



**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 задании по дисциплине АИСД
«Алгоритм поиска D*Lite»**

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

Выполнил студент
гр. Б9121-09.03.03 пикд
Масличенко Елизавета Андреевна

Доклад защищен:
С оценкой _____

Руководитель практики
Доцент ИМКТ А.С. Кленин

г. Владивосток
2023

Оглавление

1 Аннотаци

2 Постановка задачи

3 Авторы и история

4 Описание алгоритма и реализация

4.1 Концепция предшественников и приемников

4.2 Поиск по принципу "best-first search"

4.3 Обработка изменения веса локально

4.4 Локальные несоответствия

4.5 Обновление и расширение узлов

4.6 Сверхсогласованный узел

4.7 Недостаточно согласованный узел

4.8 Отслеживание изменения эвристики

4.9 Общий D* Lite(конкретизация):

5 Пример работы алгоритма

6 Тестирование и анализ производительности

7 Заключение

8 Список литературы

Аннотация

В данной работе представлен *DstarLite*-алгоритм инкрементного эвристического поиска. Инкрементный поиск повторно использует информацию из предыдущих результатов поиска. Эвристический поиск предполагает предположительное расстояние от начала до цели. Поэтому в данной работе мы уделяем особое внимание поэтапному планированию и его реализации.

Постановка задачи

Дан взвешенный ориентированный граф $G(V, E)$. Даны вершины стартовая и конечная. Требуется в процессе движения по кратчайшему пути в графе G обновлять значения функции $g(s)$ при поступлении новой информации о графе G . Ориентированным графом G называется пара $G = (V, E)$, где V — множество вершин, а $E \in V \times V$ — множество рёбер. Взвешенным графом называется граф, вершинам или ребрам которого присвоены «весы» — некоторые числа.

Авторы и история

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

Применение

Dstar и его варианты широко использовались для мобильной навигации роботов и автономных транспортных средств. Современные системы, как правило, основаны на *DstarLite*, а не на оригинальном *Dstar* или сфокусированном *Dstar*. Фактически, даже лаборатория Стенца использует *DstarLite*, а не *Dstar* в некоторых реализациях. К таким навигационным системам относятся прототипная система, протестированная на марсоходах Opportunity и Spirit, и навигационная система победителя конкурса DARPA Urban Challenge, разработанные в Университете Карнеги-Меллона.

Описание алгоритма и реализация

DstarLite-рассматривают с двух точек зрения.

Первая точка зрения рассматривает модификацию основного пространства. При первой итерации, до каких либо изменений алгоритм ведет себя как *Astar*, а именно происходит поиск от начальной вершины к конечной и добавляется в очередь по формуле

$$f(s) = g(s) + h(s, s_{goal})$$

- $f(s)$ -общая стоимость вершины
- $g(s)$ - стоимость перехода от одного узла к другому
- $h(s, s_{goal})$ -эвристика стоимость получения вершины от s, s_{goal}

Вторая точка зрения рассматривает “ослабление” веса ребер. Метод заключается в том что бы выбирать только те вершины, которые мы хотим эффективно и рационально использовать.

Различие двух узлов с одинаковой общей стоимостью в очереди.

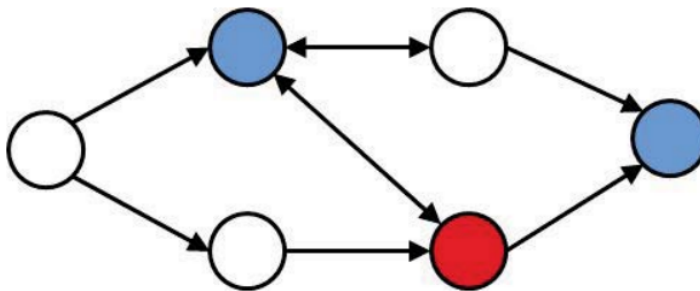
Итак мы вводим пару, состоящую из 2 значений $f(s) = \langle f_1(s), f_2(s) \rangle$

- $f_1(s) = g(s) + h(s, s_{goal})$.
- $f_2(s) = g(s)$ - стоимость перехода от одной вершины к другой
- $f(s) = \langle g(s) + h(s, s_{goal}), g(s) \rangle$

Таким образом порядок расширения узлов из очереди сначала упорядочивает их по значению f_1 и если два узла имеют одинаковое значение f_2 , мы выбираем узел с наименьшим значением $g(s)$

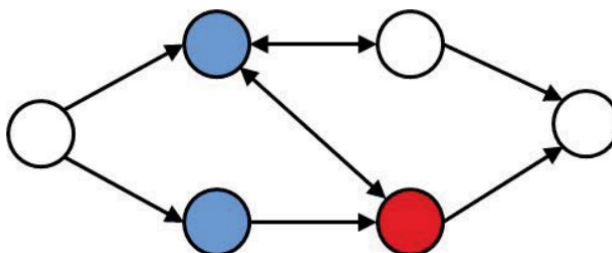
Концепция предшественников и приемников

- $Succ(s)$ - множество вершин исходящих из вершины s . Таким образом приемниками($Succ(s)$) красной вершины, являются вот эти две синих вершины.



•

- $Pred(s)$ - множество вершин входящих в вершину s . Таким образом предшественниками($Pred(s)$) красной вершины, являются вот эти две синих вершины.



•

Поиск по принципу “best-first search”

Dstar Lite использует поиск по принципу “best-first search”. Алгоритм поиска, исследующий граф путём расширения наиболее перспективных узлов, выбираемых в соответствии с указанным правилом.

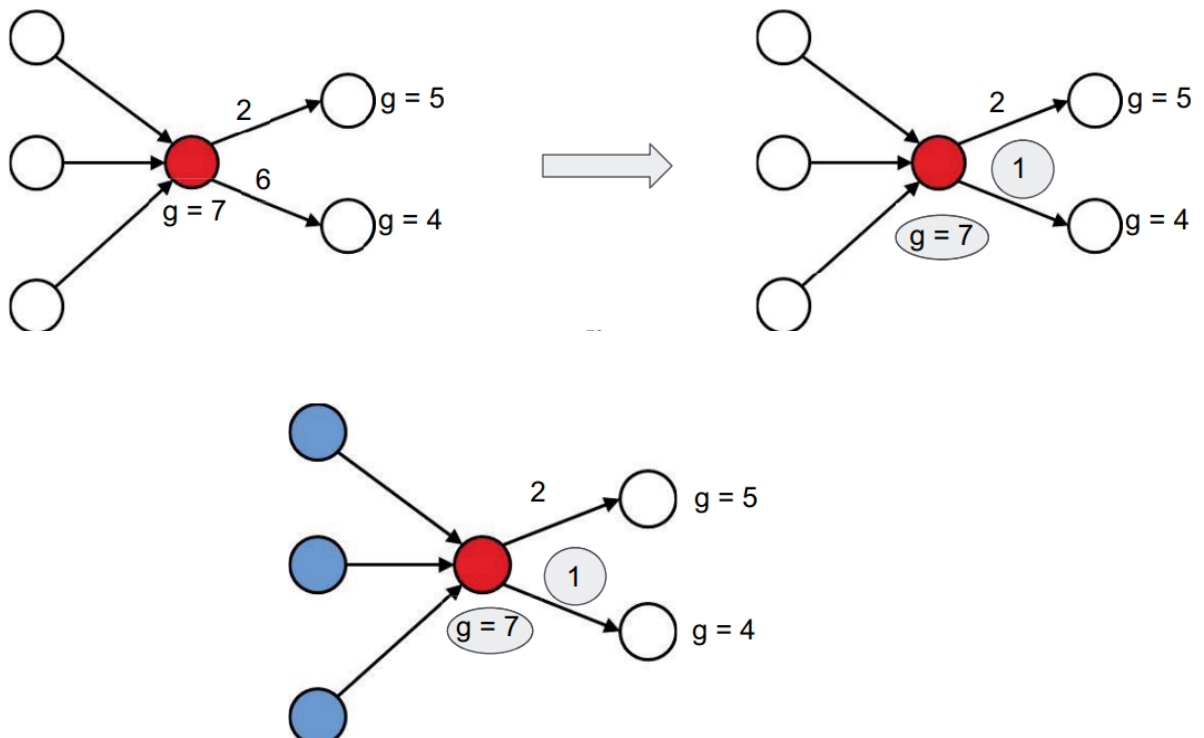
Best-First-Search(Graph g , Node start)

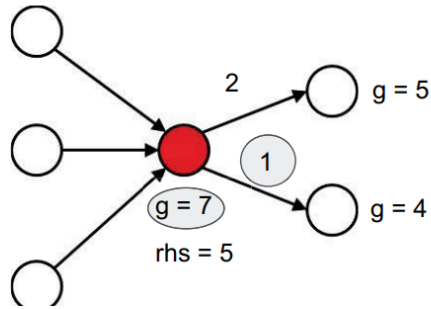
1. Создайте пустую PriorityQueue

- PriorityQueue pq;
2. Insert "start" in pq.
 - pq.insert(start)
 3. До тех пор пока PriorityQueue не станет пустой
 - u = PriorityQueue.DeleteMin
 - If u is the goal
 - Exit
 - Else
 - Foreach neighbor v of u
 - * If v "не посещена"
 - * Mark v "Посещена"
 - * pq.insert(v)
 - Пометить как u "Исследованный"

Обработка изменения веса локально

Пример заключается в том что на этапе планирования пути не сразу изменять значение $g(s)$. На графике показано изменение стоимости ребер с 6 на 1. Новое значение $g(s) = 5$. Но пока это не сохраняется.





