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 .

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

Постановка задачи

$DstarLite$ - это алгоритм выполняющий поиск от одного начального узла к другому целевому(конечному,следующему узлу).По мере того как мы видим что на пути происходят изменения,мы будем это корректировать и учитывать.$DstarLite$-рассматривают с двух точек зрения.
Первая точка зрения должна думать как о модификации основного пространства.В своей первой итерации или при первом прогоне до каких-либо модификаций он ведет себя похоже на алгорит $Astar$ а именно мы производим поиск от начального узла к узлу цели, мы сортируем узлы в очереди по общей стоимости, которая является суммой,значения g и h(эвристики)$f(s) = g(s) + h(s, s_{goal})$

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

Вторая точка зрения на алгорит $Astar$ с точки зрения алгоритма Дейкстры,когда мы исправляем или изменяем граф,наши методы заключаются в том, что бы ослабить веса ребер пока они не достигнут своего истинного значения.Теперь, когда мы отличаемся от алгоритма Дейкстры используя $DstarLite$ мы выбираем только те узлы,которые хотим рационально(эффективно) использовать.В то время как алгоритма Дейкстры, пытается уместить(вычислить) весь граф.Мы хотим пройти только от одного начального узла к одному конечному узлу.
Теперь что обычно не рассматривается в алгоритме $Astar$, но очень важно для $DstarLite$ так это то как различить два узла с одниковой общей стоимостью в очереде.
Итак мы вводим пару(introducing a couplet),состоящую из 2 значений $f(s) =< f_1(s), f_2(s) >$

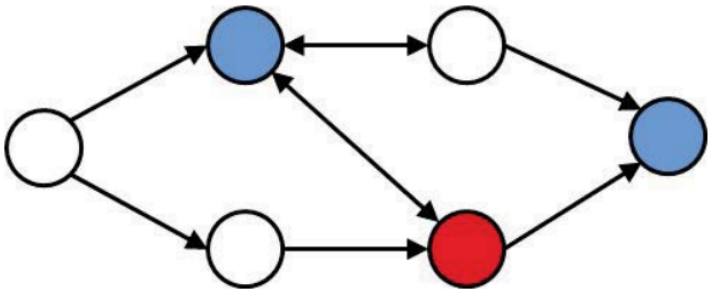
- $f_1(s) = g(s)+h(s, s_{goal})$Это линейная комбинация наших затрат на достижение конечного узла и эвристической функции.
- $f_2(s) = g(s)$ - стоимость перехода от одно узла к другому
- $f(s) =< g(s) + h(s, s_{goal}), g(s) >$

Таким образом порядок расширения узлов из очереди сначала упорядочивает их по значению $f1$ и если два узла имеют одинаковое значение $f1$, мы выбираем узел с наименьшим значение $g(s)$

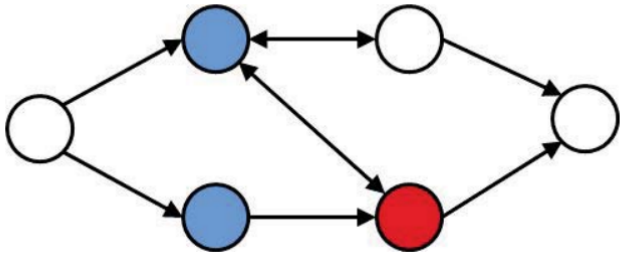
Концепция предшественников и приемников

Концепция предшественников(predecessor) и приемников(successors) любого заданного узла.В любом случае граф состоит из направленных ребер и у нас может быть граф с ненаправленными ребрами,но мы можем думать о любом ребре направленном в обе стороны.
Концпепция **приемника** узла состоит в том что каждый узел может быть достигнут из данного узла по ребрам которые можно найти.
Концепция **предшественников** представлет собой каждый узел, из которого этот узел может быть достигнут, т.е. узлы от которых мы можем следовать по направленому ребру к узлу.

- $Succ(s)$ -множество узлов(вершин) изсходящих из вершины s .Таким образом приемниками($Succ(s)$) красного узла, являются вот эти два синих узла.

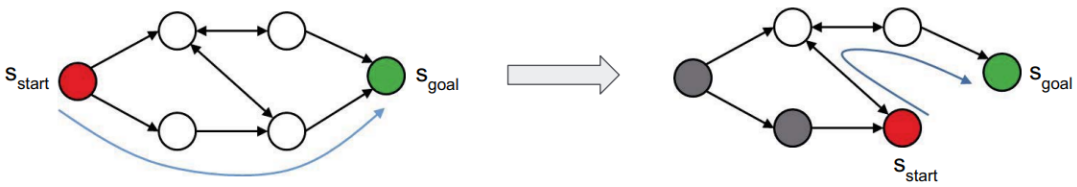


- $Pred(s)$ - множество узло вершин входящих в вершину s.
  Таким образом предшественниками($Pred(s)$) красного узла, являются вот эти два синих узла.



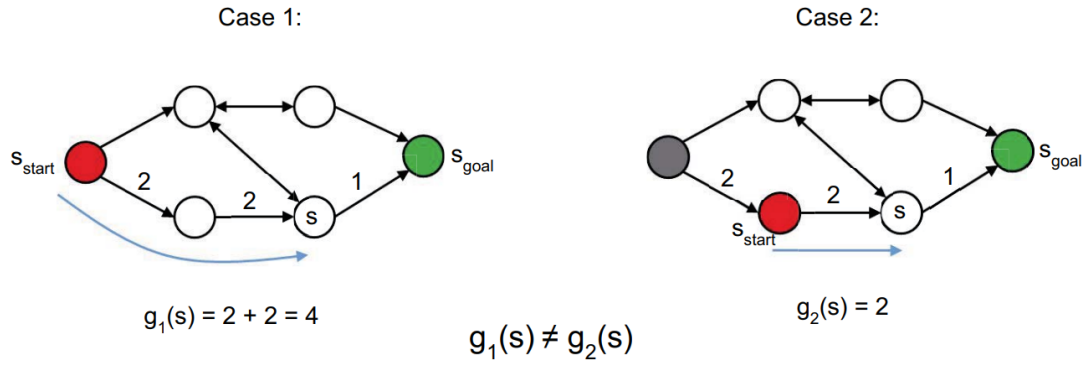
Переформулировка: Сведите к минимуму повторные вычисления

Подумайте над $DstarLite$ в том смысле, что это повторяющийся поиск по принципу ”лучше всего сначала”(best-first search).Когда принципы алгоритма $Astar$ проявляються через граф с изменяющимися весами ребер по мере прохождения этого графа, что означает что по мере того как мы перемещаемся от нашего начального узла к нашему конечному узлу вдоль пути,который мы планируем мы можем видеть что веса ребер меняются.



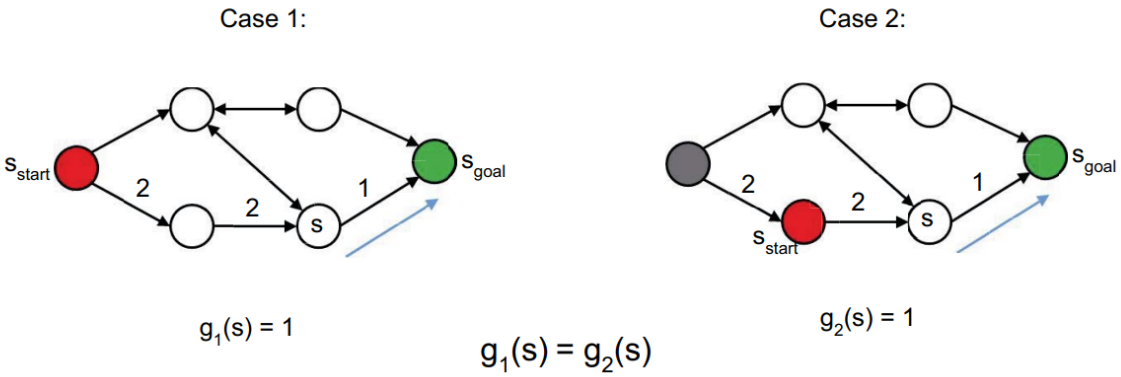
И в этом примере я моделирую препятсвие возникающее между этим узлом и целью, что означает что мы удалили ребро.И это фактически то же самое что и изменение веса ребра.Моделируем это изменение веса до значения бесконечности, что означает что по этому ребру нельзя двигаться.
Мы рассматриваем это как перепланирование за счет уменьшение стоимости пути.Как

только мы переместились s_{start} как мы найдем путь отсюда к s_{goal},когда это ребро по сути удалено?



Поэтому мы хотим сделать $DstarLite$ эффективным.И это повторный поиск по графу по мере его обхода. Поскольку мы используем эти значения $g(s)$ и если мы сохраним формулировку того, что мы измеряем растояние, которое нужно пройти или достичь узла от нашего начального узла.Эта формулировка не сохраняеться, когда мы перемещаем наш s_{start}. Перемещение начально узла происходит при обновлении позии робота.

