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

Оглавление

8 Список литературы

Аннотация

В данной работе представлен $Dstar Lite$-алгоритм инкрементного поиска. Инкрементный поиск повторно использует информацию из предыдущих результатов поиска, чтобы найти решения для ряда похожих проблем потенциально быстрее, чем это возможно при решении каждой проблемы с нуля. Это важно, поскольку многим системам искусственного интеллекта приходиться постоянно адоптировать свой план к изменения окружающей среды. 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 Свеном 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
$f(s) = g(s) + h(s, s_{goal})$

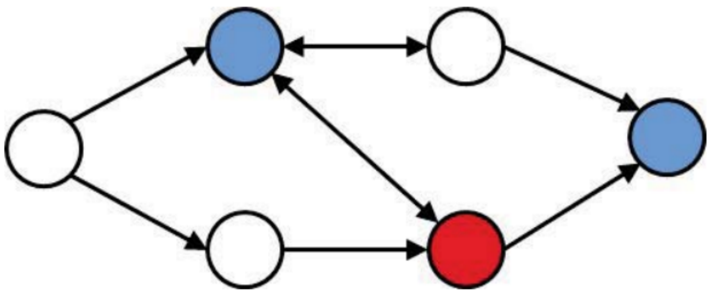
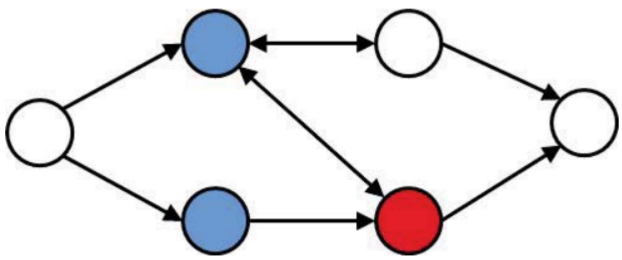
- $f(s)$ -общая стоимость вершины
- $g(s)$ - стоимость перехода от одно узла к другому
- $h(s, s_{goal})$ -эвристика стоимость получения вершины от s, s_{goal}

Вторая точка зрения рассматривает “ослабление” веса ребер. Метод заключается в том что бы выбирать только те вершины, которые мы хотим эффективно и рационально использовать.
Но как по выше определенной формуле различить два узла с одинаковой общей стоимостью в очереде?
Итак мы вводим пару,состоящую из 2 значений $f(s) =< f_1(s), f_2(s) >$

- $f_1(s) = g(s) + h(s, s_{goal})$.
- $f_2(s) = g(s)$ - 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 $f2$, мы выбираем узел с наименьшим значение $g(s)$

Концепция предшественников и приемников

- $Succ(s)$ -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.
- 

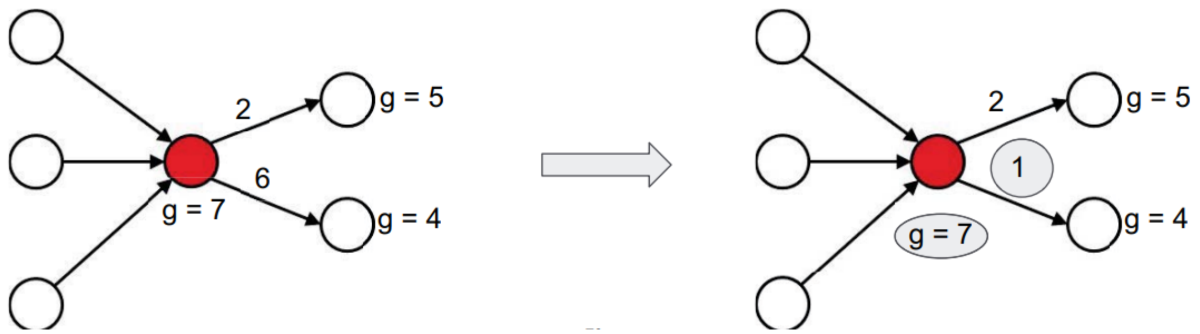
Поиск по принципу”best-first search”

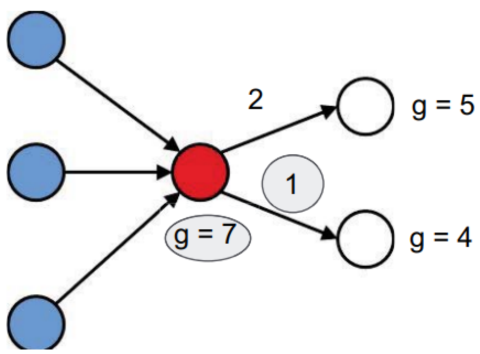
$DstarLite$ использует поиск по принципу ”best-first search”. Алгоритм поиска, исследующий граф путём расширения наиболее перспективных узлов, выбираемых в соответствии с указанным правилом.
Best-First-Search(Graph g, Node start)

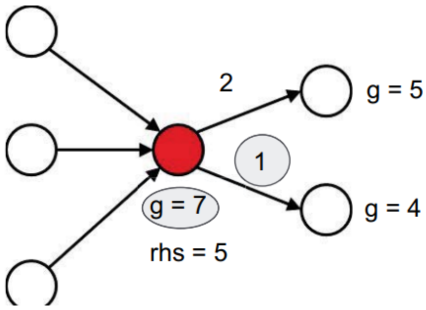
1. Создайте пустую PriorityQueue
    - PriorityQueue pq;
2. Insert ”start” in pq.
    - pq.insert(start)
3. До тех пор пока PriorityQueue не станет пустой
    - u = PriorityQueue.DeleteMin
    - If u is the goal
        - Exit
    - Else
        - Foreach neighbor v of u
            * If v ”не посещена”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

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 несоотвествия

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

Обновление и расширение узлов

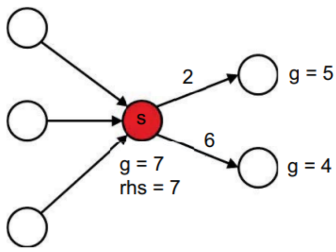


рис.1

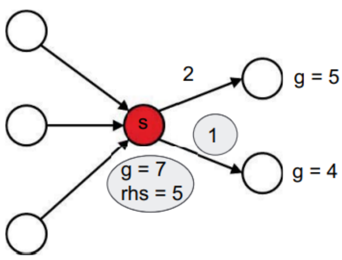


рис.2

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 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

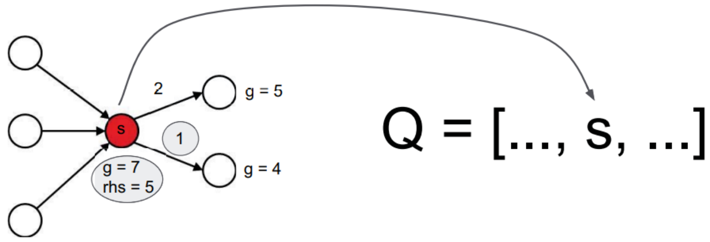


рис.3

Значение приоритетной очереди основанно на этой формуле
$Q =< min[g(s), rhs(s)] + h(s, s_{start}), min[g(s), rhs(s)] >$
Суть заключается в том чтобы взять минимальное значение одно из параметров$g(s), rhs(s)$Порядок очереди заключается в принцепе от меньшего к большему.

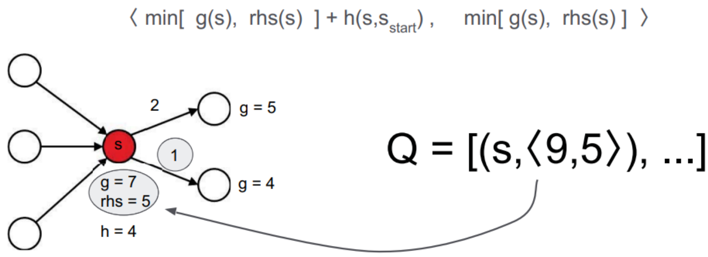


рис.4

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.

