LPA* и D* lite

Лев Сорвин, Максим Хабаров
под руководством Константина Яковлева

2025-01-29

1 Мотивация

Во многих областях искусственного интеллекта естественным образом появляется планирования путей на больших графах. Большая часть исследований в этой задаче покрывает задачу в случае, когда граф нам целиком известен заранее. Однако многим системам приходится сталкиваться с динамическими изменениями в графе и приходиться адаптироваться к изменившимся условиям, поскольку предыдущий итог планирования может потерять актуальность

2 Постановка задачи

Пусть нам дан граф $G = (V, E)$, где V — конечное множество вершин, $E \subset V \times V \times \mathbb{R}_{\geqslant 0}$ — множество взвешенных ребер. При этом будем считать, что в графе нет кратных ребер, ввиду чего можно ввести функцию веса ребра $w : V \times V \to \mathbb{R}_{\geqslant 0} \vee \infty$ Путем в графе будем называть конечную последовательность $A = [v_1, v_2, \ldots, v_n]$ такую, что для каждого $i = 1, \ldots, n-1$: $w(v_i, v_{i+1}) \neq \infty$. Множество конечных последовательностей вершин мы будем обозначать $P = P(V)$. Весом пути мы будем называть выражение $w(A) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$. Началом пути мы будем называть при этом первый элемент последовательности и обозначать $s(A)$, а концом пути — последний элемент последовательности, и обозначать $d(A)$.

Мы хотим решать задачу минимального пути — нам даны вершины $s, d \in V$ (далее мы считаем, что s, d, V фиксированы) и мы должны

найти

$$A_{plan} = \underset{\substack{A \subset V \text{ — путь в } G \\ s(A)=s \\ d(A)=d}}{\arg\min} w(A).$$

Мы будем также говорить тогда, что $A = \texttt{optpath}(G, s, d)$.

Сформулируем более точно алгоритмическую задачу. На i-ом шаге мы получаем информацию $E_i \subset V \times V \times \mathbb{R}_{\geqslant 0} \vee \infty$ об изменении весов ребер в графе. Тогда *алгоритмом для решения задачи долгосрочного планирования* мы будем называть тройку $(T, \texttt{plan}, \texttt{extract})$ из произвольного множества и двух вычислимых фукнций соответственно, такую что:

$$\begin{aligned}
&\texttt{empty} \in T \\
&\texttt{plan} : E \times T \to T \\
&\texttt{extract} : T \to P(V) \\
&\texttt{extract}(\texttt{plan}(E, \texttt{empty})) = \texttt{optpath}(G, s, d) \\
&\texttt{extract}(t_0) = \texttt{optpath}((V, E'), s, d) \to \\
&\qquad \texttt{extract}(\texttt{plan}(E_{new}, t_0)) = \texttt{optpath}((V, E' \leftarrow E_{new}), s, d)
\end{aligned}$$

Мы будем применять алгоритм в следующем коде (который следует рассматривать как синтаксический сахар над машиной Тьюринга):

```
active_plan = empty
for i in 1..n:
     e_new_i = get_e_new_i()
     active_plan = plan(e_new_i, active_plan)
     shortest_path = extract(active_plan)
```

Мы хотим минимизировать число итераций алгоритма `plan` и `extract` в ходе исполнения псевдокода выше.

3 Метод решения

Несложно заметить, что реализация функции `plan`, которая заново находит кратчайший путь в графе с обновленными ребрами, является решением нашей задачи. Однако оптимальность этого решения оставляет желать лучшего: даже при небольших изменениях ребер в графе нам придется выполнять заново всю процедуру планирования. В связи с этим хочется использовать *инкрементальные*, которые во время планирования сохраняют полезную в дальнейшем информацию, чтобы ее затем

эффективно переиспользовать в новой задаче планирования. Такие алгоритмы уже есть — например, Dynamic SWSF-FP [2].