Что бы решить проблему, мы собираемся хранить дополнительное значение

$$rhs(s) = \min_{s' \in Succ(s)} (g(s') + (s, s'))$$

(англ. right-hand side value) В данном случае $rhs(s) \neq g(s)$ становится **локально несогласованным**.

Локальные несоответствия

У нас есть 2 типа локального несоответствия

- $g(s) > rhs(s)$ локально сверхсогласованно
- $g(s) < rhs(s)$ локально недостаточно согласованно

Обновление и расширение узлов

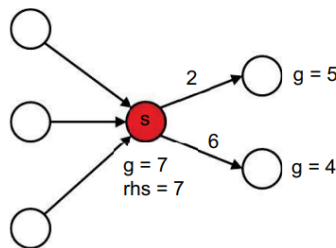


рис.1

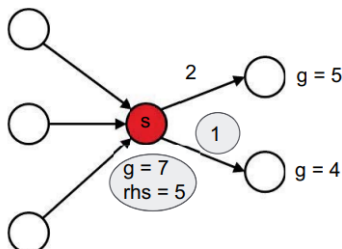


рис.2

Под "обновлением узлов" я подразумеваю повторное вычисление значения $rhs(s)$ а затем добавление этой вершины в очередь приоритетов, если эта вершина локально несовместима. Что бы привести пример я изменила значение стоимости ребра с 6 на 1 предварительно подсчитав на рисунке 1 значение $rhs(s) = 7$ до изменений и на рисунке 2 после изменений $rhs(s) = 5$. Теперь мы видим на рис.2, что узел локально несогласован.

И поэтому мы помещаем его в очередь приоритетов.рис.3

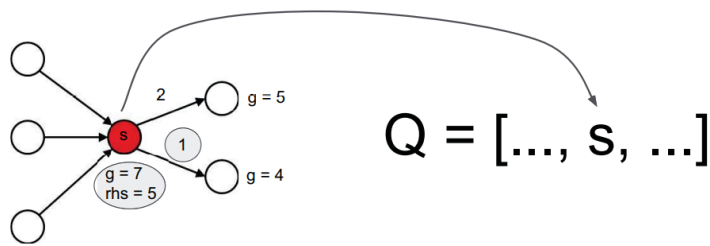


рис.3

Значение приоритетной очереди основанно на этой формуле
 $Q = \langle \min[g(s), rhs(s)] + h(s, s_{start}), \min[g(s), rhs(s)] \rangle$

Суть заключается в том чтобы взять минимальное значение одно из параметров $g(s)$, $rhs(s)$ Порядок очереди заключается в принципе от меньшего к большему.

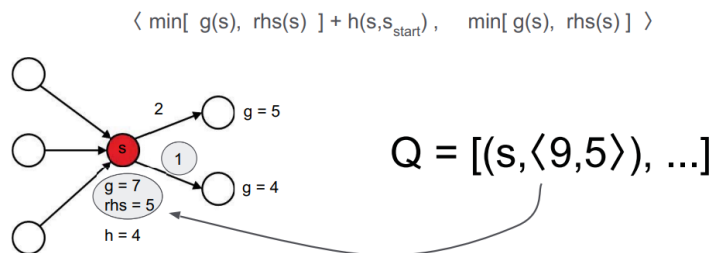


рис.4

Расширяем предыдущие вершины удаляя из очереди приоритетов и изменяя значение $g(s)$. Значения $g(s)$ уменьшаются до значения $rhs(s)$, поэтому мы просто устанавливаем $g(s) = rhs(s)$. В этом случае вершина локально непротеричива(согласована) и она остается такой.

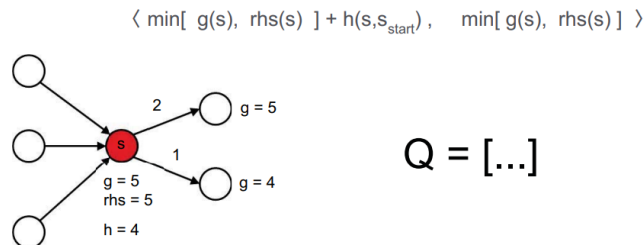


рис.5

```

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 }

```

Сверхсогласованный узел

$$g(s) > rhs(s)$$

- Стоимость нового пути $rhs(s)$ лучше, чем стоимость старого пути $g(s)$.
- Немедленно обновите $g(s)$ до $rhs(s)$ и распространите на всех предшественников
- Установить $g(s) = rhs(s)$
- Обновление всех предшественников узла s

Узел теперь локально согласован и он останется таким.

Недостаточно согласованный узел

$$g(s) < rhs(s)$$

- Стоимость старого пути $g(s)$ лучше, чем стоимость нового пути $rhs(s)$
- Задержите расширение вершины и распространите на всех предшественников
- Установить $g(s) = \infty$
- Обновление всех предшественников узла s и сам s

Вершина теперь локально согласована или локально сверхсогласованна. Она будет оставаться в очереди до тех пор, пока $rhs(s)$ не станет следующей наилучшей стоимостью. Логика того, к какому типу относиться значение $rhs(s)$ реализована в функции *computeShortestPath()*

```
1  int Dstar::computeShortestPath() {
2
3      list<state> s;
4      list<state>::iterator i;
5
6      if (openList.empty()) return 1;
7      int k=0;
8      while ((!openList.empty()) &&
9              (openList.top() < (s_start = calculateKey(
10                  s_start))) ||
11              (getRHS(s_start) != getG(s_start))) {
12
13          if (k++ > maxSteps) {
14              return -1;
15          }
16          state u;
17          bool test = (getRHS(s_start) != getG(s_start));
18          while(1) {
19              if (openList.empty()) return 1;
20              u = openList.top();
21              openList.pop();
22              if (!isValid(u)) continue;
```



```

23         if (!(u < s_start) && (!test)) return 2;
24         break;
25     }
26     ds_oh::iterator cur = openHash.find(u);
27     openHash.erase(cur);
28     state k_old = u;
29     if (k_old < calculateKey(u)) {
30         insert(u);
31     } else if (getG(u) > getRHS(u)) {
32         setG(u, getRHS(u));
33         getPred(u, s);
34         for (i=s.begin(); i != s.end(); i++) {
35             updateVertex(*i);
36         }
37     } else {
38         setG(u, INFINITY);
39         getPred(u, s);
40         for (i=s.begin(); i != s.end(); i++) {
41             updateVertex(*i);
42         }
43         updateVertex(u);
44     }
45 }
46 return 0;
47 }

```

В строке 8 в цикле while происходит проверка локальной несогласованности узла. С 29 по 42 строчку проверка к какому типу относиться локально несогласованный узел.

Отслеживание изменения эвристики

Ключевой модификатор в котором будем хранить сумму после каждой смены позиции начальной вершины.

$$k_m = k_m + h(s_{last}, s_{start})$$

```

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 }
12 double Dstar::heuristic(state a, state b) {
13     return eightCondist(a,b)*C1;
14 }
15 }

```

Формула приоритетной очереди изменяется:

$$< \min[g(s), rhs(s)] + h(s, s_{start}) + k_m, \min[g(s), rhs(s)] >$$

```

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

Общий D* Lite(конкретизация):

1. Инициализация всех значений $g(s) = \infty$, $rhs(s, s_{goal}) = \infty$, $rhs(s_{goal}) = 0$, $k_m = 0$, $s_{last} = s_{start}$,
2. Поиск по принципу "best-first search" (лучший — первый), пока s_{start} не будет локально согласован и расширен
3. Переместить s_{start} в зависимости от результатов приоритетной очереди
4. Если какие-либо стоимости ребер изменились
 - (a) $k_m = k_m + h(s_{last}, s_{start})$
 - (b) Обновите $rhs(s)$ и положение в очереди для исходных вершин измененных ребер
5. Повторите шаг № 2

Всегда сортируйте по формуле:

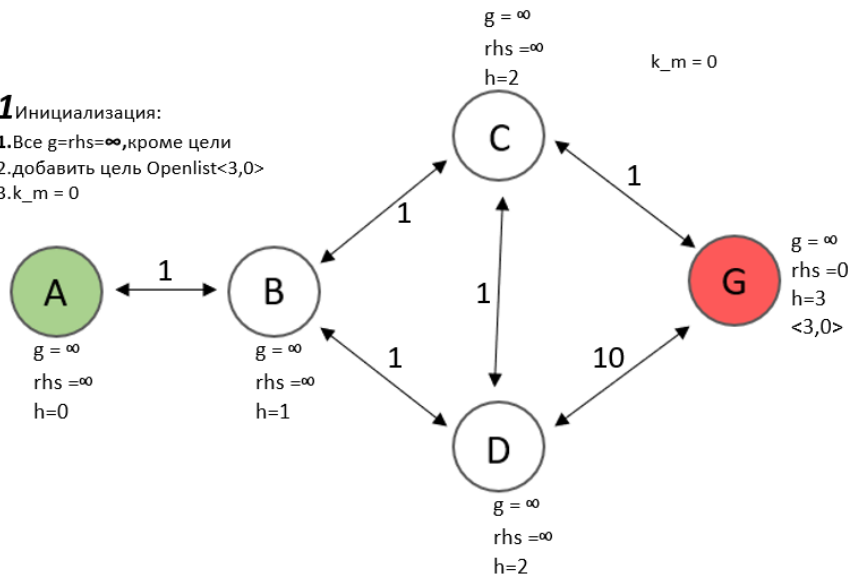
$$< \min[g(s), rhs(s)] + h(s, s_{start}) + k_m, \min[g(s), rhs(s)] >$$

Пример работы алгоритма

Рассмотрим краткий пример с пятью вершинами. Робот начинает с вершины А и его цель найти кратчайший путь к вершины G, за исключением того что граф может измениться по мере прохождением робота или робот может получить новую информацию. Это двунаправленный граф, каждое ребро идет в обоих направлениях. Эвристика - это количество вершин до начальной вершины. Все вершины локально согласованны так как $g(s) = rhs(s)$

1 Инициализация:

1. Все $g=rhs=\infty$, кроме цели
2. добавить цель $Openlist\langle 3,0 \rangle$
3. $k_m = 0$



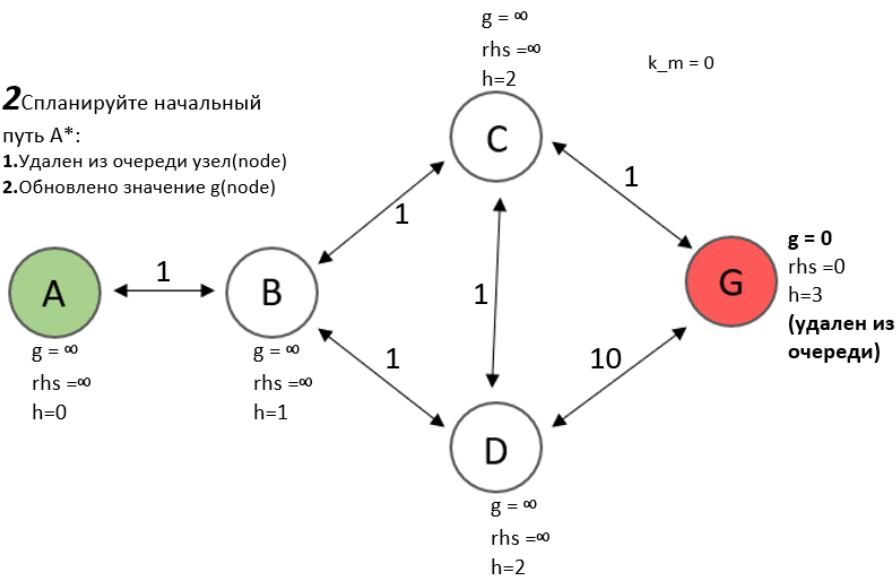
Мы помещаем вершину G в Openlist(очередь) используя формулу $\langle \min[g(s), rhs(s)] + h(s, s_{start}) + k_m, \min[g(s), rhs(s)] \rangle$. Теперь в очереди находится одна вершина. Так что нам придется выйти из очереди.

- удаляем узел G из OpenList
- проверка $computeShortestPath(g(s) > rhs(s))$, следовательно $g(s) = rhs(s) = 0$

2 Спланируйте начальный

путь A*:

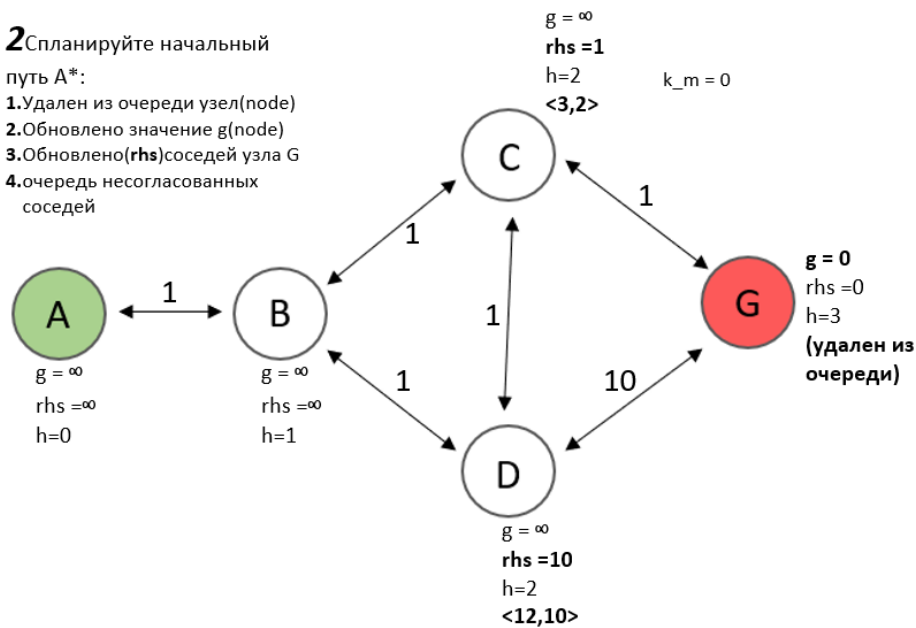
1. Удален из очереди узел(node)
2. Обновлено значение $g(node)$