Перейдем к примеру.Значение $g_1(s)$ или стоимость перехода от одного начального узла к другому представляет комбинацию 2 ребер на примеер Case1:таким образом $g_1(s) = 2 + 2 = 4$
Теперь когда мы движемся по тому пути,который мы спланировали к следующему узлу прошлое значение $g_1(s) \neq g_2(s)$ не сохранилось на примере Case2.



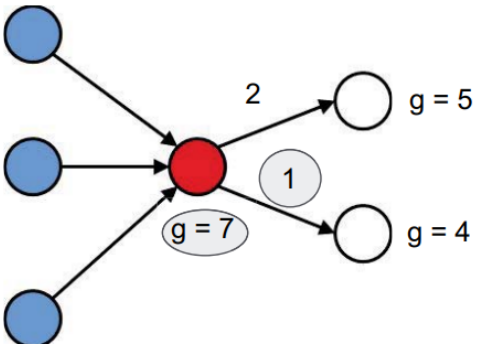
Так что мы делаем это переформулируем наш поиск,чтобы начать **искать в противоположном направлении**.Теперь мы измеряем для наших значений g стоимость достижения цели от конкретного узла. И вместо эвристики измеряем затраты, чтобы добраться от конкретного узла или добраться от начального к конкретному.И так ищем от цели s_{goal} назад по графу.
В этом случае независимо от того,являеться ли узел начальным,мы сохраняем значение $g(s)$ и стоимость путешствия от текущего узла к цели.Мы сохраним эту формулировку для максимальной эффекивности.

Общий D* Lite:

1. Инициализируйте все узлы как нерасширенные.
2. Выполнить поиск по принципу ”best-first search”(лучший — первый)от конечного узла к начальному, пока начальный узел не согласуеться со своими соседями
3. Затем мы перемещаем начальный узел в следующую лучшую вершину(по созданному ранее нами плану)
4. Если какие-либо граничные затраты изменились(имееть ввиду стоимость ребер):
   (a) отследить как изменилась эвристика
   (b) обновить исходные узлы измененных ребер
5. Повторить пункт №2

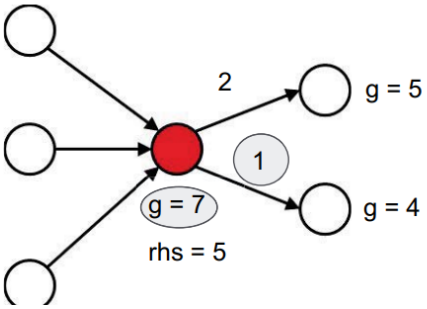
Обработка изменения веса локально

Как мы справляемся с изменением веса локально на месте.И делаем это потому что не хотим отслеживать изменение веса на протяжении всего графика.Мы хотим справиться с эти эффективно,поэтому нам нужно обрабатывать только те изменения,которые действительно влияют на нашу проблему.
Пример заключаеться в том,что я изменила стоимость веса ребра с 6 на 1.И теперь мы можем добраться до красного узал по новому оптимальному пути.Новое значение $g(s) = 5$.Но это не сохранилось.



Затем эти изменения распространяються на наших предшественников $Pred(s)$, потому что стоимость каждого процесса зависит от значения $g(s)$ для их приемника.И поэтому когда мы попадаем в узел, мы должны распространить это изменение по всему граффу.Поэтому,что

бы сделать это эффективно мы используем подход подобный $Astar$,где мы сначала обновляем узлы с наименьшей стоимостью и мы не расширяем узел,пока он не станет следующим с наименьшей стоимостью.



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) И когда ваше значение $rhs(s) \neq g(s)$ это называеться,то что имеем как **локальная несогласованность.** Учитывая логику значение $rhs(s)$ должно быть таким,каким будет скоректированно ваше значение $g(s)$.Таким образом при изменении стоимости ребра $rhs(s) = 4 + 1 = 5$

Локальные несоответствия

У нас есть 2 типа локального несоотвествия

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

Обновление и расширение узлов

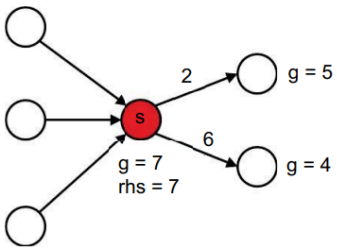


рис.1

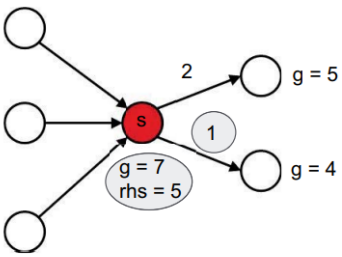


рис.2

Под ”обновлением узлов” я подразумеваю повторное вычисление значения $rhs(s)$ а затем помещение этого узла в очередь приоритетов,если этот узел локально несовместим.Что бы привести пример я изменила значение стоимость 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

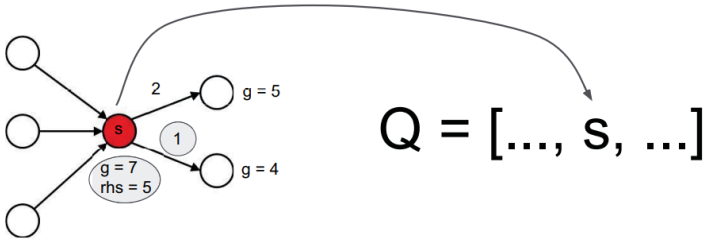


рис.3

Значение приоритетной очереди основанно на этой формуле
$Q =< min[g(s), rhs(s)] + h(s, s_{start}), min[g(s), rhs(s)] >$
Оно основано на том как мы види значение $g(s)$ $rhs(s)$.Мы хотим взять минимальное значение одногого из них,что бы мы могли обрабатывать его в первый раз,когда оно вызовет изменения,распространяющиеся по графу.Таким образом это наш порядок в приоритетной очереди.

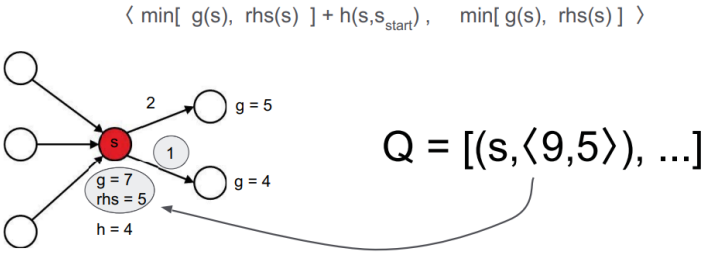


рис.4

Итак мы расширяем наши предыдущие узлы удаляя из очереди приоритетов и изменяя значение $g(s)$.Значения $g(s)$ уменьшаються до значения $rhs(s)$,поэтому мы просто устанавливаем $g(s) = rhs(s)$.В этом случае узел локально непротеричив(согласован) и он остается таким.

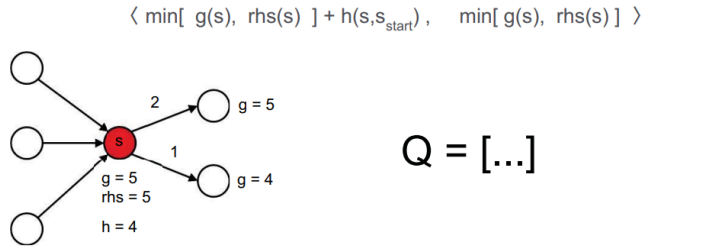


рис.5

Почему происходит процесс изменения $rhs(s)$ и отслеживаю это новое значение для того,что бы поставить в очередь,а затем снова убрать его,вместо того чтобы просто пересчитать значение $g(s)$? Можно подумать что вы добавляете что-то в очередь,если мы незамедлительно обновим $g(s)$,существует несколько узлов,которые могут изменить один и тот же узел,который мы только-что рассмотрели.Под этим я подразумеваю дополнительные изменения работы по мере их распространения обратно по графу,которые могут привести к изменению значения правой стороны это конкретного узла.Что нам нужно сделать если мы можем пересчитывать g(s) каждый раз так это поставить его в очередь,сигнализируя о том что мы пересчитываем

g(s) и выполнить этот перерасчет, а затем новые изменения которые мы сделали делая это снова и снова и снова и мы хотим избежать этой ситуации.
Мы хотим сохранить $rhs(s)$ как временное значение.Которое можно изменить,когда оно находиться в очереде.Итак,когда что-то стоит в очереди, оно есть там один раз и его можно обновить и снять.Но мы знаем когда его сняли,что он был снят в нужно время и его не нужно будет снова обрабатывать.

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 распространитесь на все предшественники
- Установить $g(s) = \infty$
- Обновление всех предшественников узла s и сам s

Вершина теперь локально согласована или локально сверхсогласованна. Она будет оставаться в очереде 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(
                 s_start))) ||
10            (getRHS(s_start) != getG(s_start))) {
11
12         if (k++ > maxSteps) {
13             return -1;
14         }
15         state u;
16         bool test = (getRHS(s_start) != getG(s_start));
```

```
17
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.