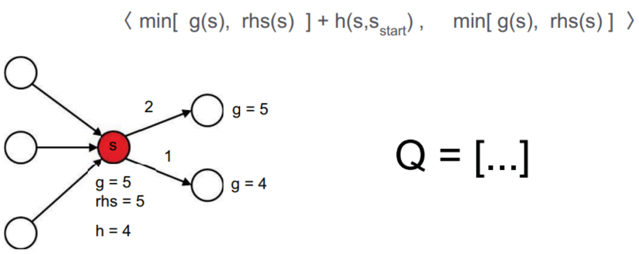


рис.5

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 $computeShortestPath($

```
int Dstar::computeShortestPath() {

    list<state> s;
    list<state>::iterator i;

    if (openList.empty()) return 1;
    int k=0;
    while ((!openList.empty()) &&
           (openList.top() < (s_start = calculateKey(
              s_start))) ||
           (getRHS(s_start) != getG(s_start))) {

        if (k++ > maxSteps) {
            return -1;
        }
        state u;
        bool test = (getRHS(s_start) != getG(s_start));

        while(1) {
            if (openList.empty()) return 1;
            u = openList.top();
            openList.pop();
            if (!isValid(u)) continue;
            if (!(u < s_start) && (!test)) return 2;
            break;
        }
        ds_oh::iterator cur = openHash.find(u);
        openHash.erase(cur);
        state k_old = u;
        if (k_old < calculateKey(u)) {
            insert(u);
        } else if (getG(u) > getRHS(u)) {
            setG(u,getRHS(u));
            getPred(u,s);
            for (i=s.begin();i != s.end(); i++) {
                updateVertex(*i);
            }