Теперь мы собираемся обновить значения $rhs(s)$ соседей вершин G и в каждом случае значение будет минимальным по формуле $rhs(s) = \min_{s' \in Succ(s)} (g(s') + (s, s'))$. И добавить в очередь $OpenList = [\langle 3, 2 \rangle; \langle 12, 10 \rangle]$

2 Спланируйте начальный путь A*:

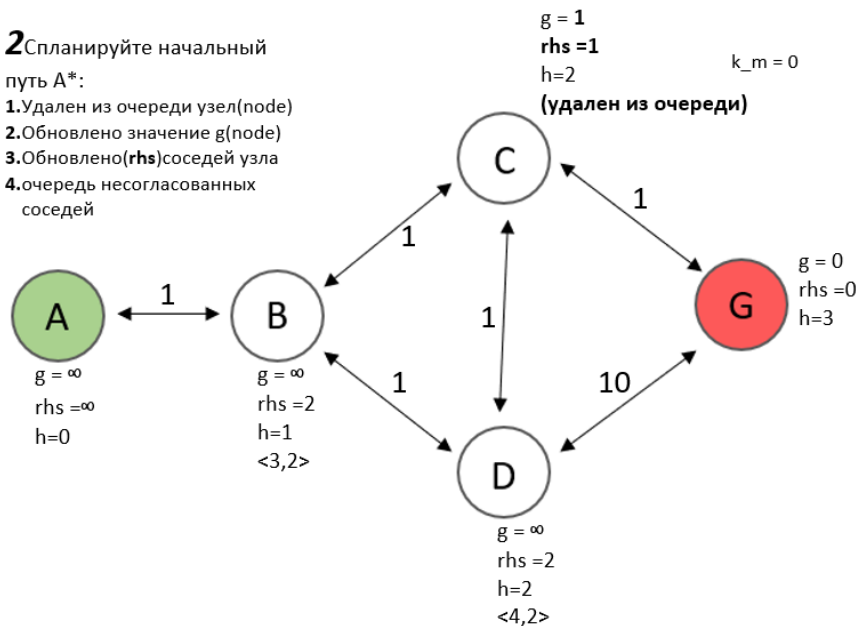
1. Удален из очереди узел (node)
2. Обновлено значение $g(\text{node})$
3. Обновлено (rhs) соседей узла G
4. очередь несогласованных соседей



Теперь мы исключим из очереди еще одну вершину C так как у него наименьший ключ $\langle 3, 1 \rangle$. Проверка $\text{computeShortestPath}(g(s) > \text{rhs}(s))$, следовательно $g(s) = \text{rhs}(s) = 1$. И обновим значения $\text{rhs}(s)$ соседей вершины C. Теперь в очереди $\text{OpenList} = [\langle 3, 2 \rangle; \langle 4, 2 \rangle]$

2 Спланируйте начальный путь A*:

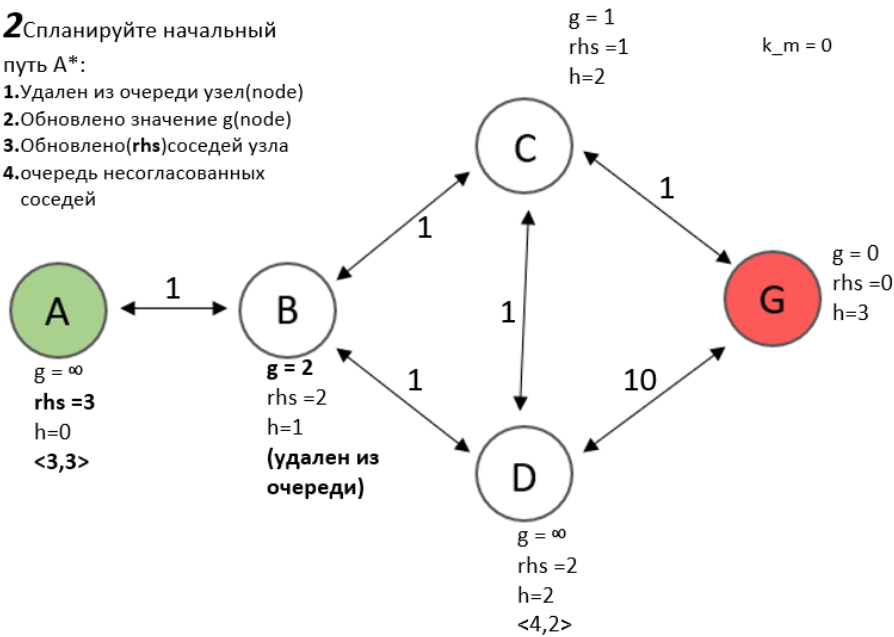
1. Удален из очереди узел (node)
2. Обновлено значение $g(\text{node})$
3. Обновлено (rhs) соседей узла
4. очередь несогласованных соседей



Дальше мы удаляем из очереди вершину B так как у него наименьший ключ $\langle 3, 2 \rangle$. Проверка $\text{computeShortestPath}(g(s) > \text{rhs}(s))$, следовательно $g(s) = \text{rhs}(s) = 2$. И обновим значения $\text{rhs}(s)$ соседей вершины B. Мы не изменяем ключевое значение вершины D так как стоимость из $G \rightarrow C \rightarrow D$ будет меньше чем из $G \rightarrow C \rightarrow B \rightarrow D$. Теперь в очереди $\text{OpenList} = [\langle 3, 3 \rangle; \langle 4, 2 \rangle]$

2 Спланируйте начальный путь A^* :

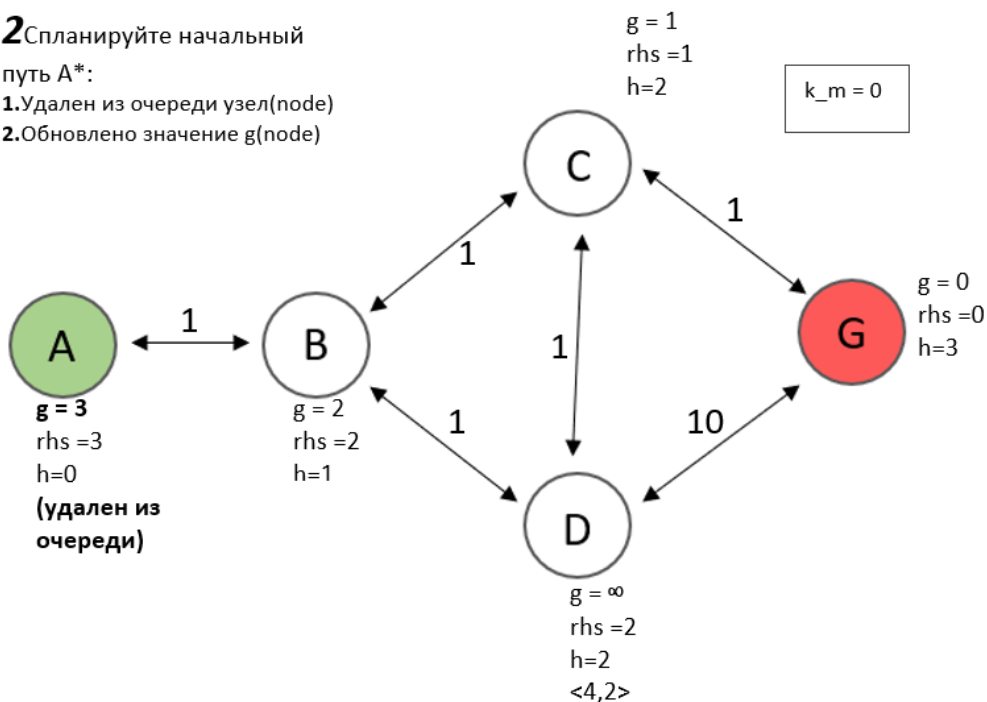
1. Удален из очереди узел (node)
2. Обновлено значение $g(node)$
3. Обновлено (rhs) соседей узла
4. очередь несогласованных соседей



Далее мы удаляем из очереди вершину A так как наименьший ключ. Проверка $computeShortestPath(g(s) > rhs(s))$, следовательно $g(s) = rhs(s) = 3$. $OpenList = [<4, 2>]$ И на это мы закончили.