Перенос приоритетной очереди и изменение эвристики

Проблема в том что мы движемся s_{start} к другому узлу. Наше эвристическое значение измеряеться от любого заданного узла до узла конечного.Поэтому когда наш начальный узел изменился эвристика не будет,такого же значения ,что и прежде как на новом узле.Итак мы хотим, то что уже находиться в очереди было сопастовимо с тем что мы вычисляется для нашего нового значения,что бы мы могли использовать нашу логику которые мы уже сделали.
Мы вводим что-то называемое ключевым модификатором.
$k_m = k_m + h(s_{last}, s_{start})$
Мы переходим от предыдущего начального узла,который здесь я указала как последний к новому начальному узлу.Тепер когда мы добавляем новые узлы в очередь,вместо того,чтобы вычитать это значение из всего что уже находиться в очереди,мы просто увеличиваем для всех узлов,которые мы помещаем в очередь,их значения на этот новый модификатор.Потому что когда мы выходим из очереди,все что имеет значение

-это относительное различие между ними.Таким образом,вместо того,что бы вычитать значение из всего в очереди,добавление этого значения ко всем новым значениям,которые появляются в очереди,позволит достичь той же цели.
Формула приоритетной очереди изменяется:
$< min[g(s), rhs(s)] + h(s, s_{start}) + k_m, min[g(s), rhs(s)] >$

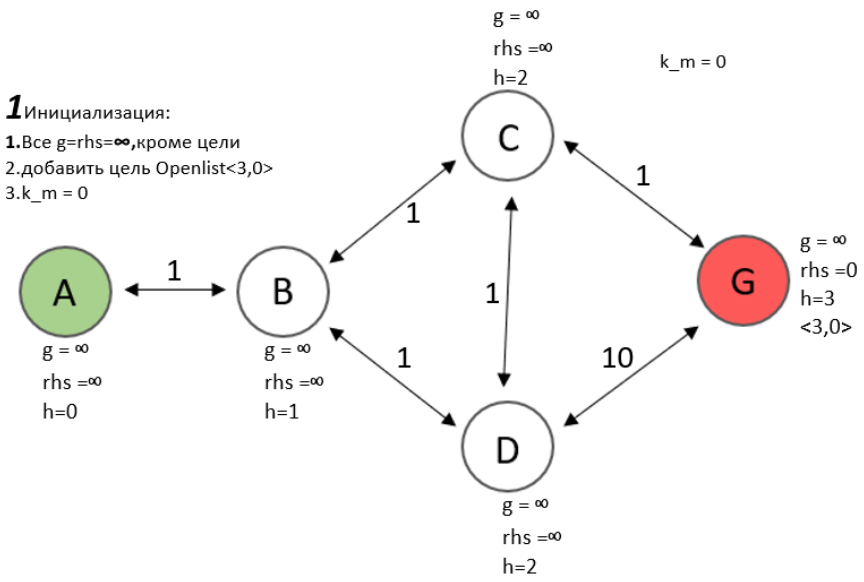
Общий D* Lite(конкретизация):

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),покаs_{start} не будет локально согласован и расширен
3. Переместить $s_{start} = argmin_{s' \in Succ(s^{start})}(c(s_{start}, s') + g(s'))$
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 узлов измененных ребер
5. Повторите шаг № 2

Всегда сортируйте по формуле:
$< min[g(s), rhs(s)] + h(s, s_{start}) + k_m, min[g(s), rhs(s)] >$

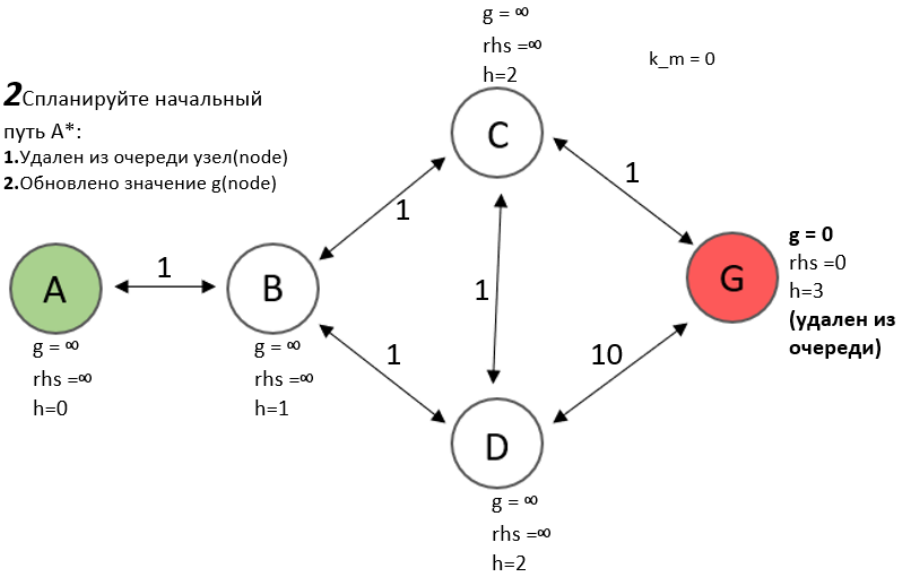
Пример работы

Рассмотрим краткий пример с пятью узлами.Робот начинает с узла A и его цель найти кратчайший путь к узлу G,за исключением того что граф может измениться по мере прохождением робота или робот может получить новую информацию.
Изначально робот думает что все узлы доступны.Он знает что все длины пути равны 1 за исключением пути от D до G=10.Это двунаправленный граф,каждое ребро идет в обоих направлениях.
Эвристика - это количество узлов до начального расстояния. Все узлы правильно локально согласованны так как g(s)=rhs(s)

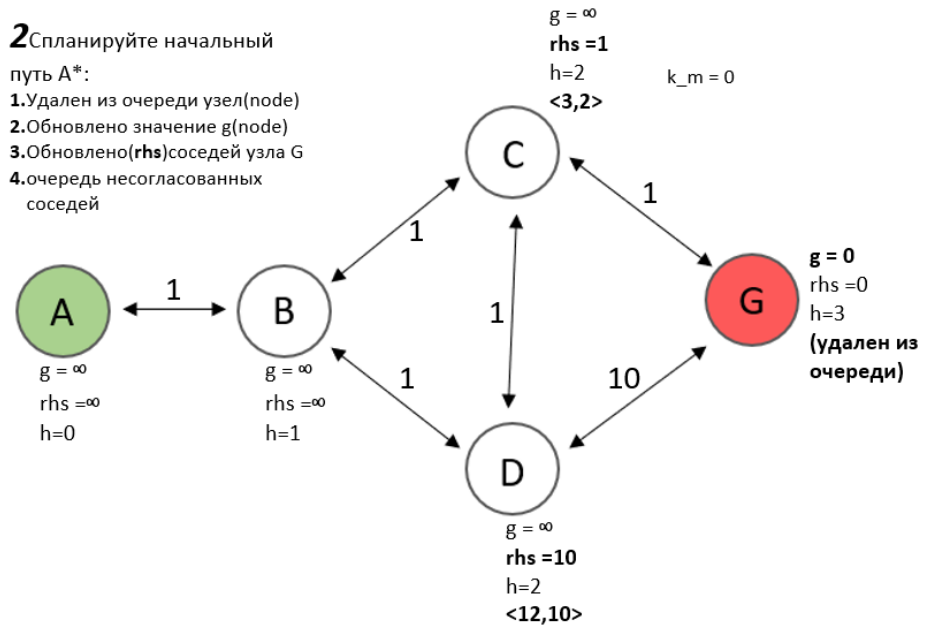


Мы помещаем вершину G в Openlist(очередь) используя формулу $< min[g(s), rhs(s)] + h(s, s_{start}) + k_m, min[g(s), rhs(s)] >$ Теперь у нас есть один узел(вершина в очереди).Так что нам придеться выйти из очереди.

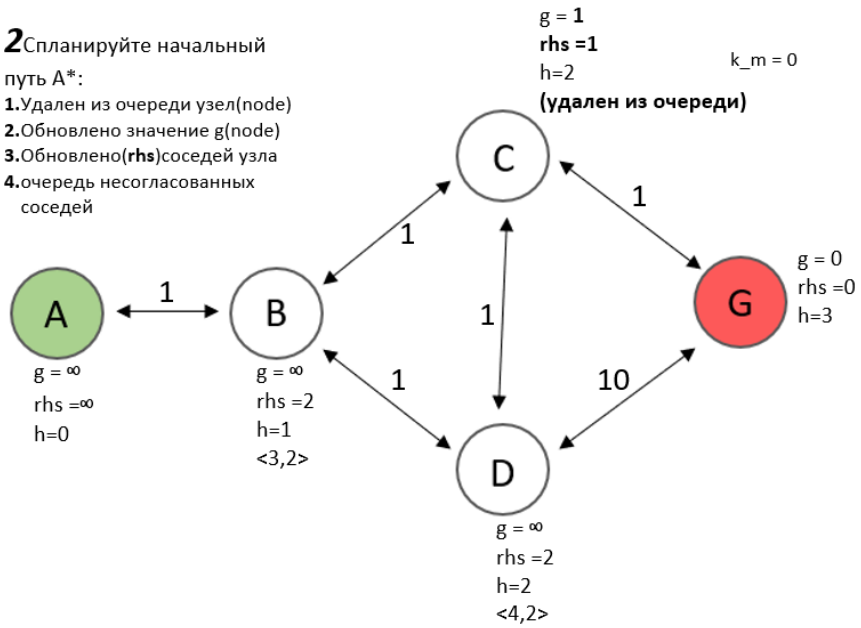
- удаляем узел G из OpenList
- проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=0