        } else {
            setG(u,INFINITY);
            getPred(u,s);
            for (i=s.begin();i != s.end(); i++) {
                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.
$k_m = k_m + h(s_{last}, s_{start})$
Происходит переход от предыдущей начальной вершины,который указан как последний к новой начальной вершине.Тепер когда мы добавляем новые вершины в очередь,вместо того,чтобы вычитать это значение из всего что уже находиться в очереди,мы просто увеличиваем для всех вершин,которые мы помещаем в очередь,их значения на этот новый модификатор.При выхода 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):

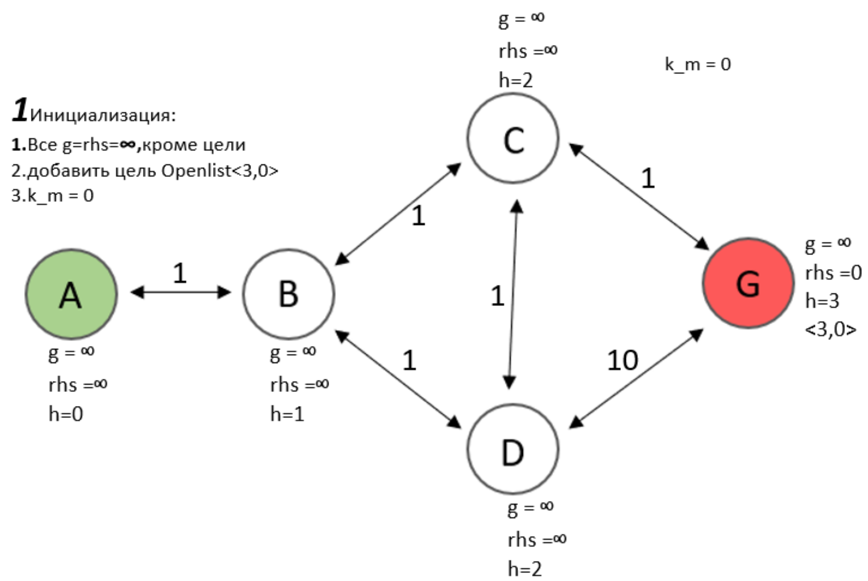
1. Инициализация всех значений $g(s) = \infty$, $rhs(s, s_{goal}) = \infty$, $rhs(s_{goal}) = 0$, $k_m = 0$, $s_{last} = s_{start}$,
2. Поиск по принципу"best-first search"(лучший — первый),покаs_{start} не будет локально согласован и расширен
3. Переместить $s_{start} = argmin_{s' \in Succ(s^{start})}(c(s_{start}, s') + g(s'))$
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 алгоритма

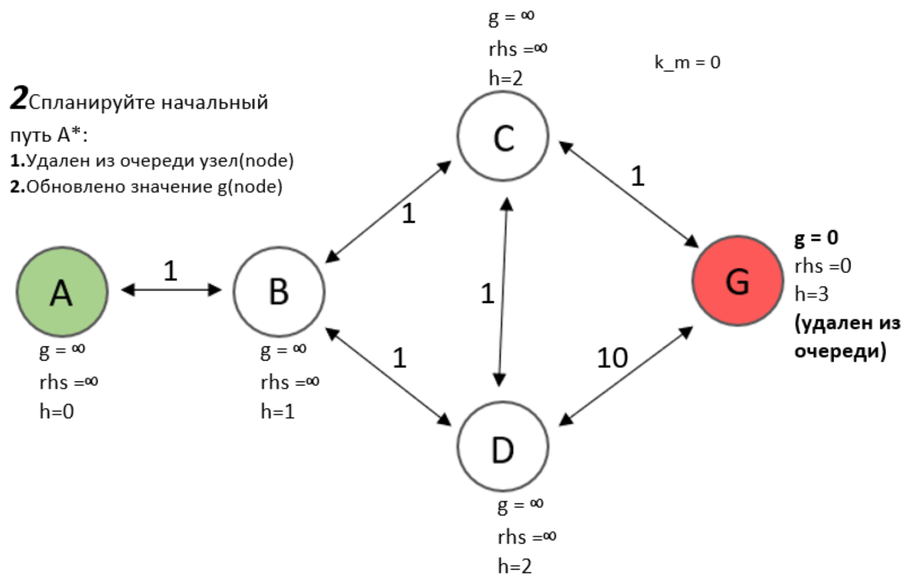