2 Спланируйте начальный путь A^* :

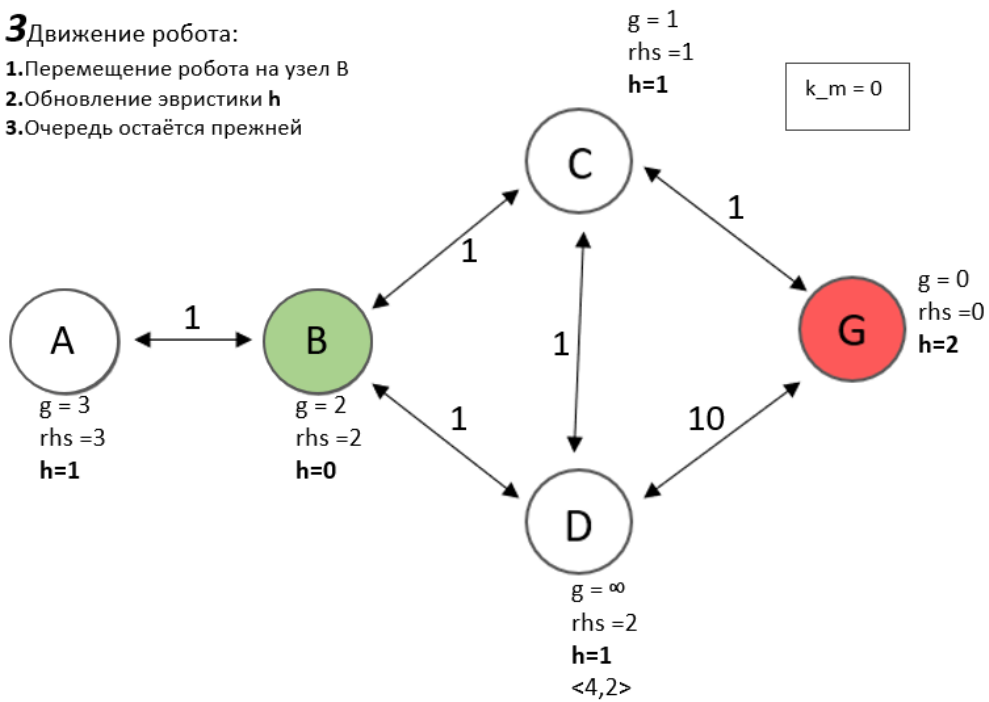
1. Удален из очереди узел (node)
2. Обновлено значение $g(node)$



Робот переместился к своей лучшей вершине B . Мы обновляем значения эвристики h для каждой вершины в графе. Мы не изменяем очередь, но также должны обновить $k_m = 1$ Появляется препятствие. В точке C есть препятствие, которое робот теперь может видеть так как он ближе переместился к вершине C , но раньше он его не видел. Мы укажем эти изменения изменив ребра вершины C на стоимость равной бесконечность. Это означает что робот никогда не сможет туда добраться.

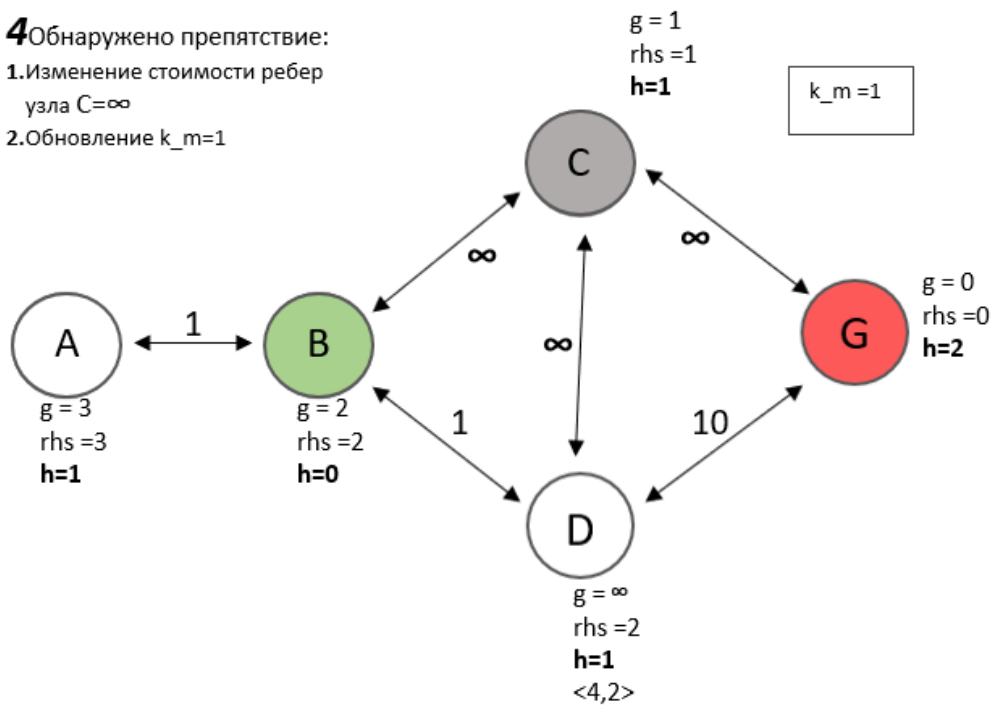
3 Движение робота:

1. Перемещение робота на узел B
2. Обновление эвристики h
3. Очередь остаётся прежней



4 Обнаружено препятствие:

1. Изменение стоимости ребер узла $C = \infty$
2. Обновление $k_m = 1$

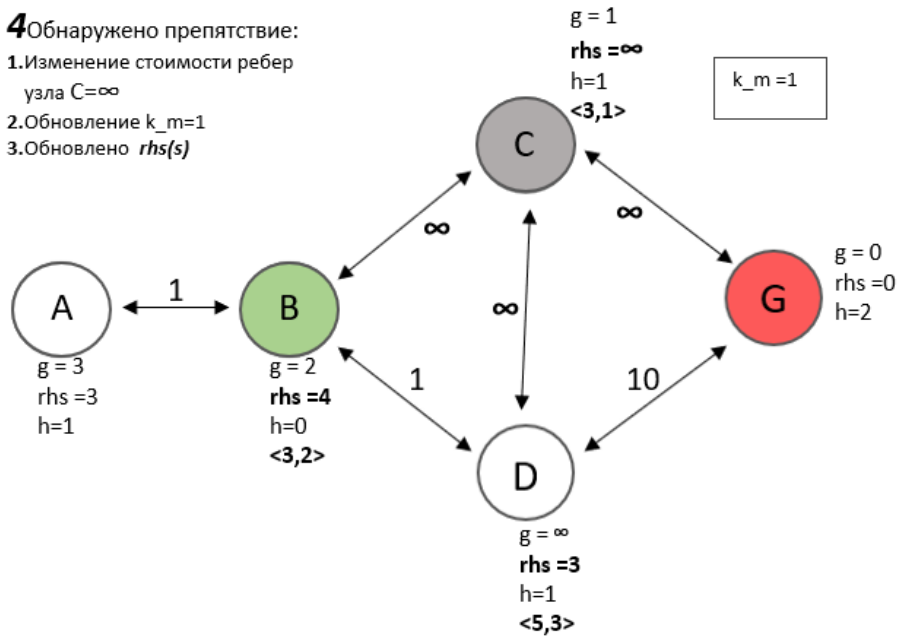


Далее мы собираемся обновить значения rhs вершины B и его соседей.

$OpenList = [\langle 3, 1 \rangle; \langle 3, 2 \rangle \langle 5, 3 \rangle]$

4 Обнаружено препятствие:

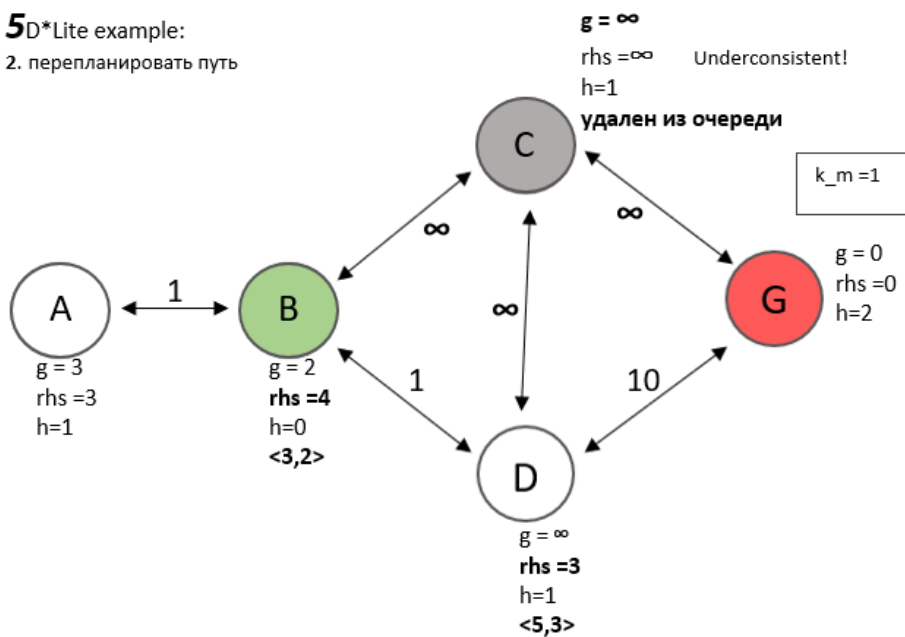
1. Изменение стоимости ребер узла $C = \infty$
2. Обновление $k_m = 1$
3. Обновлено $rhs(s)$



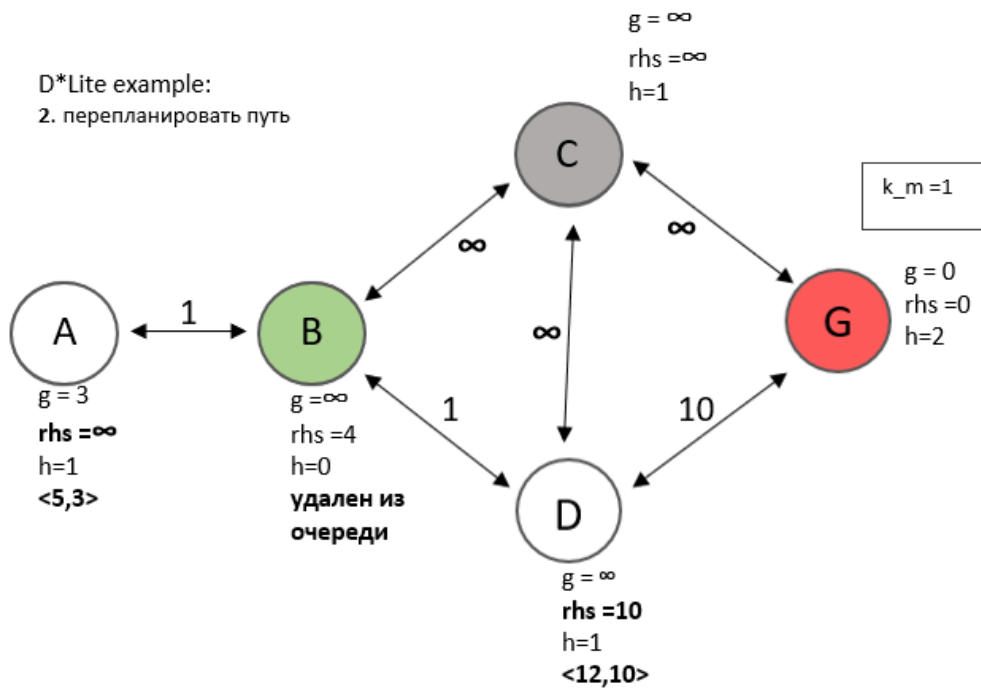
Далее мы удаляем из очереди вершину C. Проверка $computeShortestPath(g(s) < rhs(s))$, следовательно $g(s) = \infty$

5 D*Lite example:

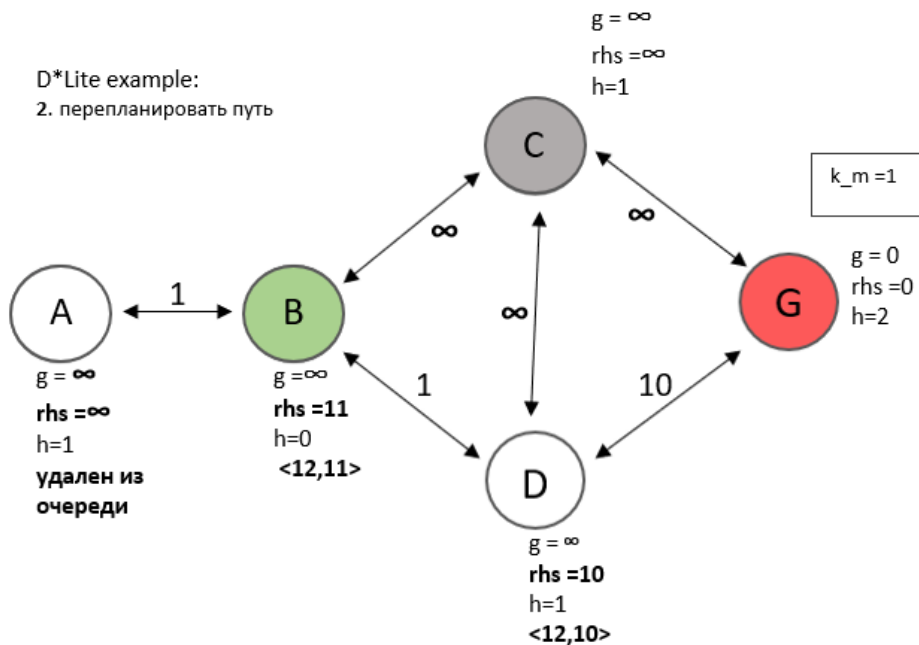
2. перепланировать путь



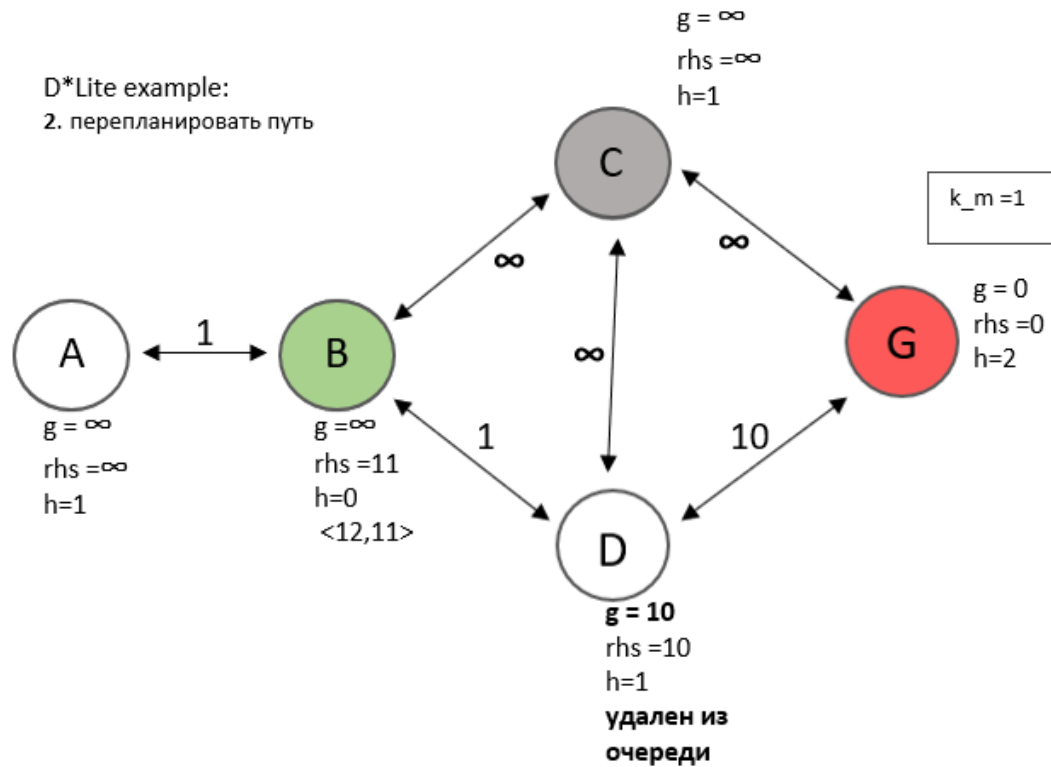
Далее мы удаляем из очереди вершину B так как это наименьшее значение в очереди. Проверка $computeShortestPath(g(s) < rhs(s)) 2 < 4$, следовательно $g(s) = \infty$. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения. OpenList=[<5,3>;<12,10>]



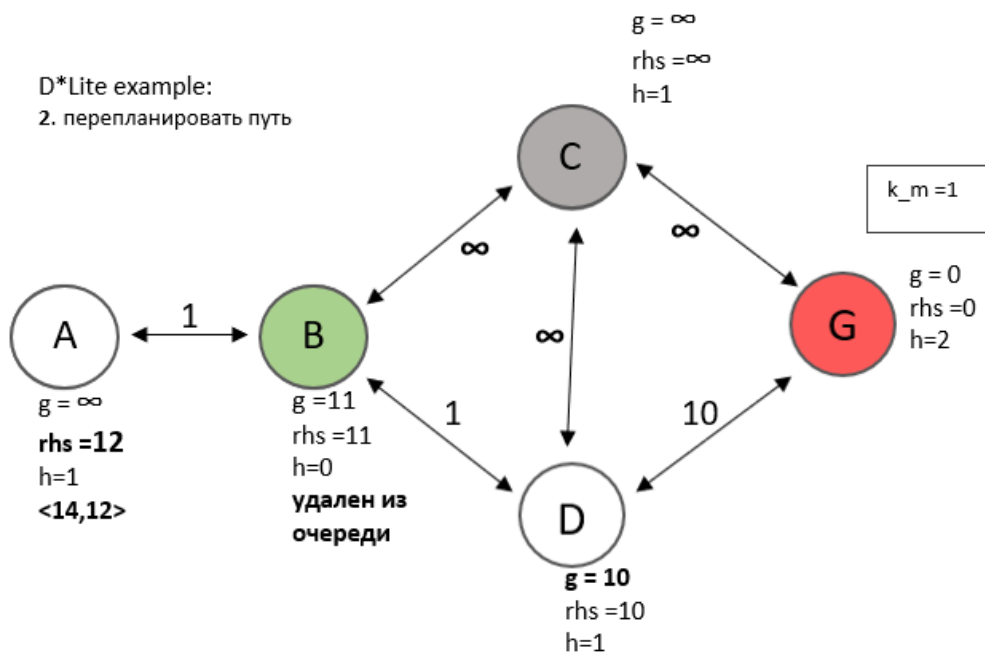
Так как $\langle 5, 3 \rangle$ меньше чем $\langle 12, 10 \rangle$ удаляем из очереди вершину A.