С другой стороны, использование эвристик в задачах поиска дает на практике очень существенное преимущество, в связи с чем широко используется. Наиболее популярным алгоритмом для задачи планирования с использованием эвристик является алгоритм A^* [1]. Однако алгоритм A^* сам по себе не является инкрементальным в связи с чем его использование в нашей задаче приведет к повторному планированию крайне часто.

Мы хотим объединить эти два подхода — использовать эвристики для направления поиска и использовать инкрементальность для оптимального пересчета решения при появлении новых данных. Для этого мы переиспользуем идею из алгоритма Dynamic SWSF-FP. Как мы знаем, уравнение

$$g^*(v) = \begin{cases} 0 & v = s \\ \min_{v' \in neighbors(v)}(w(v, v') + g^*(v')) \end{cases}$$

задает оценку $g^*(v)$ кратчайшего пути от s до произвольной вершины v. Мы будем для каждой вершины поддерживать оценку веса кратчайшего пути g в том же значении, как и в алгоритме A^*, а также величину

$$rhs(v) = \min_{v' \in neighbors(v)}(w(v, v') + g(v')).$$

Несложно заметить, что если во всех вершинах $rhs(v) = g(v)$, то $g(v) = g^*(v)$ для всех вершин v. Будем называть вершину v *неконсистентной*, если $rhs(v) \neq g(v)$. В ходе алгоритма мы будем итеративно исправлять неконсистентность, пока ее исправление может привести к нахождению кратчайшего пути.

Заметим, что при изменении веса ребер у нас меняются rhs-значения у вершин, связанных с этими ребрами. По нашему алгоритму, мы будем исправлять неконсистентность лишь у необходимого количества вершин, начиная с вершин, находящихся рядом с измененными ребрами.

4 Алгоритм D^* *lite*

Возможность пересчитывать кратчайшие пути позволяет нам решать следующую задачу. Пусть у нас есть агент с ограниченным зрением, который должен дойти до некоторой точки на карте. Изначально у него

нет никакой карты местности, но он обновляет свои данные, когда препятствия попадают в его поле зрения. Агент всегда хочет идти по оптимальному пути к цели, но информация о препятствиях будет заставлять его перестраивать план.

В этой ситуации можно использовать алгоритм LPA^* следующим образом. Агент может на очередном шаге строить кратчайший путь от цели до себя, основываясь на известных ему данных. После очередного шага он использует информацию о полученных препятствиях, чтобы обновить план кратчайшего пути от цели до себя.

5 Экспериментальные исследования

Мы будем исследовать алгоритм D^* *lite* в задаче поиска пути на карте с ограниченной видимостью. Для этого мы используем карты с датасета MovingAI [3]. Мы используем 4 карты, которые обобщают реально встречающиеся топологии у агентов в реальной жизни: план этажа, город, разные виды лабиринтов (см. рис. 1). Ширина каждой карты, кроме Labyrinth, примерно равна 250, ширина Labyrinth же примерно равна 800 (точные размеры карт доступны в репозитории). Здесь и далее, когда мы говорим о малой видимости, мы имеем в виду радиус видимости в 5 клетки, большой видимости — в 50 клеток.

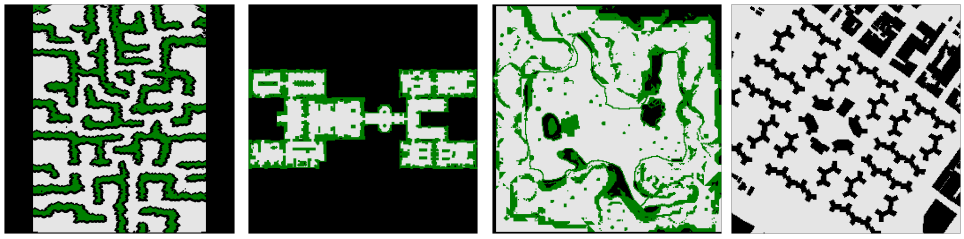
Для каждой карты мы равномерно выберем хотя бы 100 сценариев, выбирая каждый 5 сценарий в файле сценариев, и запустим на них агента с различными радиусами видимости. Будем отслеживать только производительность, считая, что память не является узким местом в областях применения данного алгоритма. В качестве метрик мы возьмем время работы алгоритма на карте, а также для воспроизводимости результатов число операций, модифицирующих очередь с приоритетом (во всех алгоритмах мы использовали для очереди с приоритетом одну и ту же структуру данных) и количество чтений/записей полей вершины.