Рассмотрим краткий пример с пятью вершинами.Робот начинает с вершины A и его цель найти кратчайший путь к вершины G,за исключением того что граф может

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)

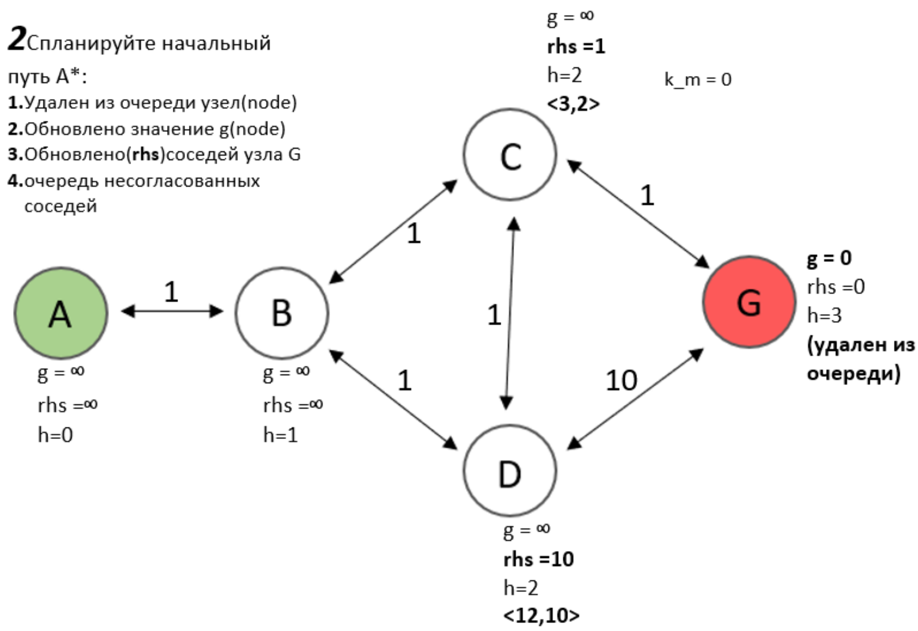


Мы помещаем вершину G в Openlist(очередь) используя формулу $< min[g(s), rhs(s)]+h(s, s_{start})+k_m, min[g(s), rhs(s)] >$ Теперь в очереди находиться одна вершина.Так что нам придется выйти из очереди.

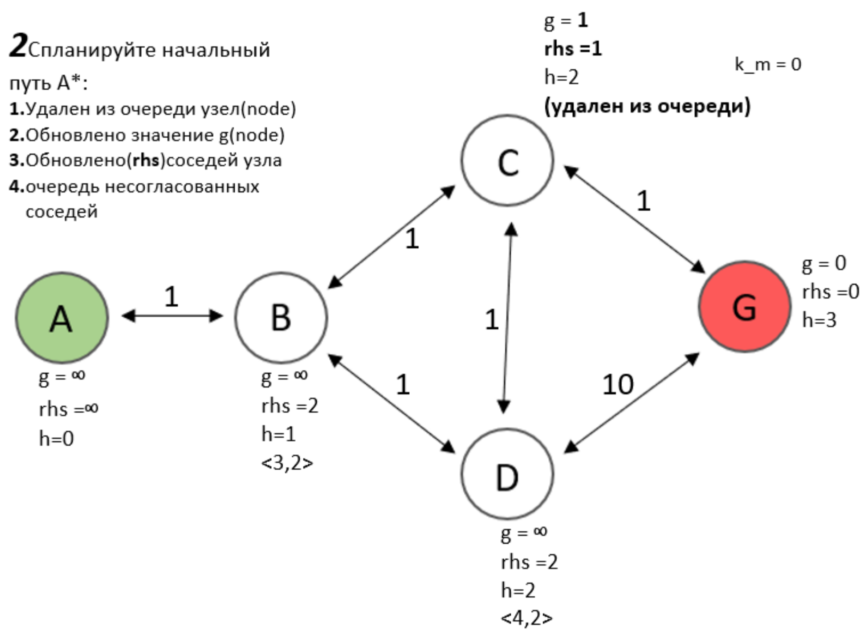
- удаляем узел G из OpenList
- проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=0



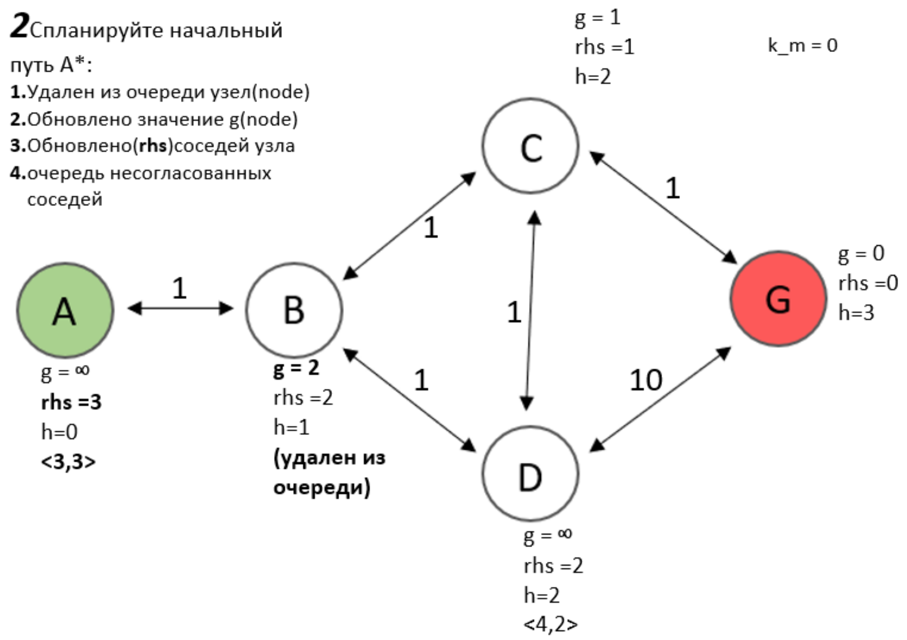
Теперь мы собираемся обновить значения $rhs(s)$ соседей вершин G и в каждом случае значение будет минимальным по формуле $rhs(s) = min_{s' \in Succ(s)}(g(s') + (s, s')$.И добавить в очередь $OpenList = [< 3, 2 >; < 12, 10 >]$



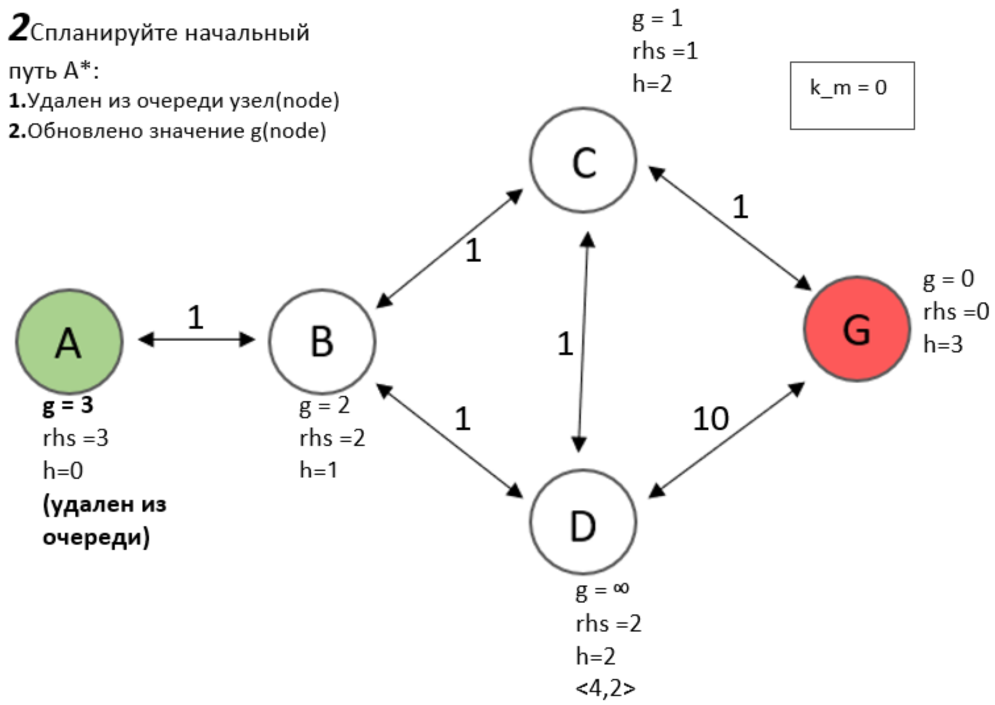
Теперь мы исключим из очереди еще одну вершину C так как у него наименьший ключ <3,1> .Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=1 И обновим значения rhs(s) соседей вершины C.
Теперь в очереди $OpenList = [< 3, 2 >; < 4, 2 >]$



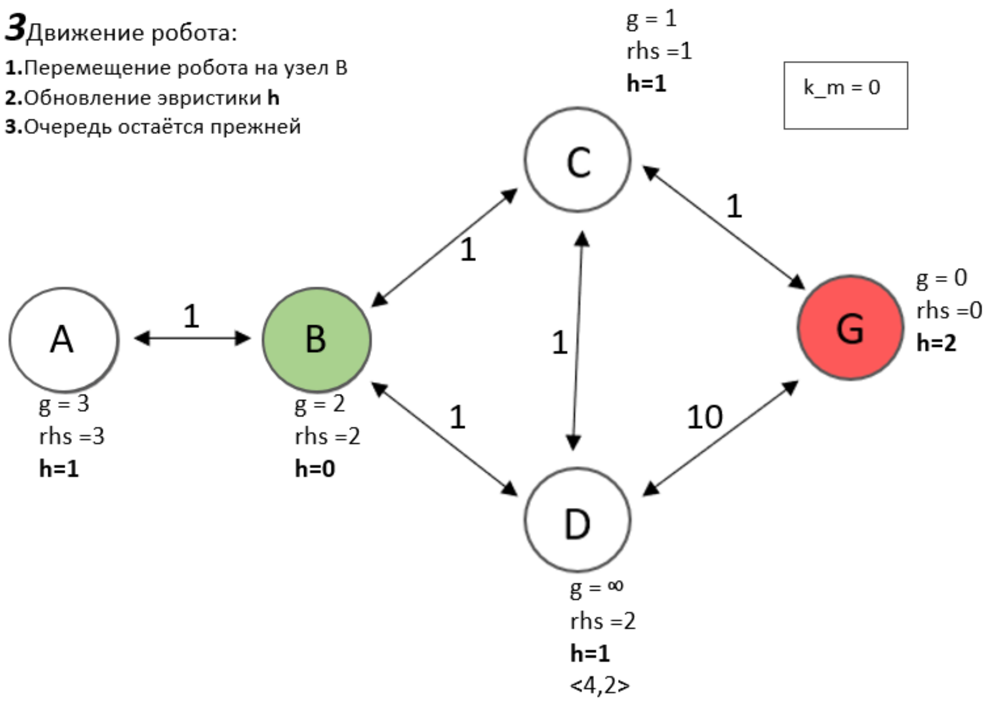