 Проверка $computeShortestPath(g(s) < rhs(s)) 3 < \infty$, следовательно $g(s) = \infty$.
 Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
 OpenList= $[\langle 12, 10 \rangle; \langle 12, 11 \rangle]$



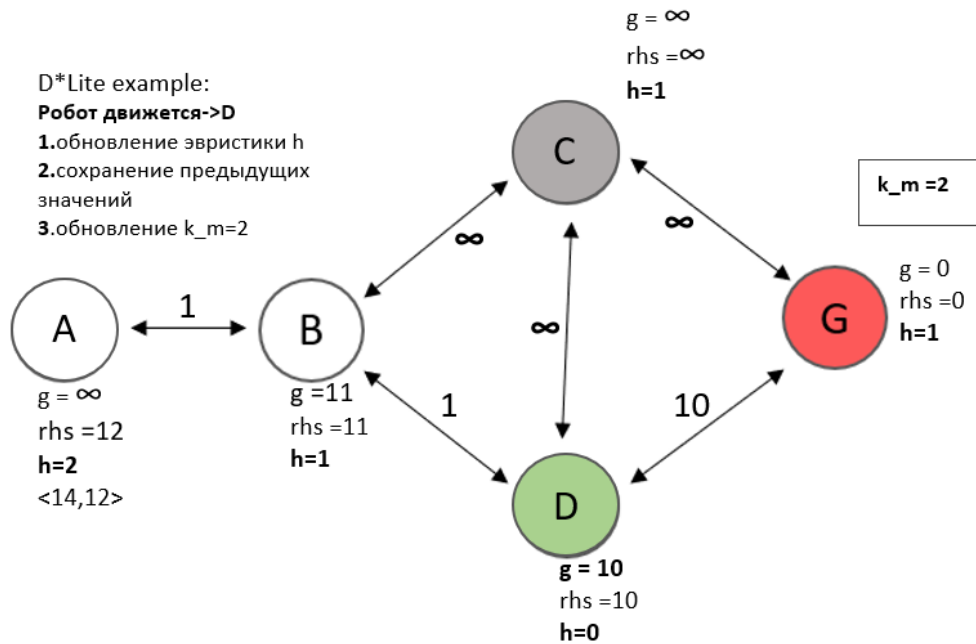
Так как $\langle 12, 10 \rangle$ меньше чем $\langle 12, 11 \rangle$ удаляем из очереди вершину D.
 Проверка $computeShortestPath(g(s) > rhs(s)) \infty > 10$, следовательно $g(s) = rhs(s) = 10$.
 Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
 OpenList= $[\langle 12, 11 \rangle]$



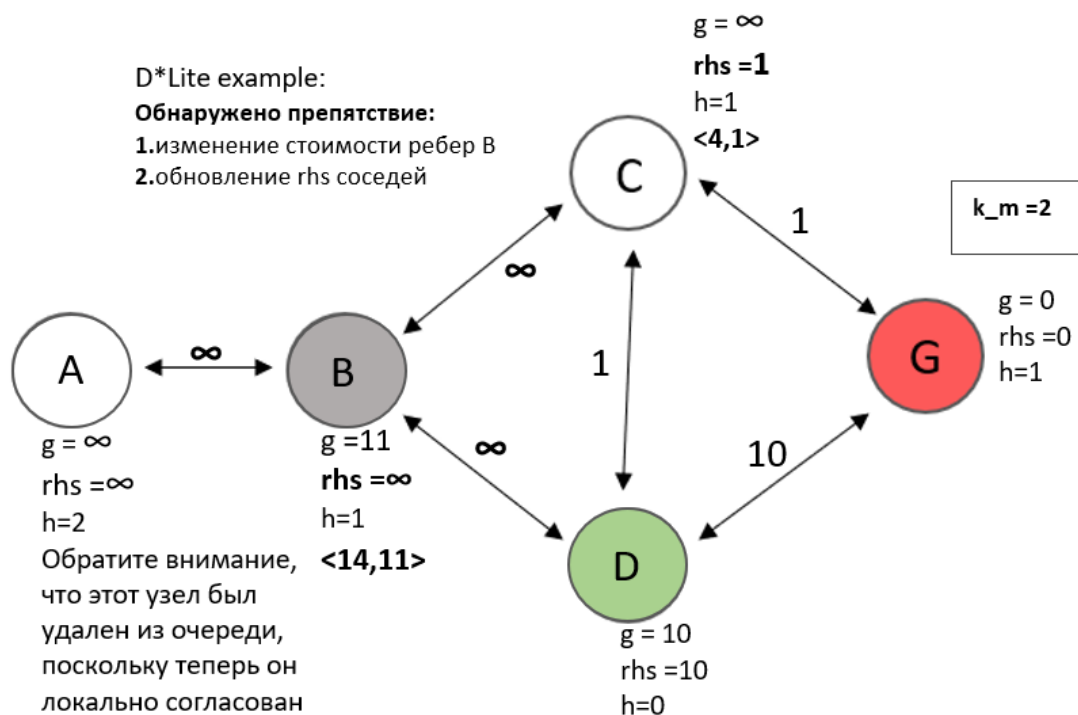
Удаляем из очереди вершину B. Проверка $computeShortestPath(g(s) > rhs(s)) \infty > 11$, следовательно $g(s) = rhs(s) = 11$. Теперь узлы в графе локально согласованы. Обновляем значение rhs соседей и добавляем в $OpenList$ изменившиеся значения. $OpenList = [\langle 14, 12 \rangle]$



Мы нашли лучший и единственный путь из B \rightarrow D \rightarrow G следовательно робот двигаться в D. Изменяется эвристика всех значений вершин на графе. Обновление значения $k_m = 2$



Оказывается препятствие тоже движется. Происходит изменение стоимости ребер вершины B, теперь их стоимость равна бесконечности. Также обновляются значения rhs соседей вершины D. В этом случае значение rhs(G) никогда не меняется потому что расстояние G до самой себя всегда будет равно нулю. $OpenList = [\langle 4, 1 \rangle, \langle 14, 1 \rangle]$

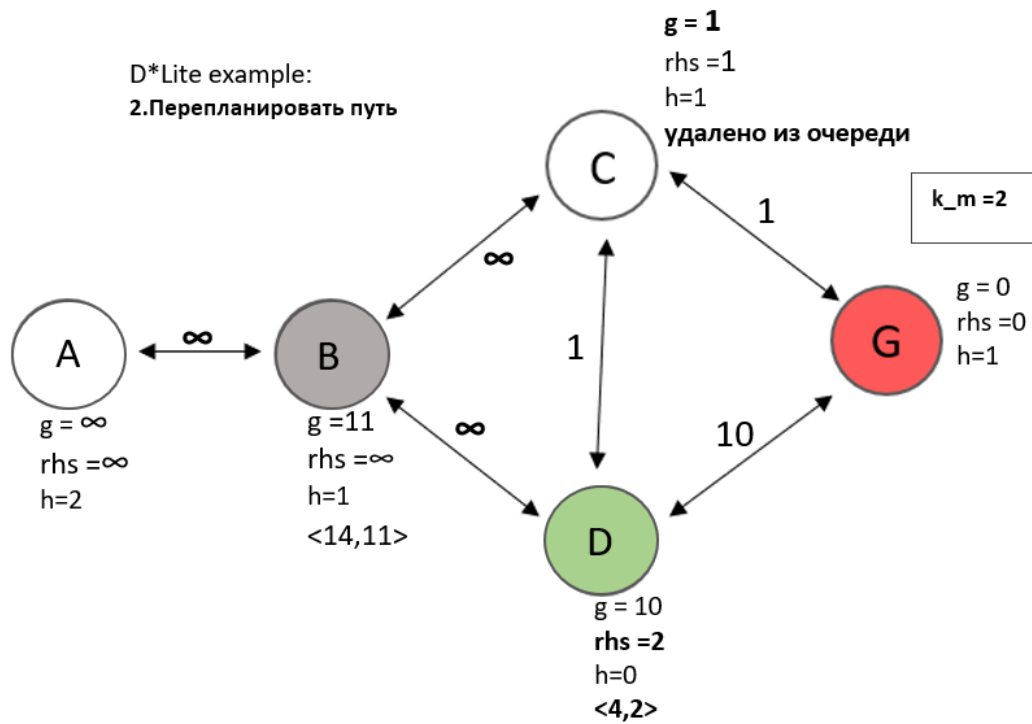


Следовательно мы удаляем из очереди вершину C.

Проверка $computeShortestPath(g(s) > rhs(s)) \infty > 1$, следовательно $g(s) = rhs(s) = 1$.

Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.

$OpenList = [\langle 4, 2 \rangle, \langle 14, 11 \rangle]$

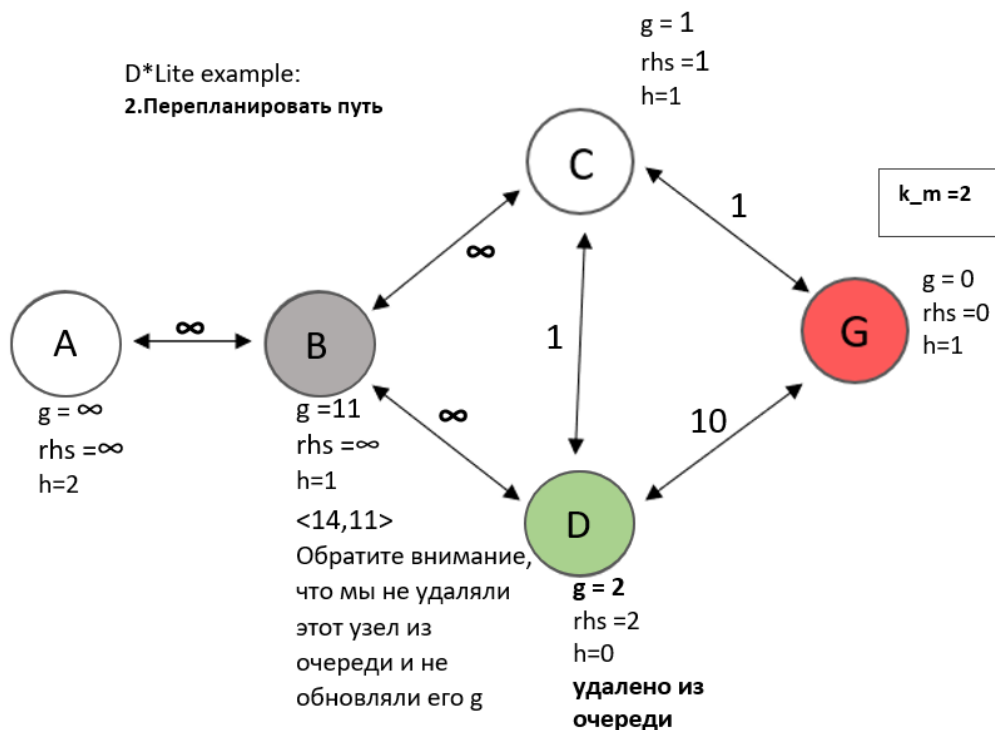


Потом удаляем из очереди вершину D.

Проверка $computeShortestPath(g(s) > rhs(s)) 10 > 2$, следовательно $g(s) = rhs(s) = 2$.

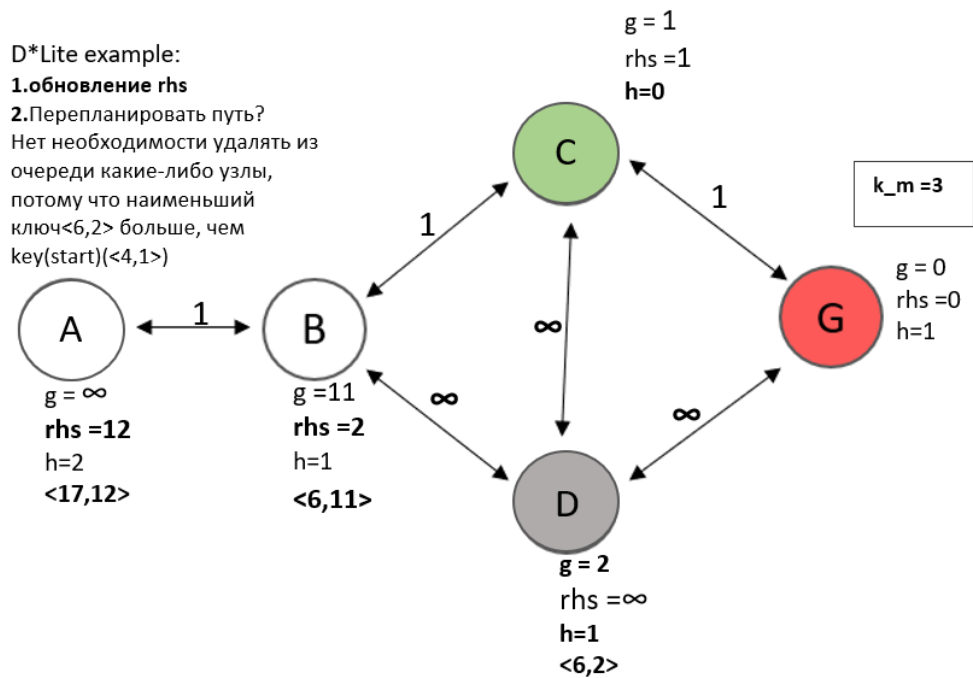
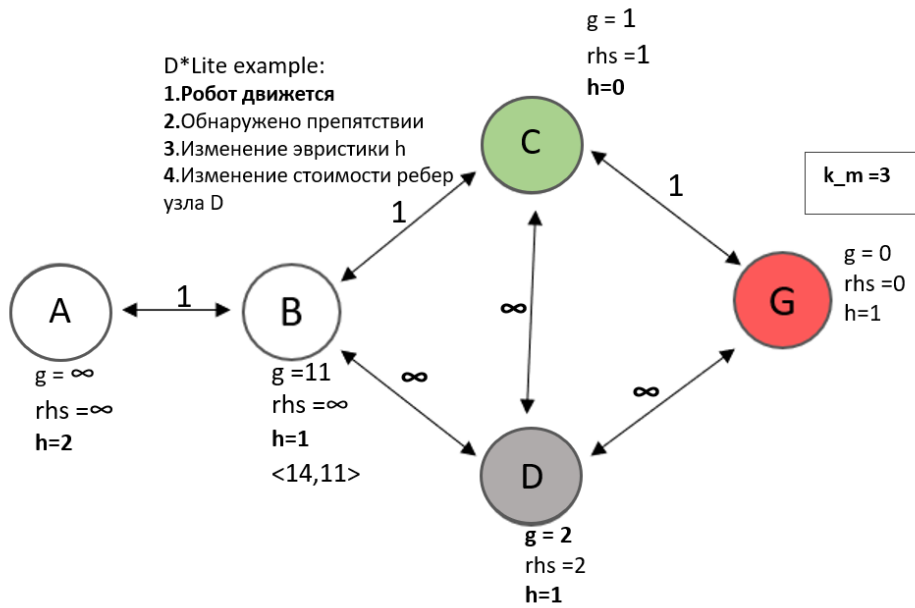