Во-первых, результаты экспериментов показали, что использование эвристик дает крайне существенный прирост к производительности, в связи с чем нерационально использовать инкрементальность алгоритма вместо использования эвристик.

Во-вторых, на низкой видимости в лабиринтоподобных картах (Labyrinth, brc504d) топологиях алгоритм D^* *lite* дает существенный прирост к производительности: практически отсутствуют выбросы, межквартильный размах меньше более чем в полтора раза (см. рис. 2); медиана при этом примерно такая же, но ощутимо меньше на более сложной карте – см.

```
def CalculateKey(s):
     return (min(g(s), rhs(s) + h(s), min(g(s), rhs(s))))
def Initialize():
     U = dict()
     for s in S:
          rhs(s) = g(s) = infinity
     rhs(start) = 0
     U.insert((start, (h(start), 0)))

def UpdateVertex(u):
     if u != start:
          rhs(u) =min([g(v) + c(v, u) for v in pred(u)])
     if u in U:
          U.remove(u)
     if g(u) != rhs(u):
          U.insert(u, CalculateKey(u))

def ComputeShortestPath():
     while U.topKey() < CalculateKey(goal) or rhs(goal) != g(
    goal):
          u = U.pop()
          if g(u) > rhs(u):
               g(u) = rhs(u)
               for s in succ(u):
                    UpdateVertex(s)
          else:
               g(u) = infinity
               for s in succ(u) + {u}:
                    UpdateVertex(s)

def Main():
     Initialize()
     while True:
          ComputeShortestPath()
          // make move along the shortest path to goal
          for (u, v) in observedEdges:
               UpdateVertex(u)
               UpdateVertex(v)