Дальше мы удаляем из очереди вершину B так как у него наименьший ключ <3,2>Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=2 И обновим значения rhs(s) соседей вершины B.Мы не изменяем ключево значение вершины D так как стоимость из G->C->D будет меньше чем из G->C->B->D
Теперь в очереди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

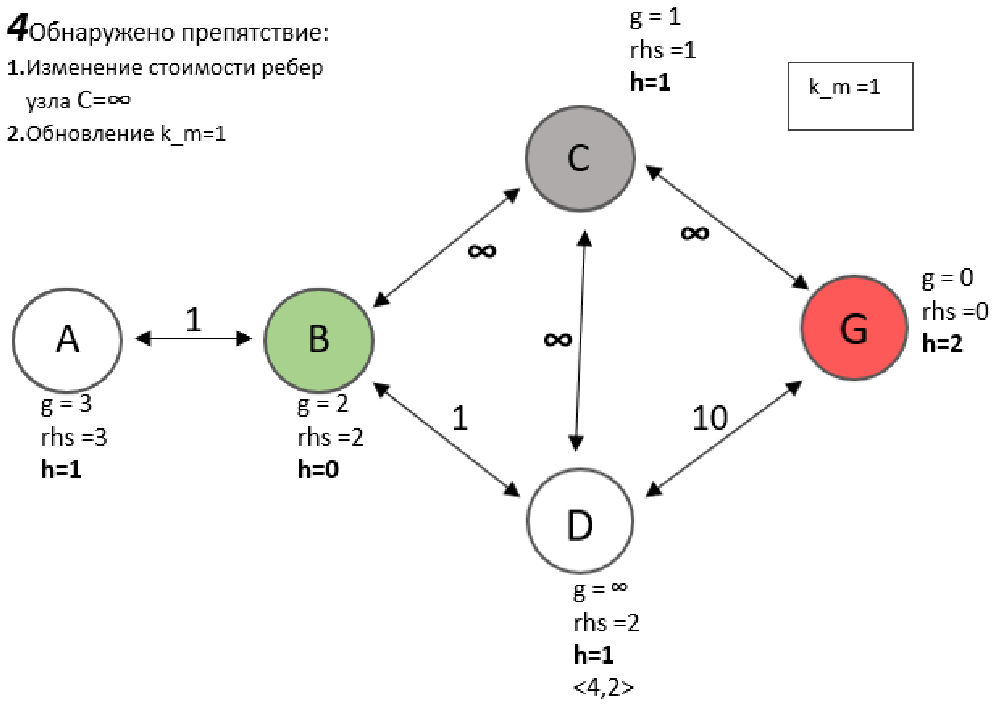


Далее мы удаляем из очереди вершину A так как наименьший ключ.Проверка computeShortestPath(g(s)>rhs(s)),следовательно g(s)=rhs(s)=3. $OpenList = [< 4, 2 >]$ И на это мы закончили.

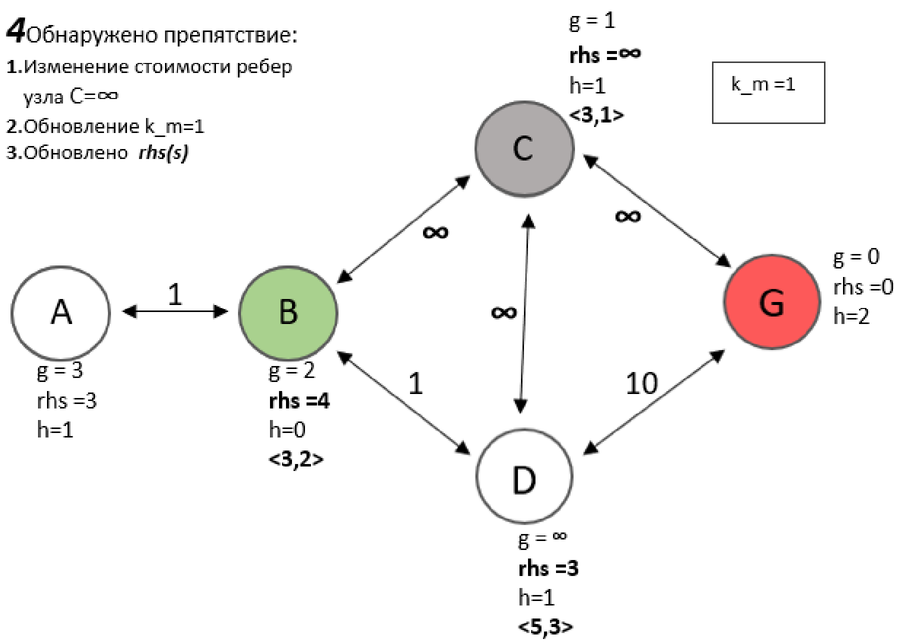


Робот переместился к своей лучшей вершине B.Мы обновляем значения эвристики h для каждой вершины в графе. Мы не изменяем очередь, но также должны обновить $k_m = 1$ Появляеться препятсвие.В точке C есть препятсвие, которе робот теперь может видеть так как он ближе переместился к вершине C, но раньше 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.

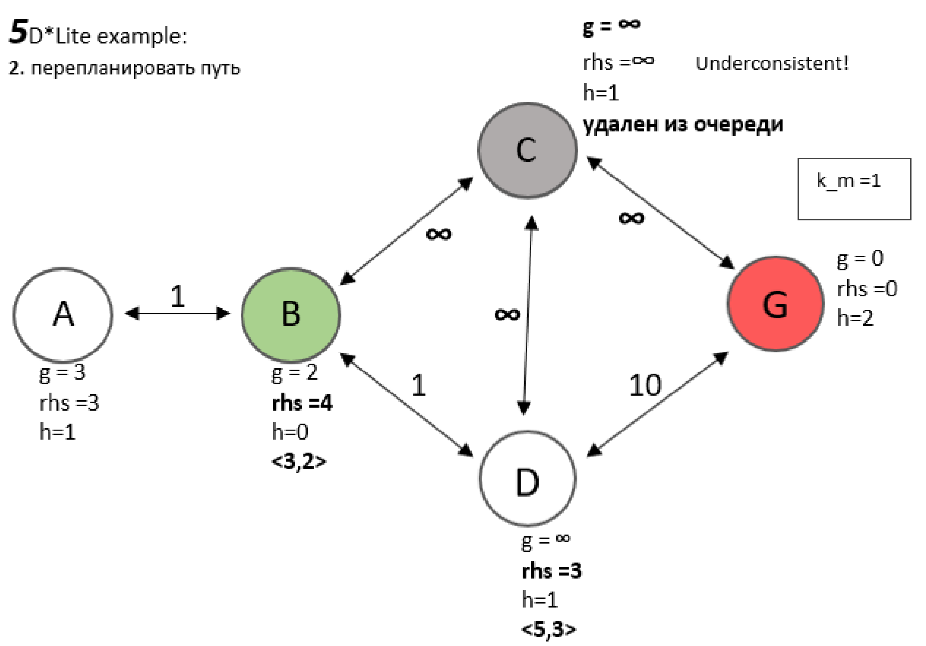




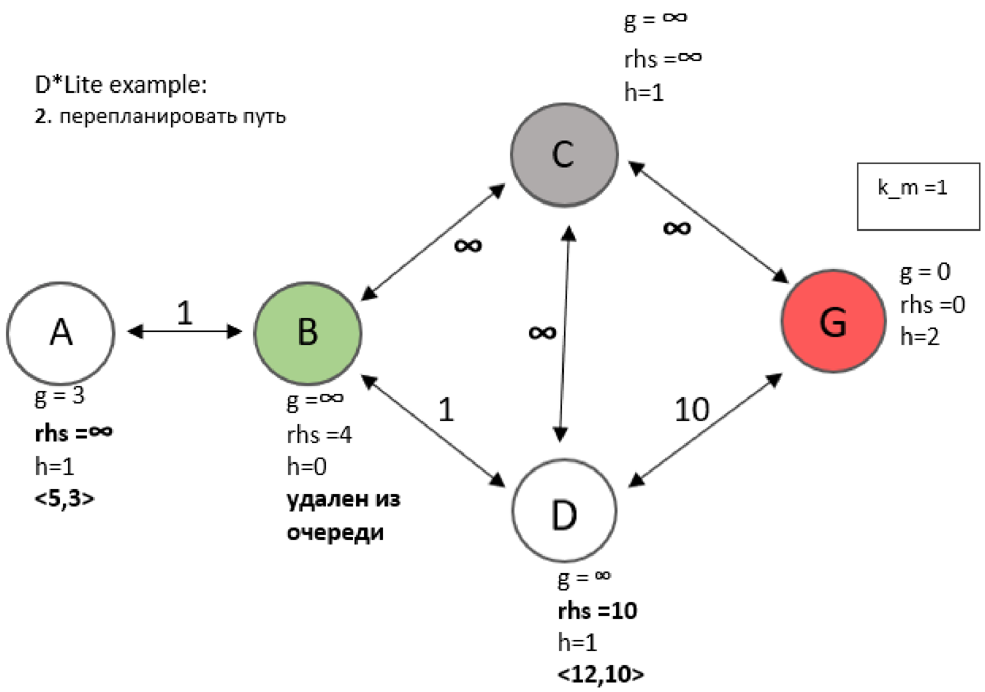
Далее мы собираемся обновить значения rhs вершины B и его соседей.
$OpenList = [< 3, 1 >; < 3, 2 >< 5, 3 >]$



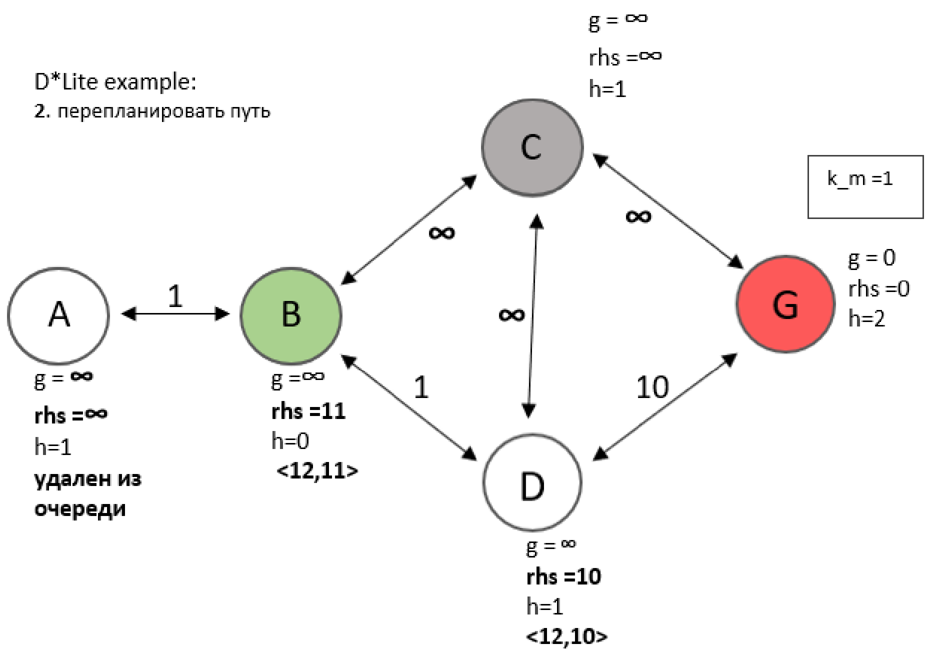