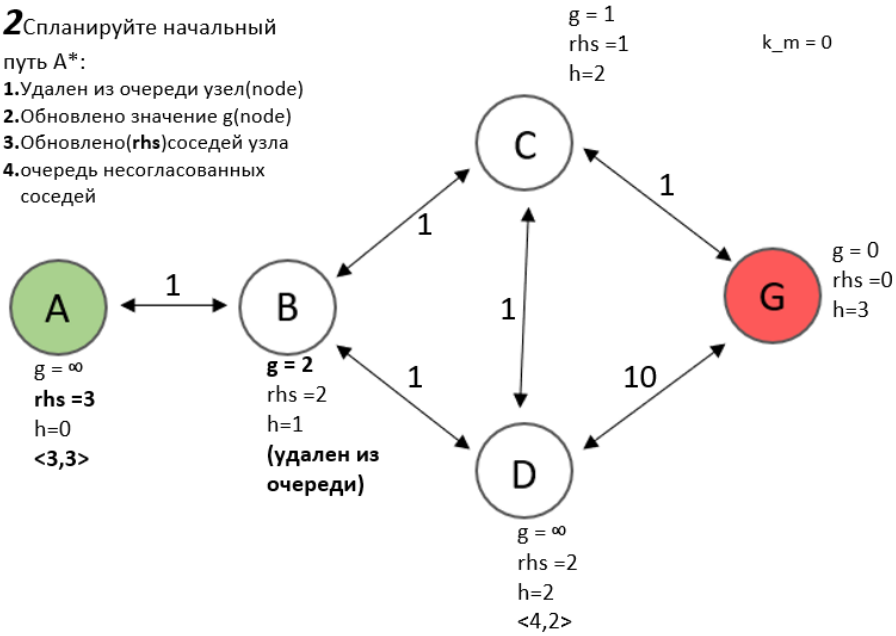
Теперь мы собираемся обновить значения $rhs(s)$ соседей узла G и в каждом случае значение будет минимальным по формуле $rhs(s) = min_{s' \in Succ(s)}(g(s') + (s, s')$.И добавить в очередь $OpenList = [< 3, 2 >; < 12, 10 >]$



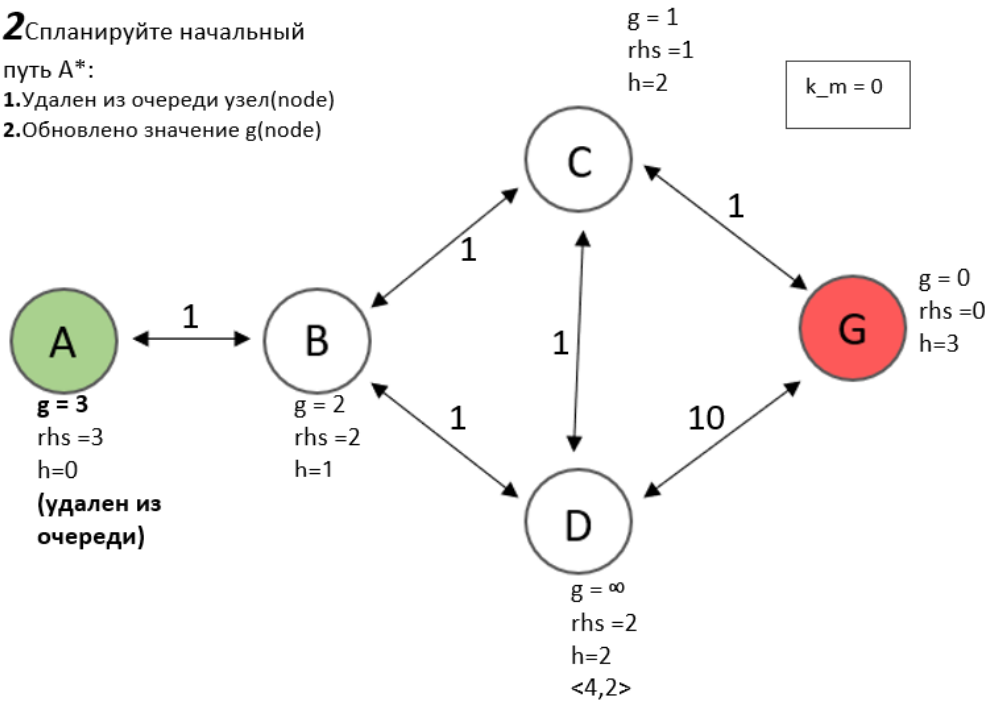
Теперь мы исключим из очереди еще один узел C так как у него наименьший ключ <3,1> .Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=1 И обновим значения rhs(s) соседей узла C.
Теперь в очереди $OpenList = [< 3, 2 >; < 4, 2 >]$



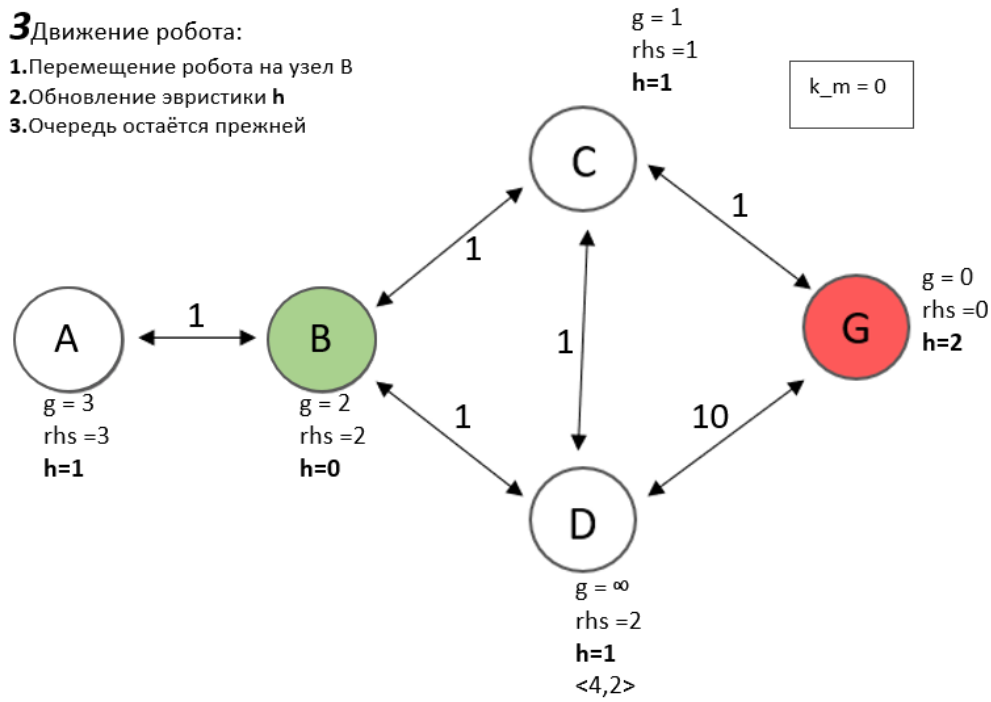
Дальше мы удаляем из очереди узел B так как у него наименьший ключ <3,2>Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=2 И обновим значения rhs(s) соседей узла B.Мы не изменяем ключево значение узла D так как стоимость из G->C->D будет меньше чем из G->C->B->D
Теперь в очереди $OpenList = [< 3, 3 >; < 4, 2 >]$

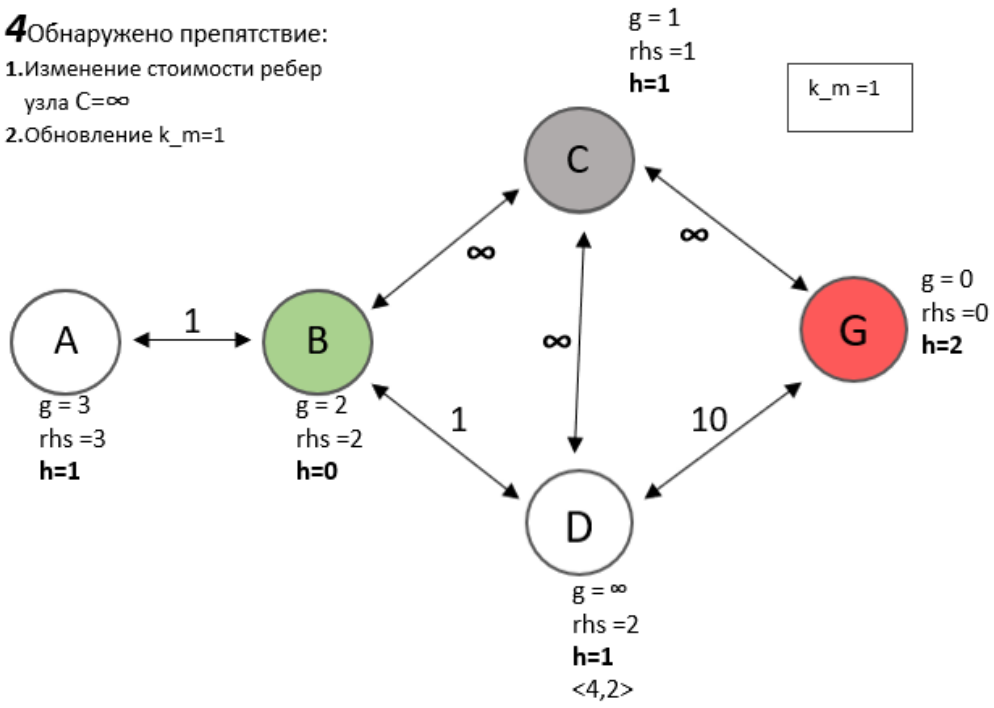


Далее мы удаляем из очереди узел A так как у него наименьший ключ.Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=3. $OpenList = [< 4, 2 >$] И на это мы закончили.

