минимизирует:

Евклидова стоимость между состоянием a и состоянием b.

```
1  double Dstar::trueDist(state a, state b) {
2
3      float x = a.x-b.x;
4      float y = a.y-b.y;
5      return sqrt(x*x + y*y);
6
7  }
```

```
1  bool Dstar::replan() {
2
3      path.clear();
4      int res = computeShortestPath();
5      FILE *file;
6      file=fopen("out.txt","w+t");
7      fprintf(file ,"res:%d ols:%d ohs:%d tk:[%f %f] sk:
           [%f %f] sgr:(%f,%f)\n",res,openList.size(),
           openHash.size(), openList.top().k.first, openList.
           top().k.second, s_start.k.first, s_start.k.second,
           getRHS(s_start), getG(s_start));
8
9      if (res < 0) {
10         return false;
11     }
12     list<state> n;
13     list<state>::iterator i;
14
```

```
15      state cur = s_start;
16
17      if (isinf(getG(s_start))) {
18          return false;
19      }
20
21      while(cur != s_goal) {
22
23          path.push_back(cur);
24          getSucc(cur, n);
25
26          if (n.empty()) {
27              return false;
28          }
29
30          double cmin = INFINITY;
31          double tmin;
32          state smin;
33
34          for (i=n.begin(); i!=n.end(); i++) {
35              double val  = cost(cur,*i);
36              double val2 = trueDist(*i,s_goal) + trueDist(
                    s_start,*i);
37              val += getG(*i);
38
39              if (close(val,cmin)) {
40                  if (tmin > val2) {
41                      tmin = val2;
42                      cmin = val;
43                      smin = *i;
44                  }
45              } else if (val < cmin) {
46                  tmin = val2;
47                  cmin = val;
48                  smin = *i;
49              }
50          }
51          n.clear();
52          cur = smin;
53      }
54      path.push_back(s_goal);
55      return true;
56  }
```

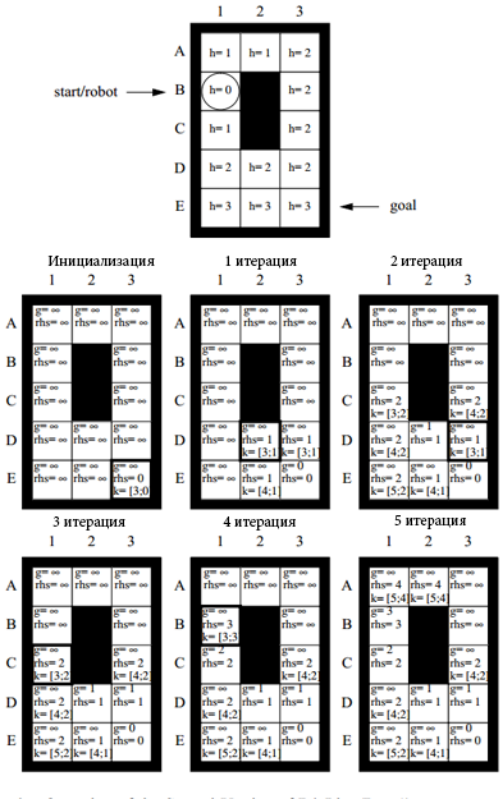
Main Функция $Main$ где мы принимаем входные данные.

- устанавливаем координаты начала и конца
- задаем координаты непроходимой ячейки
- задаем стоимость одной из координат ячеки
- планируем путь, получаем путь
- изменяем начало на другие координаты
- изменяем цель на другие координаты

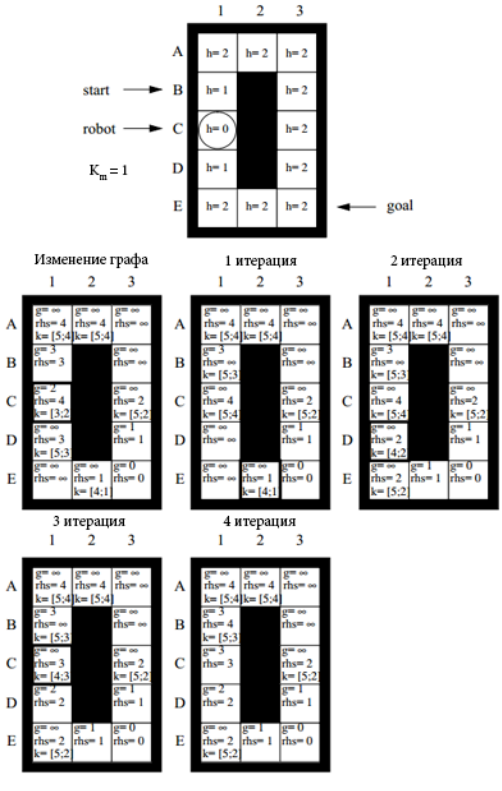
```
1  int main() {
2      Dstar *dstar = new Dstar();
3      list<state> mypath;
4      FILE *file;
5      file = fopen("test1.txt", "r");
6
7      int n1;
8      int arr[n1];
9      for (int i = 0; i < n1; i++){
10         fscanf(file, "%d,", &arr[i] );
11     }
12     dstar->init(arr[0],arr[1],arr[2],arr[3]);
13     dstar->updateCell(arr[4],arr[5],-1);
14     dstar->updateCell(arr[6],arr[7],42.432);
15
16     dstar->replan();
17     mypath = dstar->getPath();
18
19     dstar->updateStart(arr[8],arr[9]);
20     dstar->replan();
21     mypath = dstar->getPath();
22
23     dstar->updateGoal(arr[10],arr[11]);
24     dstar->replan();
25     mypath = dstar->getPath();
26     fclose(file);
27     return 0;
28 }
```

Пример работы

Итерации в функции $ComputeShortestPath$ на исходном графе.



Итерации в функции $ComputeShortestPath$ после изменения графа. (Второй вызов функции)



Тестирование и анализ производительности

При измерение времени и работы алгоритма статично выдавало 14ms. При увеличении размера расширения узлов с 8000 до 80 000 000 компилятор стабильно выводил 14ms. В test1.txt числа не превышали <=10. Тогда создав 8 тестов в зависимости от размера взодящих данных и их увелечения на графике можно увидить как увеличивалось и время выполнения алгоритма.

Список литературы

References

[1] S. Koenig and M. Likhachev. *D* Lite*. 2002. URL: http://idm-lab.org/bib/abstracts/papers/aimag04a.pdf.

[2] Sven Koenig. *Project "Fast Replanning (Incremental Heuristic Search)"*. URL: http://idm-lab.org/project-a.html.

[3] Maxim Likhachev b Sven Koenig a and David Furcy c. *Lifelong Planning A**. URL: https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf.

[4] *D-STAR*. URL: https://en.wikipedia.org/wiki/D*.

[5] *Алгоритм D**. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_D*.