Далее мы удаляем из очереди вершину С.Проверка $computeShortestPath(g(s) < rhs(s))$,следовательно $g(s) = \infty$



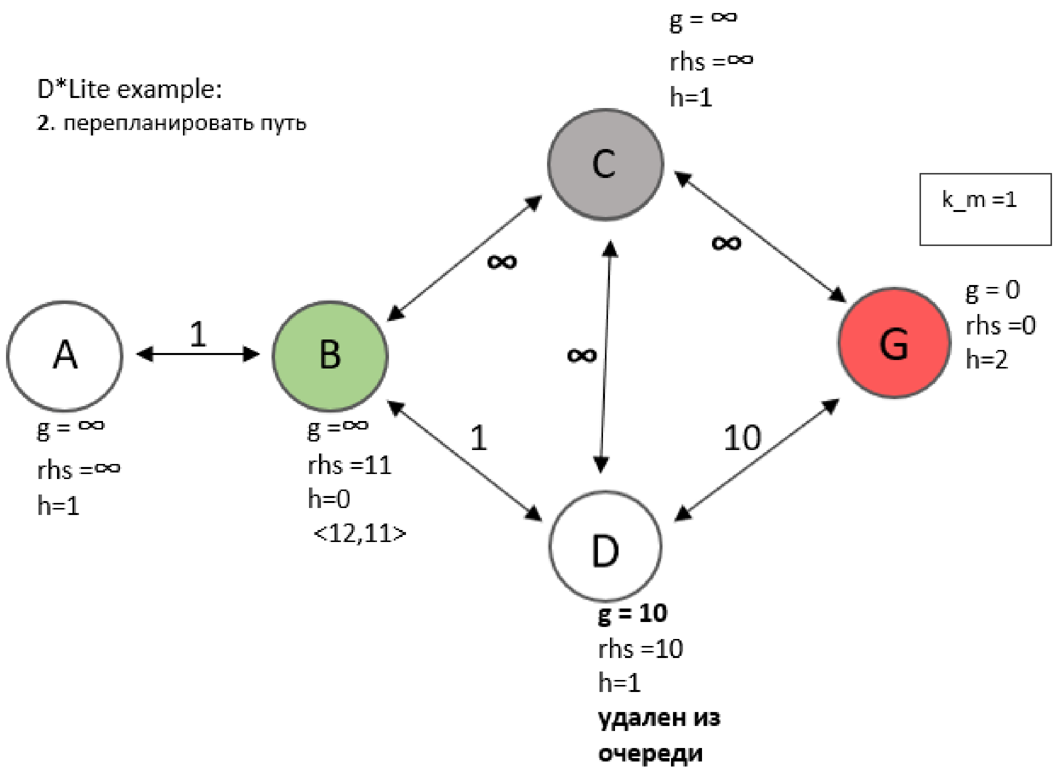
Далее мы удаляем из очереди вершину B так как это наименьшее значение в очереди.Проверка $computeShortestPath(g(s) < rhs(s))2 < 4$,следовательно $g(s) = \infty$. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<5,3>;<12,10>]



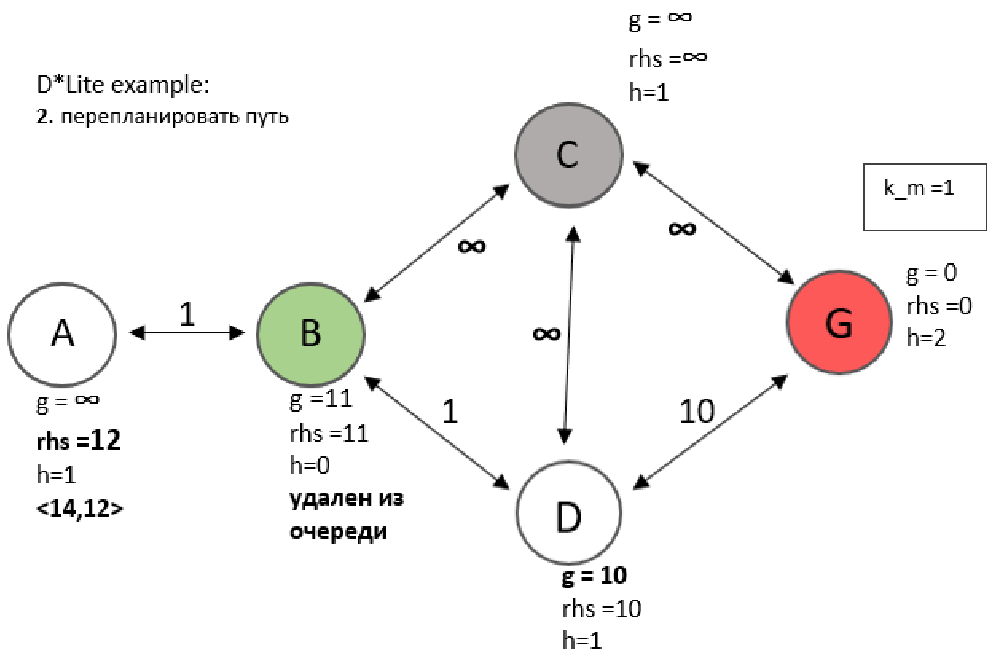
Так как <5,3> меньше чем <12,10> удаляем из очереди вершину A.
Проверка $computeShortestPath(g(s) < rhs(s))3 < \infty$,следовательно $g(s) = \infty$.
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<12,10>;<12,11>]



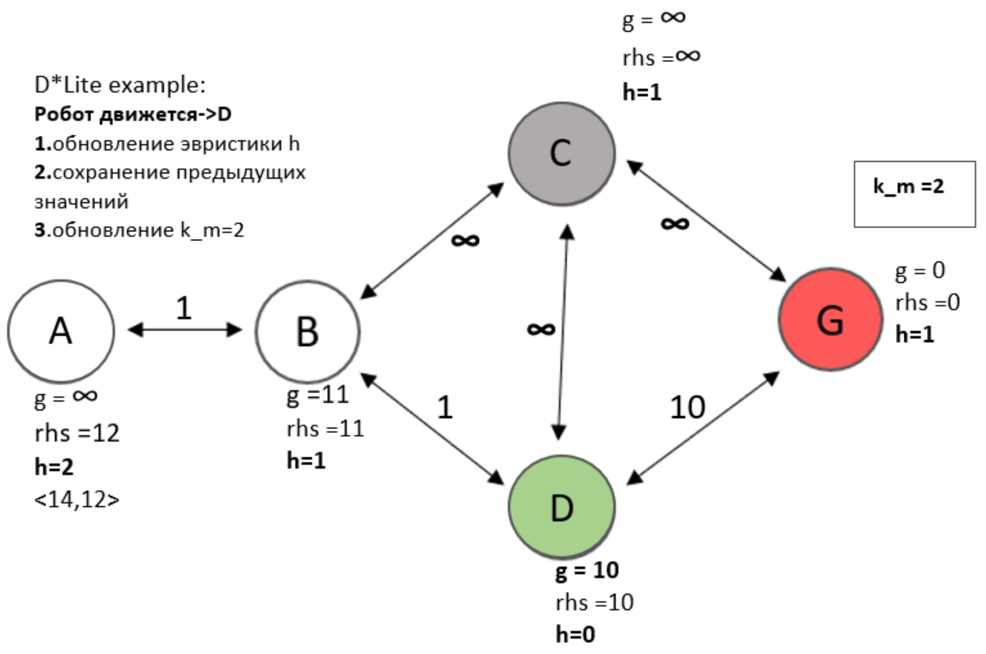
Так как <12,10> меньше чем <12,11> удаляем из очереди вершину D.
Проверка $computeShortestPath(g(s) > rhs(s))\infty > 10$,следовательно $g(s) = rhs(s) = 10$. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<12,11>]



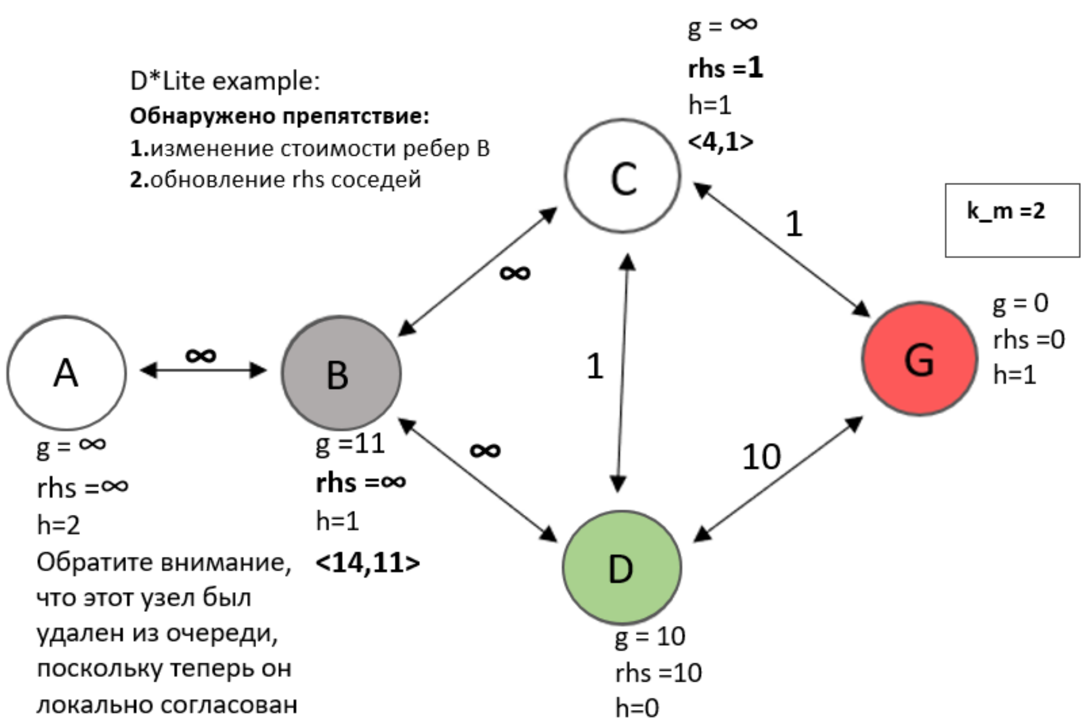
Удаляем из очереди вершину B.Проверка $computeShortestPath(g(s) > rhs(s))\infty > 11$,следовательно $g(s) = rhs(s) = 11$.Теперь узлы в графе локально согласованны. Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения. OpenList=[<14,12>]