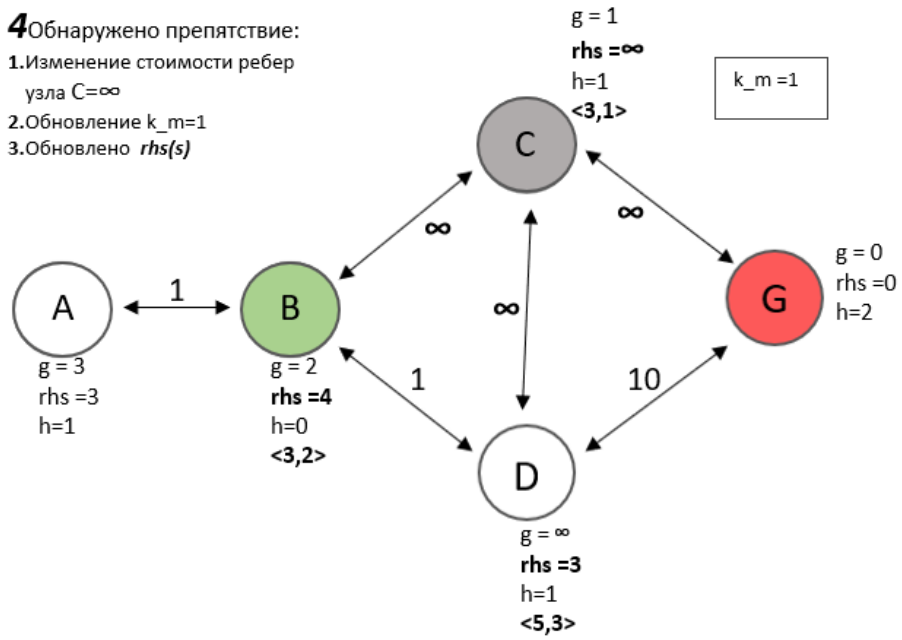


Робот переместился к своему следующему лучшему месту узлу B.Мы обновляем значения эвристики h для каждого узла в графе. Мы не изменяем очередь, но также должны обновить $k_m = 1$ Появляется препятсвие.В точке C есть препятсвие, которе робот теперь может видеть так как он ближе переместился к узлу C, но раньше он его не видел. Мы укажем эти изменения изменив ребра узла C на стоимость равной бесконечность.Это означает что робот никогда не сможет туда добраться.

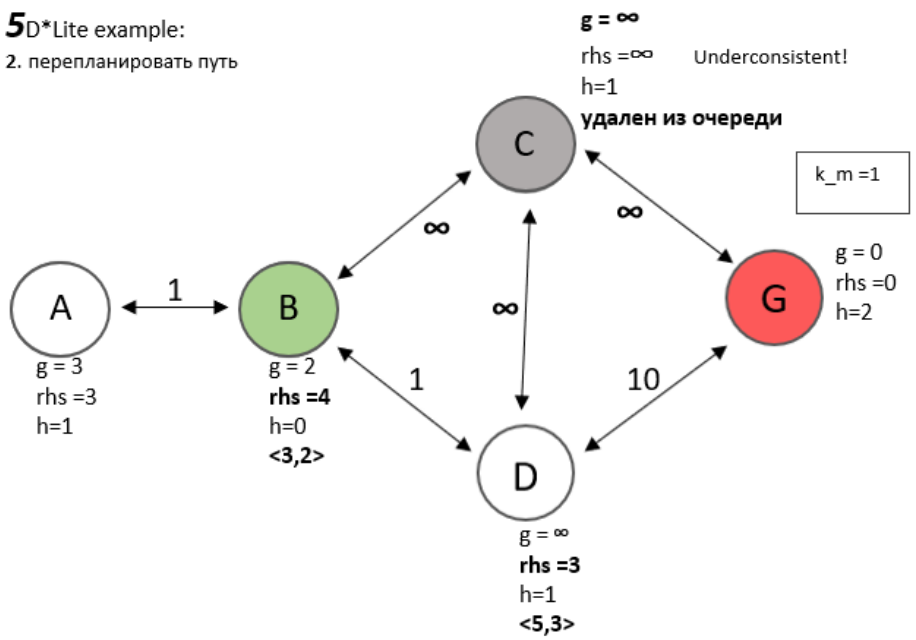




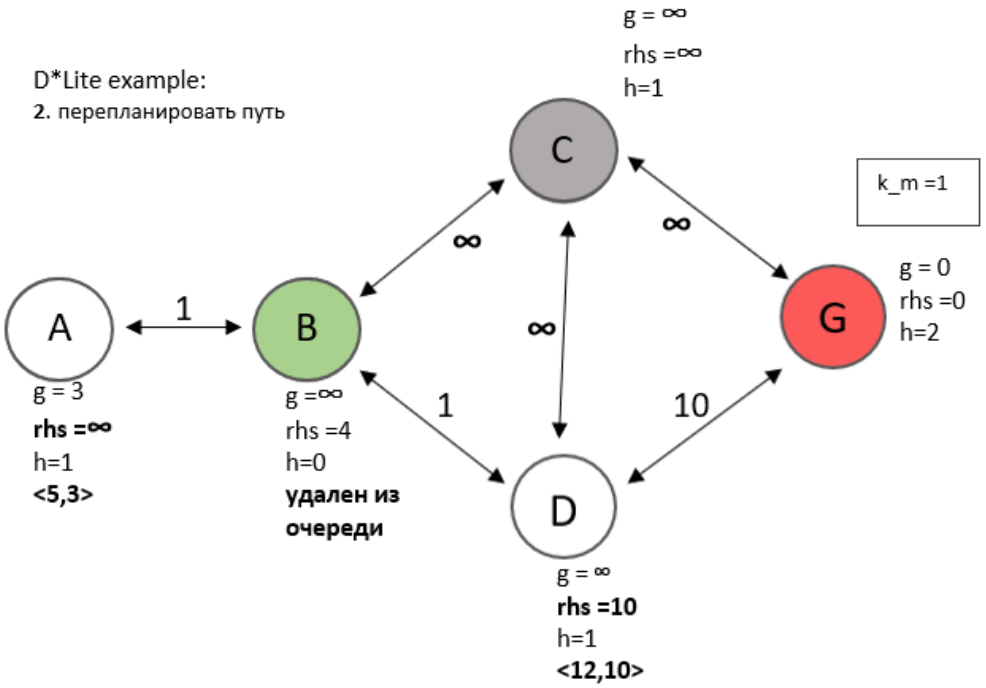
Далее мы собираемся обновить значения rhs узла B и его соседей.
$OpenList = [< 3, 1 >; < 3, 2 >< 5, 3 >]$



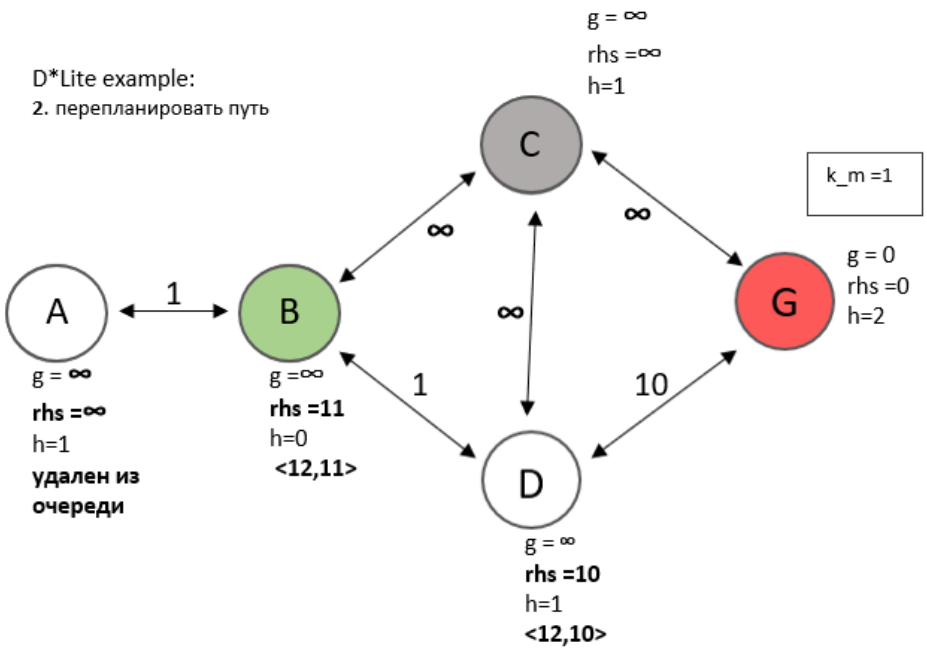
Далее мы удаляем из очереди узел C.Проверка $computeShortestPath(g(s) < rhs(s))$,следовательно $g(s) = \infty$