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.

OpenList= $\langle 14, 11 \rangle$

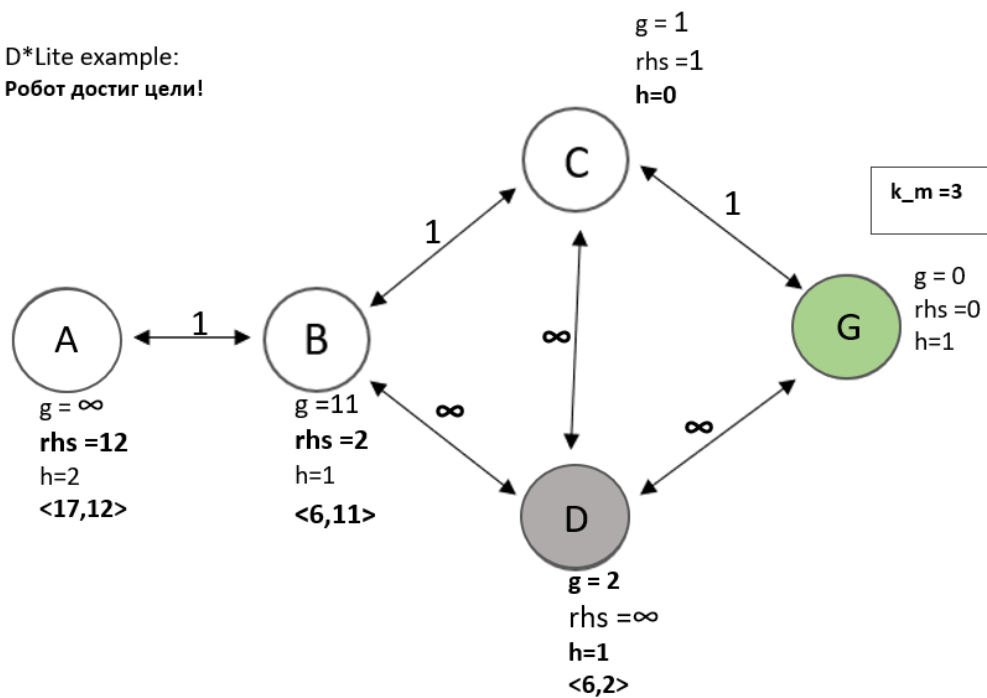


Мы обнаружили путь из $D \rightarrow C \rightarrow G$ и робот перемещается в вершину C. Также обновляется эвристика для всех вершин графа.

Препятствие не стоит на месте и приследует робота. Следовательно ребра вершин D становятся стоять бесконечность. Также обновляется значение $k_m = 3$ Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.



D*Lite example:
Робот достиг цели!



Тестирование и анализ производительности

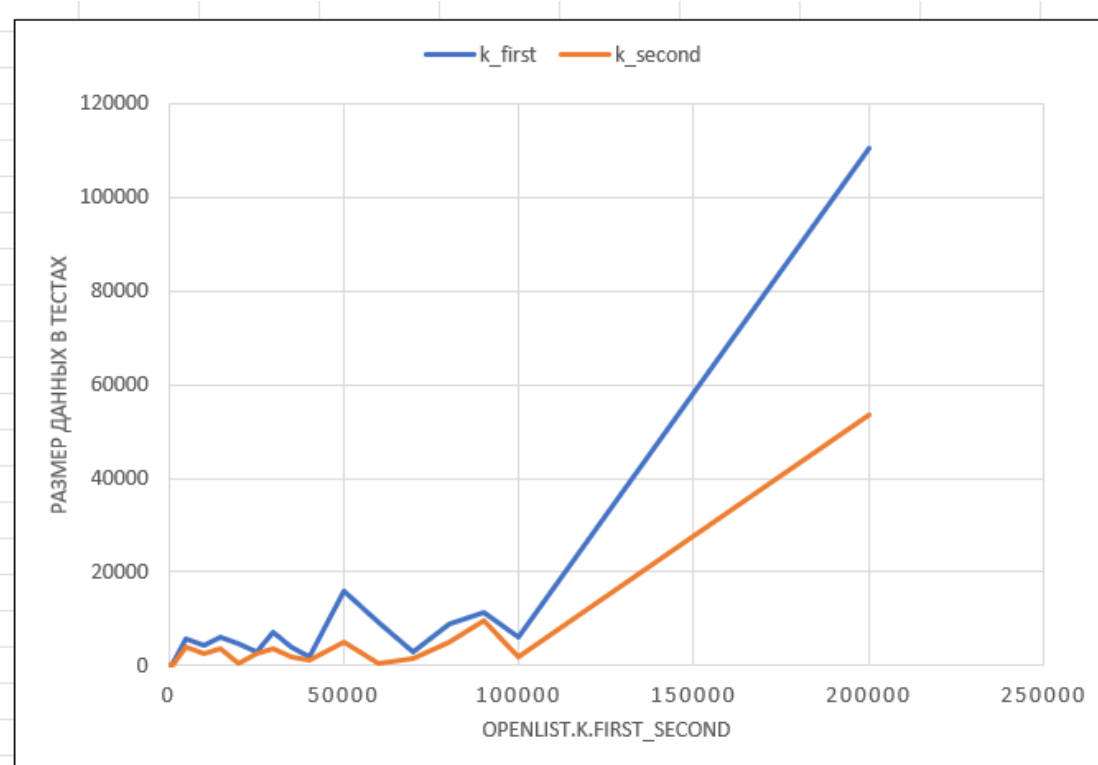
Тесты написаны в формате *in.txt out.txt*. Всего 20 ручных тестов, позволяющих измерить время от количества размера данных.



Также во все *out.txt* я измеряю значение $rhs(s)$ и $g(s)$. На таблице представлено то как зависят функции от размера входящих данных.



На данном графике произведен анализ состава *OpenList*(приоритетной очереди). А именно я сравниваю первое значение k_{first} и k_{second} в зависимости от размера данных.



Заключение

С помощью введения ключевого модификатора k_m и отложенного обновления вершин получилось убрать из итерации алгоритма $O(n \cdot \log(n))$ операций, которые тратились на обновление очереди. На основе теории, приведенной ранее, алгоритм использует не более $O(n \cdot \log(n))$ операций.

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] *Incremental Path Planning*. URL: <https://ocw.mit.edu/courses/16-412j-cognitive-robotics-spring-2016/resources/advanced-lecture-1/>.
- [5] *[Planning] D* Lite path search algorithm*. URL: <https://www.programmersought.com/article/1946589861/>.
- [6] *Research and Optimization of D-Start Lite Algorithm in Track Planning*. URL: <https://ieeexplore.ieee.org/abstract/document/9184858>.
- [7] *D-STAR*. URL: <https://ieeexplore.ieee.org/abstract/document/5501636>.
- [8] *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*.