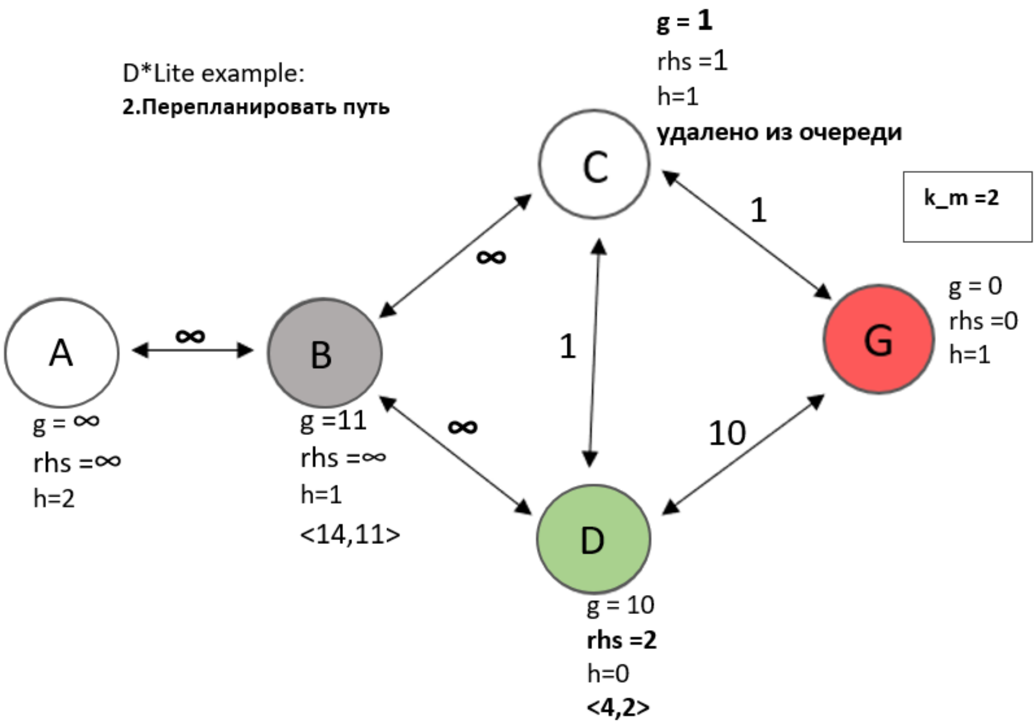
Мы нашли лучший и единственный путь из B->D->G следовательно робот движеться в D.Изменяеться эвристика всех значений вершин на графе. Обновление значения $k_m = 2$



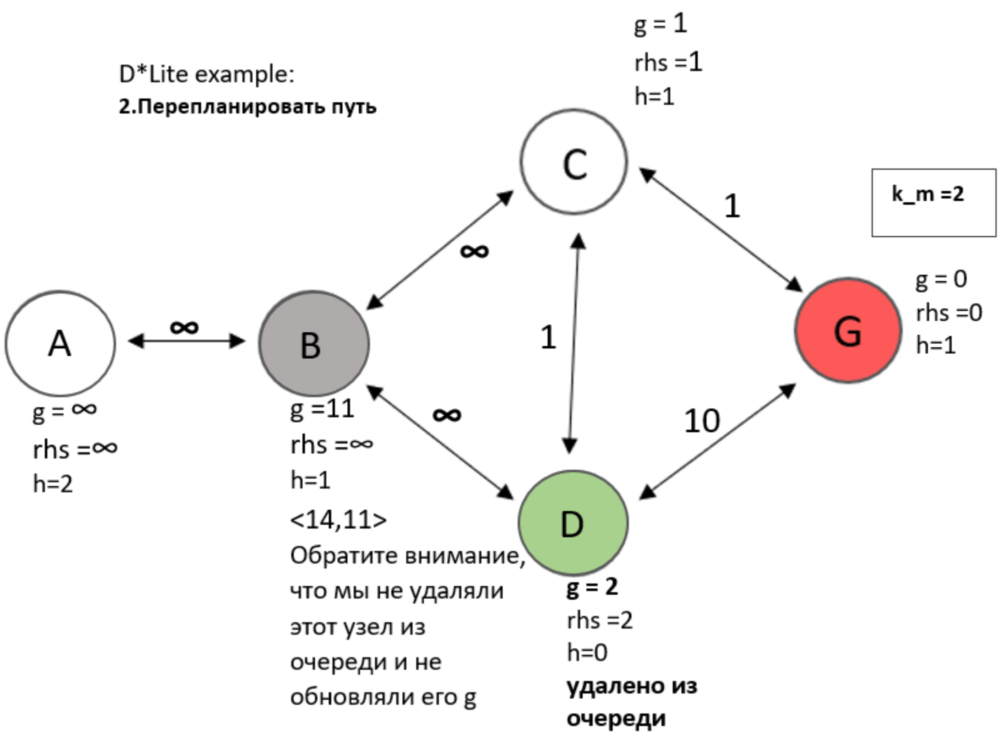
Оказываеться препятсвие тоже движется.Происходит изменение стоимости ребер вершины B,теперь их стоимость равна бесконечности.Также обнавляються значения rhs соседей вершины D. В этом случае значение rhs(G) никогда не меняется потому что растояние G до самой себя всегда будет равно нулю. OpenList=[<4,1>,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



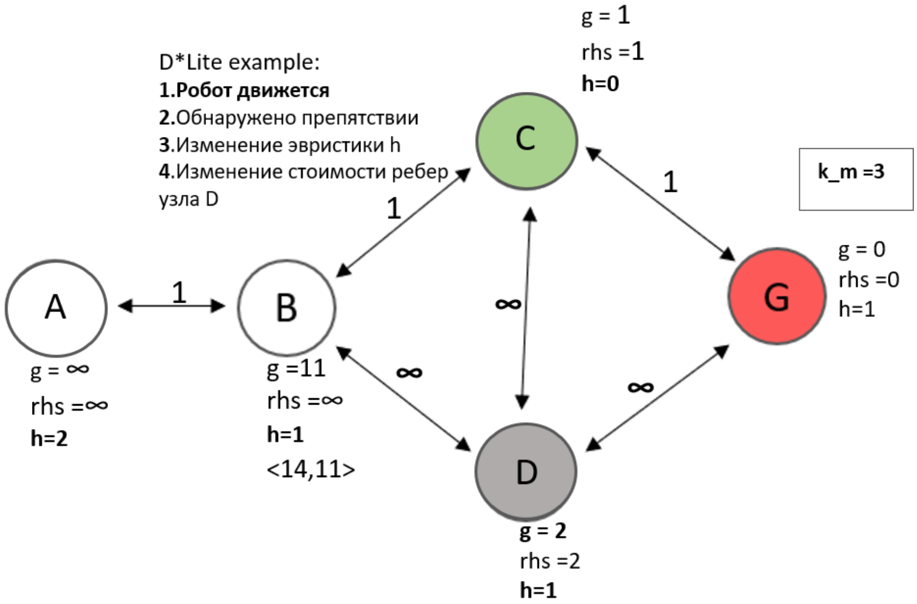
Следовательно мы удаляем из очереди вершину C.
Проверка $computeShortestPath(g(s) > rhs(s)) \infty > 1$,следовательно $g(s) = rhs(s) = 1$.
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<4,2>;<14,11>]

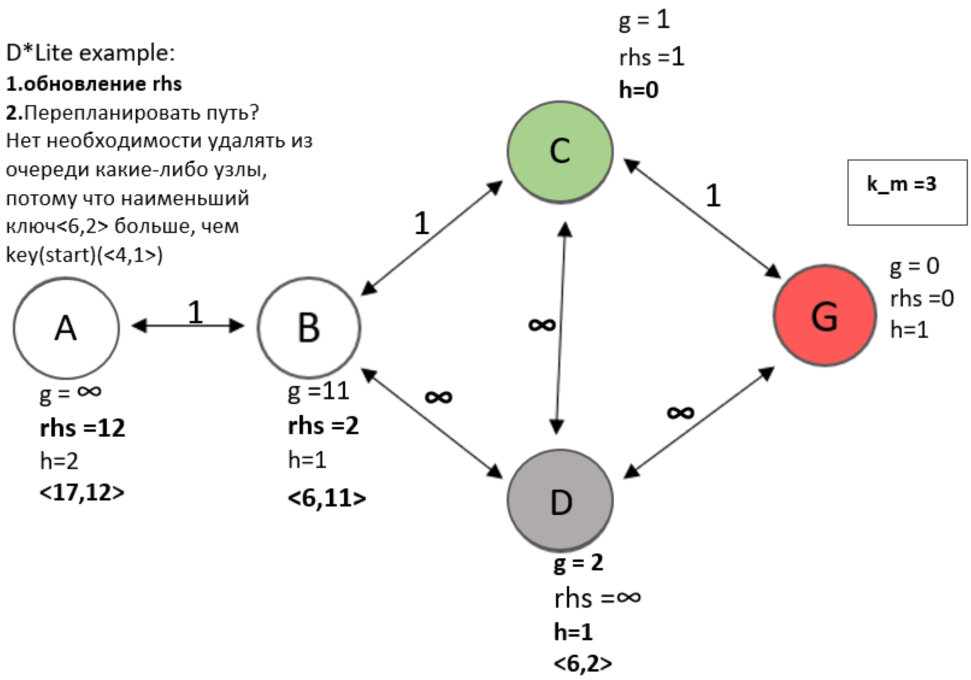


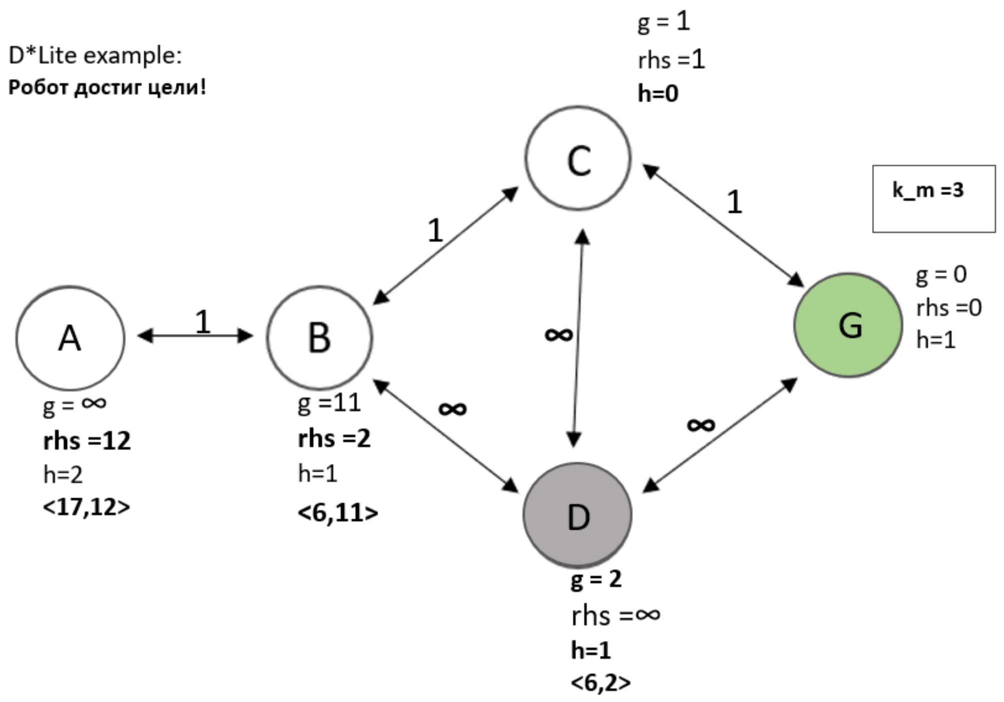
Потом удаляем из очереди вершину D.
Проверка $computeShortestPath(g(s) > rhs(s))10 > 2$,следовательно $g(s) = rhs(s) = 2$.
Обновляем значение rhs соседей и добавляем в OpenList изменившиеся значения.