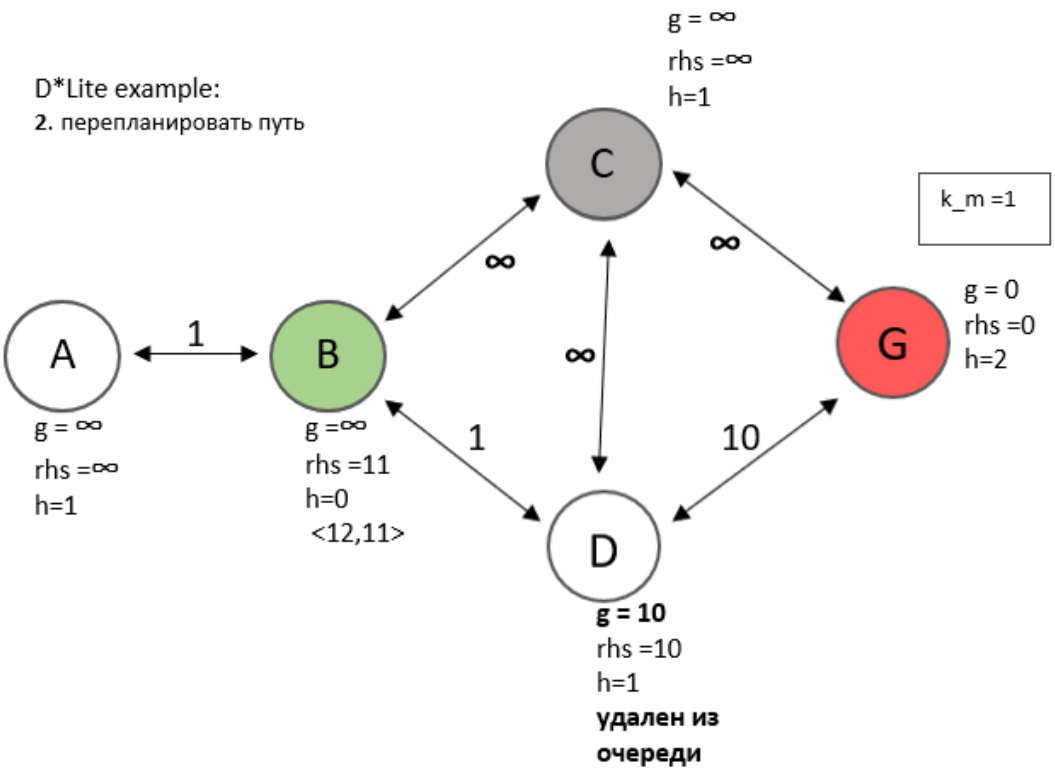
Далее мы удаляем из очереди узел B так как это наименьшее значение в очереди.Проверка $computeShortestPath(g(s) < rhs(s))2 < 4$,следовательно $g(s) = \infty$. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<5,3>;<12,10>]



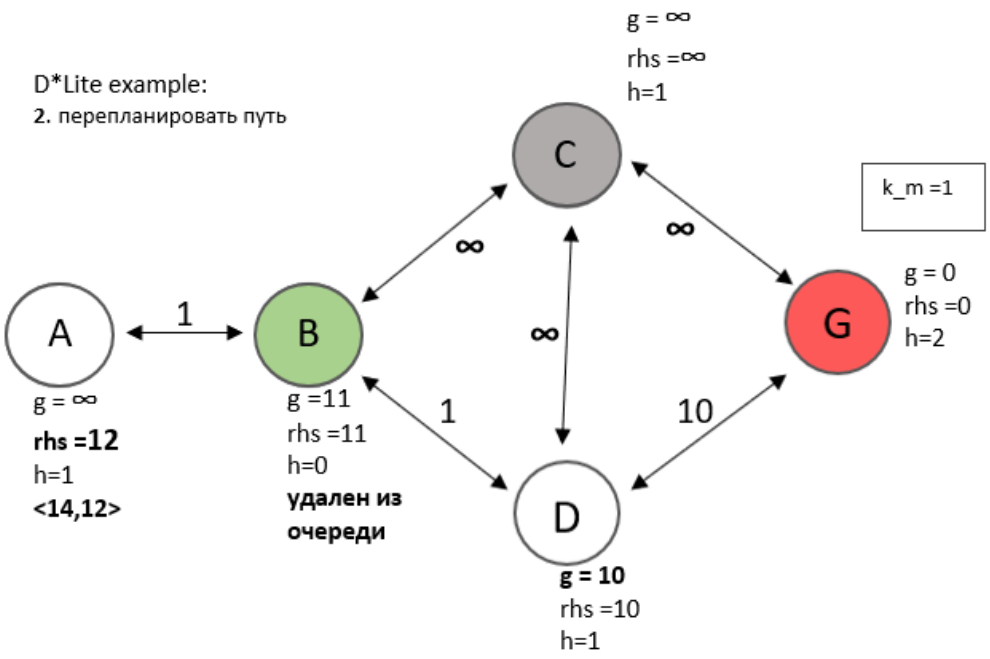
Так как <5,3> меньше чем <12,10> удаляем из очереди вершину A.
Проверка $computeShortestPath(g(s) < rhs(s))3 < \infty$,следовательно $g(s) = \infty$.
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<12,10>;<12,11>]



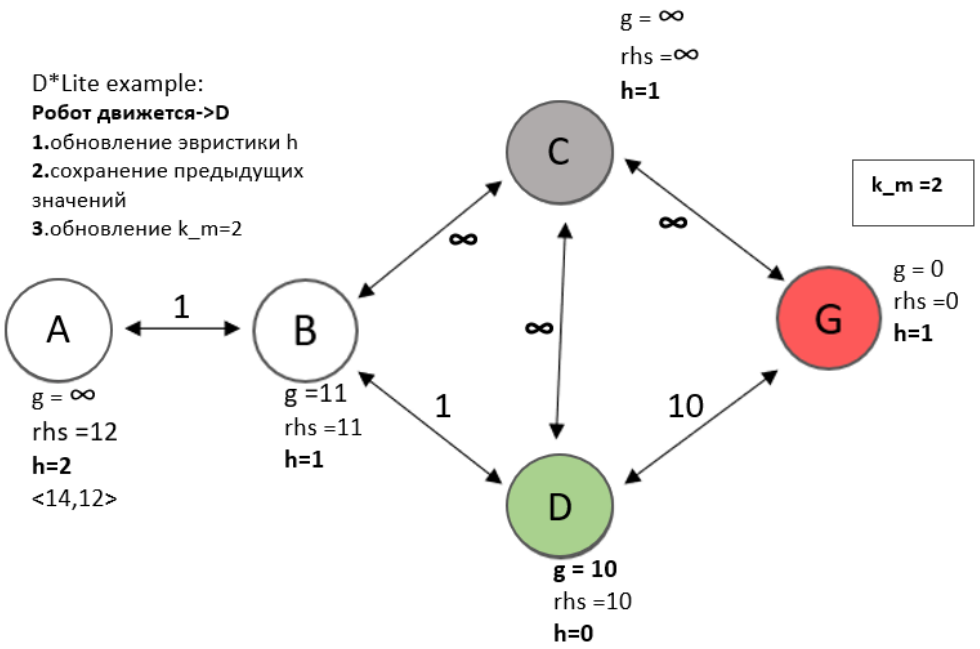
Так как <12,10> меньше чем <12,11> удаляем из очереди вершину D. Проверка $computeShortestPath(g(s) > rhs(s))\infty > 10$,следовательно $g(s) = rhs(s) = 10$. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения. OpenList=[<12,11>]