- [9] Кениг С. и Лихачев М. (2002, июль). *Чертов Лайт*. В АААИ/IAAI (стр. 476-483).
- [10] Кениг С., Лихачев М. и Фурси Д. (2004). *Планирование на всю жизнь A**. Искусственный Разведка, 155(1), 93-146.
- [11] Стенц, А. (1994, май). *Оптимальное и эффективное планирование маршрута для частично известных сред*. В материалах Международной конференции IEEE по Робототехника и автоматизация.
- [12] Лихачев М., Фергюсон Д. И., Гордон Г. Дж., Стенц А. и Траун С. (2005, июнь). *Anytime Dyna⁺ A**: Алгоритм перепланировки в любое время.
- [13] Кениг, С.; Тови, С.; и Халлибертон, У. 2001. *Жадный просмотр карты местности*. В материалах Международной конференции по робототехнике и автоматизации, 3594-3599.
- [14] *Динамический A* Lite: Доказательства. Технический отчет*, Колледж вычислительной техники Технологического института Джорджии, Атланта (Джорджия).
- [15] *Международного Конференция по интеллектуальным автономным системам*. Мур, А. и Аткесон, С. 1995. *Алгоритм парти-игры для обучения с подкреплением с переменным разрешением в многомерных пространствах состояний*.
- [16] Stentz, A. 1994. *Optimal and efficient path planning for partially-known environments*. In *Proceedings of the International Conference on Robotics and Automation*, 3310-3317.
- [17] *Real-time Path Planning for Virtual Agents in Dynamic Environments 2007 IEEE Virtual Reality Conference*. URL: <https://www.computer.org/csdl/proceedings-article/vr/2007/04161010/120mNASraPs>.

- [18] *A Two-level Path Planning Method for On-road Autonomous Driving* 2012 Second International Conference on Intelligent System Design and Engineering Application. URL: <https://www.computer.org/csdl/proceedings-article/isdea/2012/4608a661/120mNvq5jGv>.
- [19] S. Koenig. *A Comparison of Fast Search Methods for Real-Time Situated Agents*. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 864-871, 2004. URL: <http://idm-lab.org/project-a.html>.