OpenList=[<14,11>]



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


D*Lite example:
1.Робот движется
2.Обнаружено препятствии
3.Изменение эвристики h
4.Изменение стоимости ребер узла D
g = 1
rhs =1
h=0
C
k_m =3
1
1
1
∞
∞
∞
A
B
G
D
g = 0
rhs =0
h=1
g = ∞
rhs =∞
h=2
g =11
rhs =∞
h=1
<14,11>
g = 2
rhs =2
h=1


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>)
g = 1
rhs =1
h=0
C
k_m =3
1
1
1
∞
∞
∞
A
B
G
D
g = 0
rhs =0
h=1
g = ∞
rhs =12
h=2
<17,12>
g =11
rhs =2
h=1
<6,11>
g = 2
rhs =∞
h=1
<6,2>

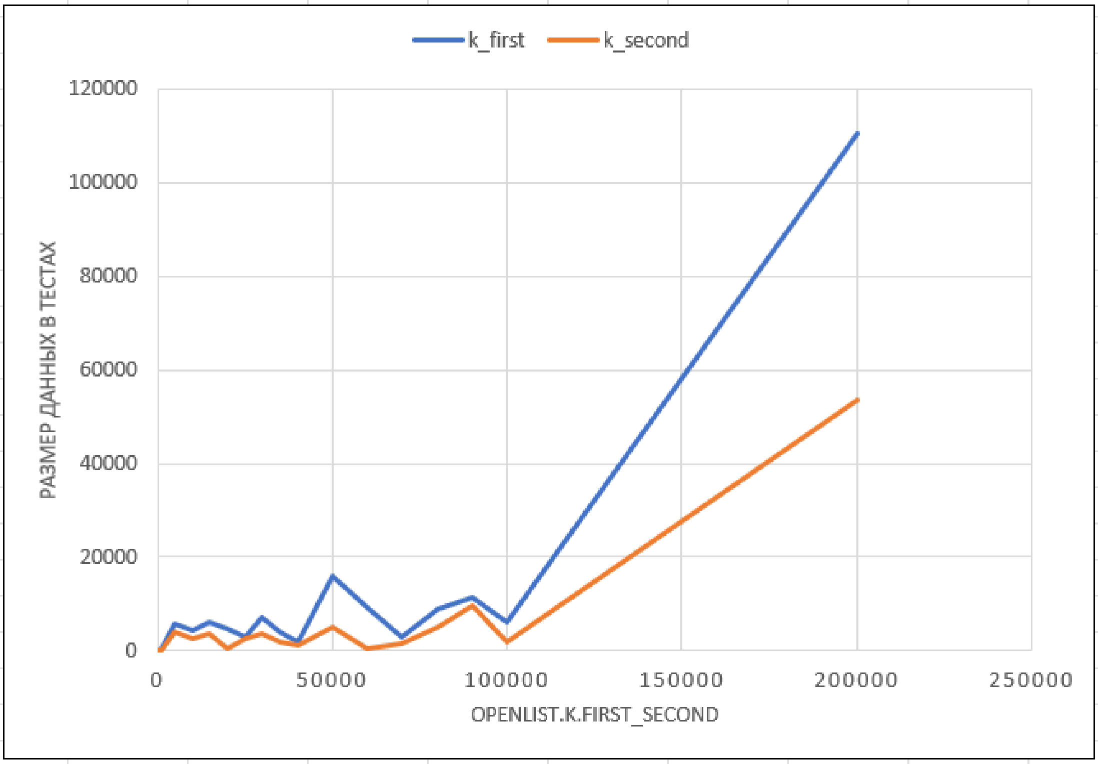


Тестирование и анализ производительности

Тесты написанны 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 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.



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 AAAI/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 Dynamic 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 ofthe International Confer- ence 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/2007/04161010/12OmNASraPs.

[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/2012/4608a661/12OmNvq5jGv`.

[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`.