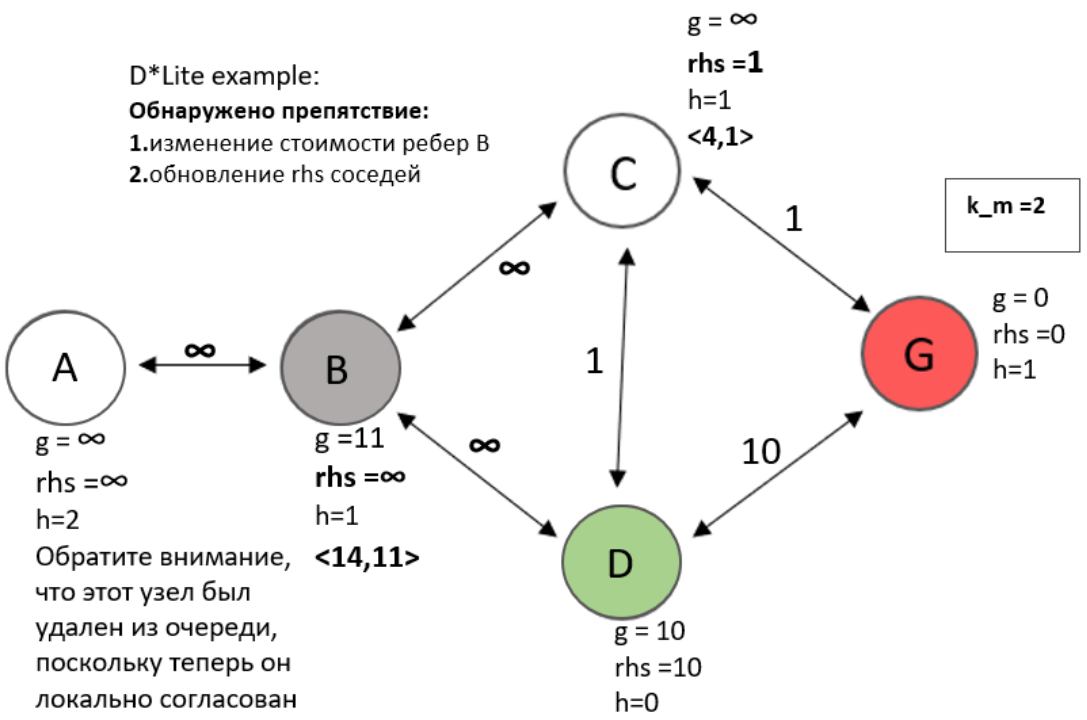
Удаляем из очереди вершину B.Проверка $computeShortestPath(g(s) > rhs(s))\infty > 11$,следовательно $g(s) = rhs(s) = 11$.Теперь узлы в графе локально согласованны. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения. OpenList=[<14,12>]



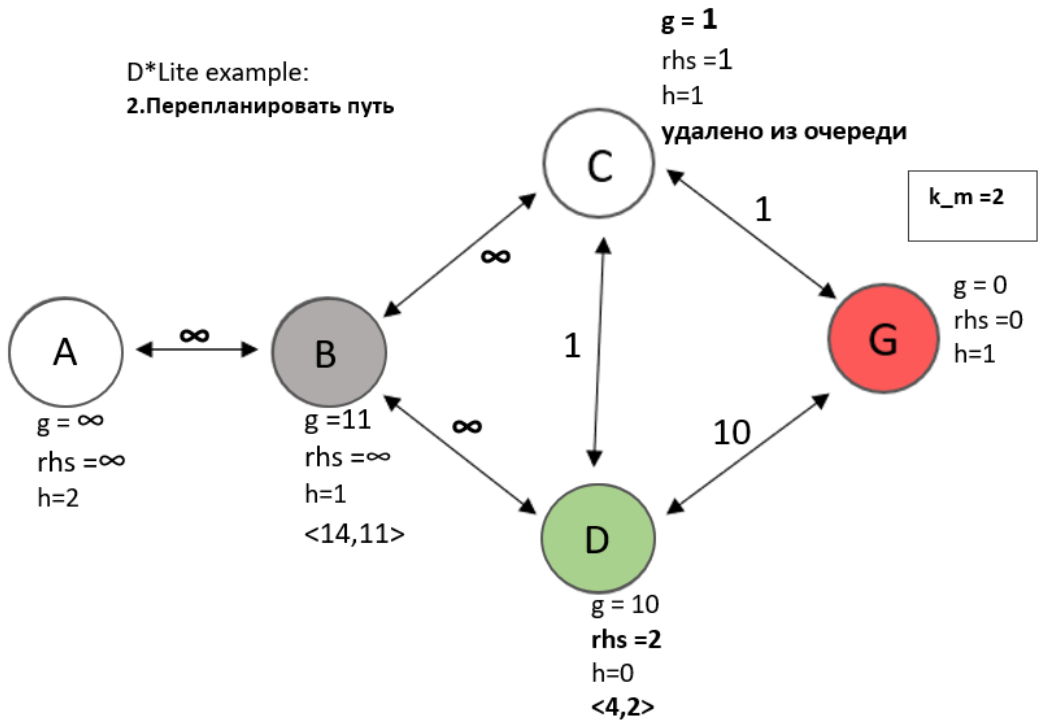
Мы нашли лучший и единственный путь из B->D->G следовательно робот движеться в D.Изменяеться эвристика всех значений узлов на графе. Обновление значения $k_m = 2$



Оказываеться препятсвие тоже движеться.Происходит изменение стоимости ребер узла вершины B,теперь их стоимость равна бесконечности.Также обнавляються значения rhs соседей вершины D. В этом случае значение rhs(G) никогда не меняеться потому что растояние G до самой себя всегда будет равно нулю. OpenList=[<4,1>,<14,1>]

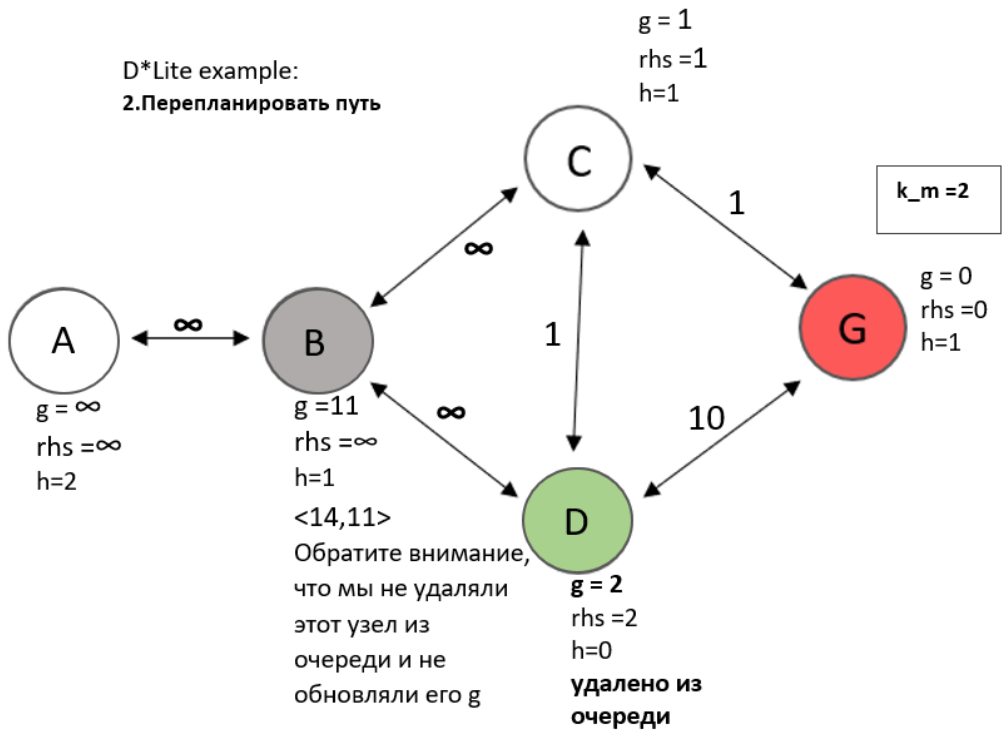


Следовательно мы удаляем из очереди узел C.
Проверка $computeShortestPath(g(s) > rhs(s))\infty > 1$,следовательно $g(s) = rhs(s) = 1$.
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<4,2>;<14,11>]

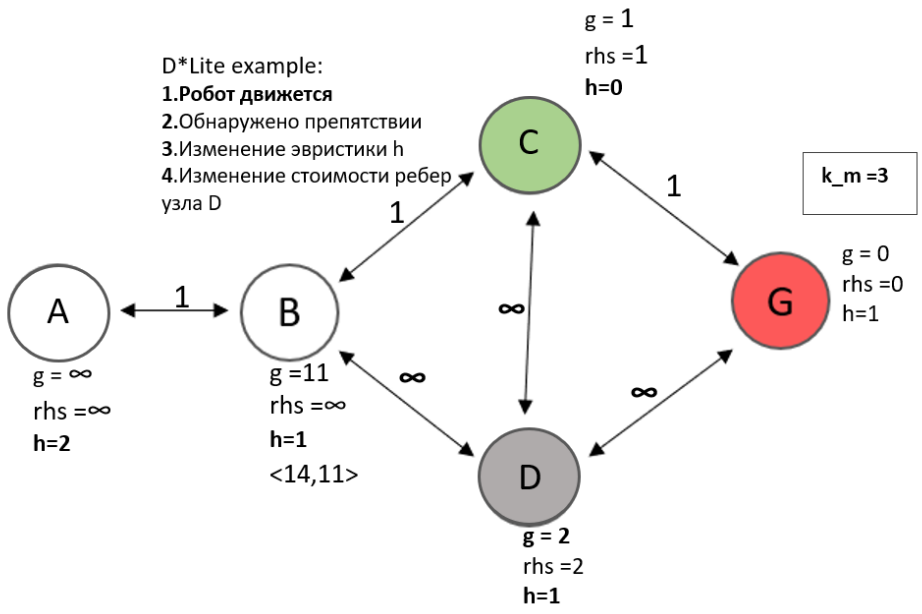


Потом удаляем из очереди узел D.
Проверка $computeShortestPath(g(s) > rhs(s))10 > 2$,следовательно $g(s) = rhs(s) = 2$.