```

Листинг 1: D* lite



(a) Labyrinth (b) den401d (c) brc504d (d) NewYork

Рис. 1: Карты, использующиеся для измерения производительности

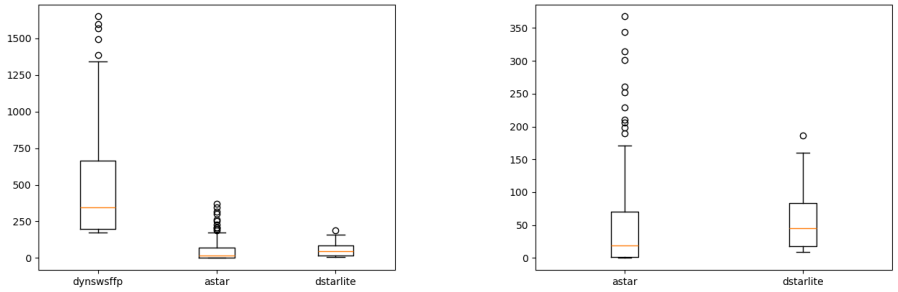


Рис. 2: den401d, малая видимость, производительность в ms

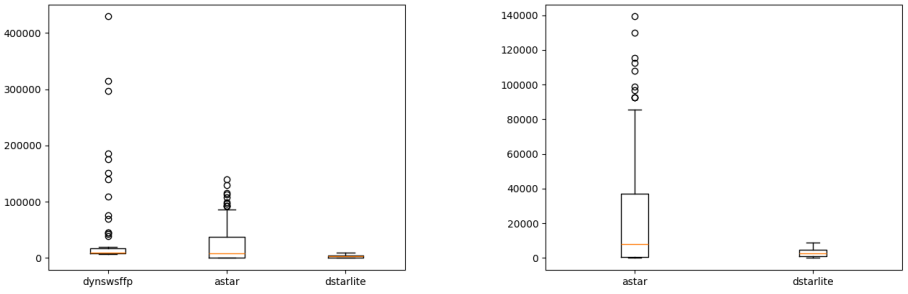


Рис. 3: Labyrinth, малая видимость, производительность в ms

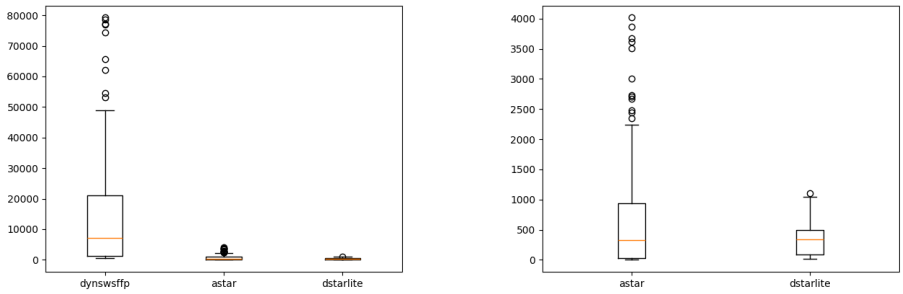


Рис. 4: brc504d, малая видимость, производительность в ms

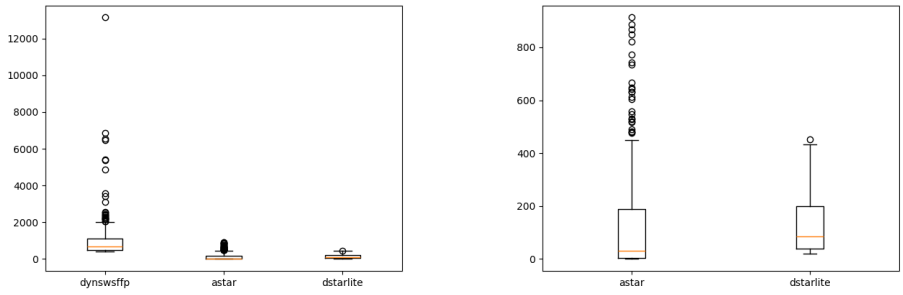


Рис. 5: NewYork_1_256, малая видимость, производительность в ms

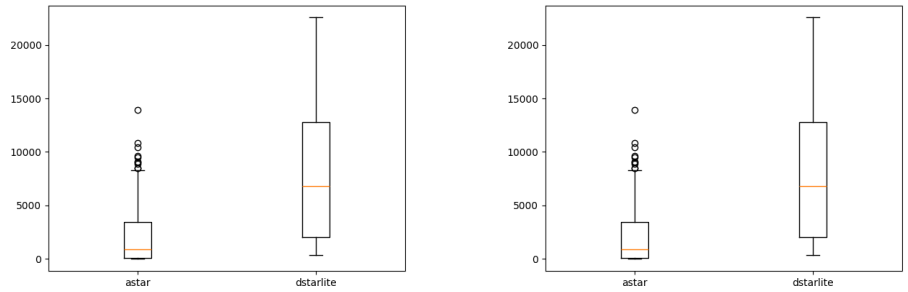


Рис. 6: Labyrinth, большая видимость (радиус 50), производительность в ms

3, 4). На других топологиях D^* *lite* дает немного большие затраты времени, но реальная производительность получается сравнимой с A^*, см. Рис. 5.

В-третьих, при большой видимости карты (иными словами, с увеличением радиуса видимости) алгоритм D^* *lite* дает уже не такие впечатляющие результаты, существенно уступая в производительности A^*, см. Рис. 6.

6 Заключение

Мы реализовали алгоритм D^* *lite*, включащий в себя инкрементальность Dynamic SWSF-FP и использование эвристик, подобно A^*. Также мы провели его экспериментальное исследование, сравнив с вышеупомянутыми алгоритмами. Алгоритм показал себя исключительно хорошо в ситуациях малой видимости и на сложных картах. В других же ситуациях алгоритм давал результаты, которые в среднем несколько хуже, чем результаты A^*.

Список литературы

[1] Nils J. Nilsson. Problem-solving methods in artificial intelligence. In *McGraw-Hill computer science series*, 1971.

[2] G. Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[3] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.