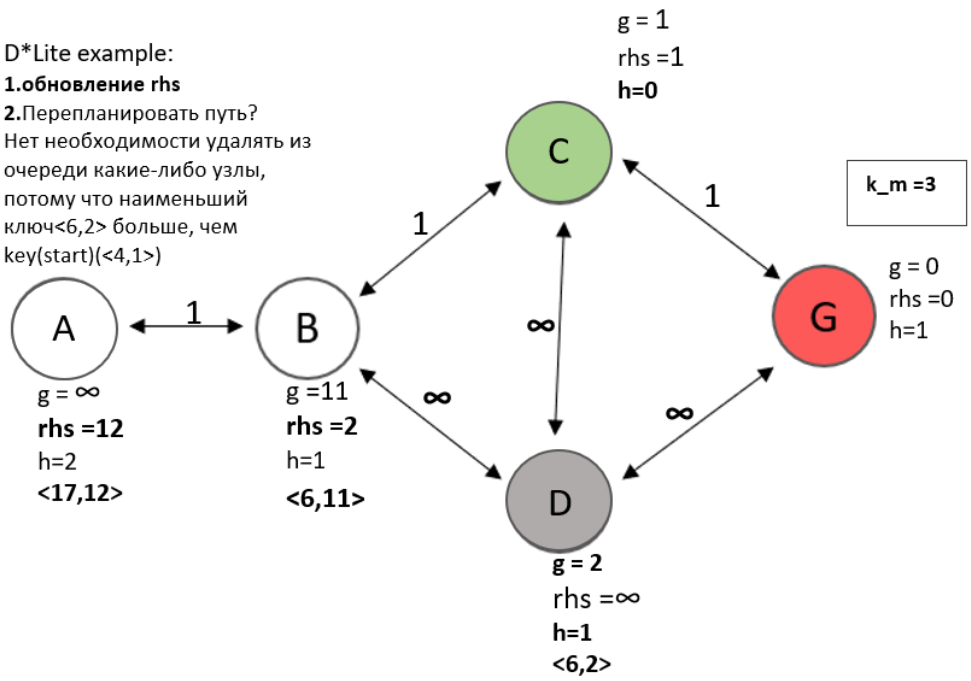
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<14,11>]



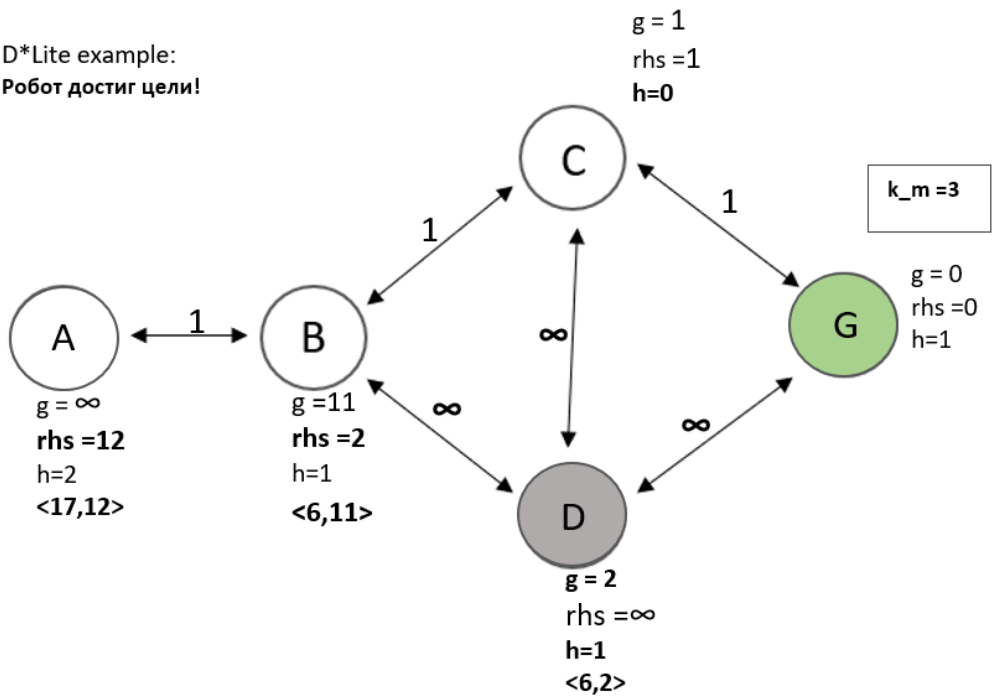
Мы обнаружили путь из D->C->G и робот перемещается в узел C.Также обновляется эвристика для всех улозлов графа.
Препятсвие не стоит на месте и приследует робота.Следовательно ребра узла D становяться стоять бесконечность. Также обновляться значение $k_m = 3$ Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.



D*Lite example:
1.обновление rhs
2.Перепланировать путь?
Нет необходимости удалять из очереди какие-либо узлы, потому что наименьший ключ<6,2> больше, чем key(start)(<4,1>)



D*Lite example:
Робот достиг цели!

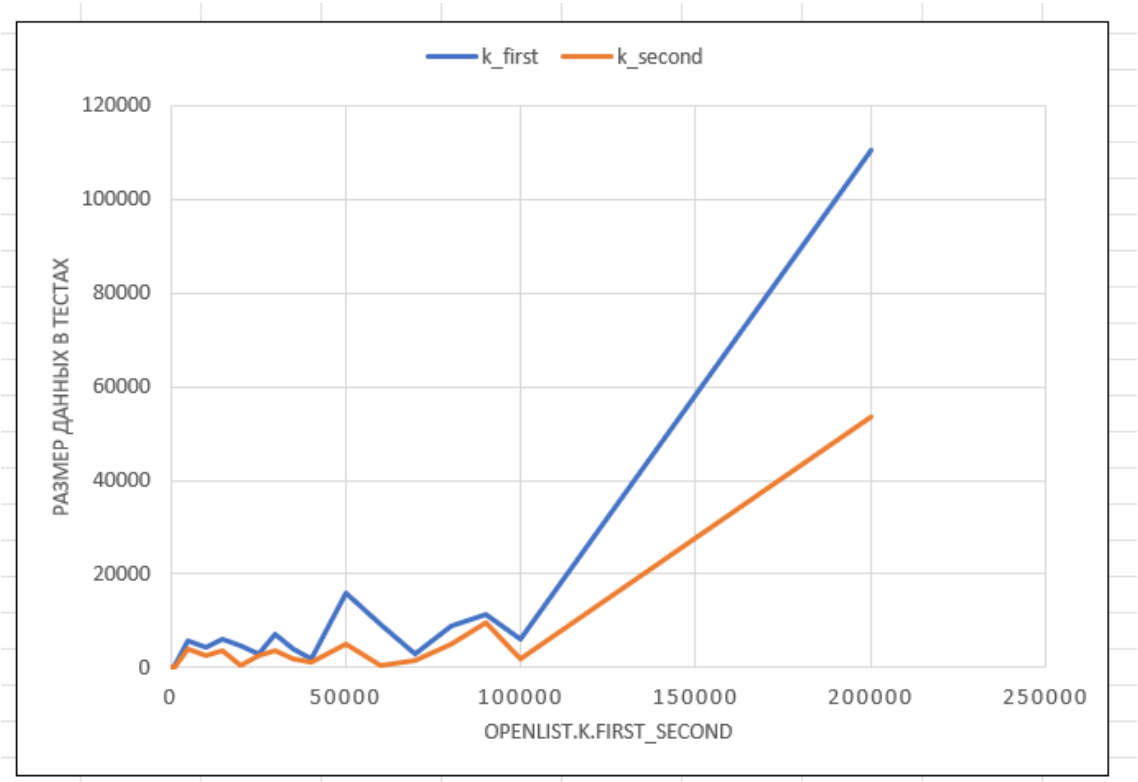


Тестирование и анализ производительности

Тесты написанны в формате in.txt out.txt в одной папке содержаться входные in.txt и выходняе данные 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 представленно то как зависят функции от размера входящих данных.

На данном графике произведен анализ состава $OpenList$(приоритеной очереди).А именно я сравниваю первое значение k_{first} и k_{second} взависимости от размера данных.


k_first
k_second
120000
100000
80000
60000
40000
20000
0
РАЗМЕР ДАННЫХ В ТЕСТАХ
0
50000
100000
150000
200000
250000
OPENLIST.K.FIRST_SECOND

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*.