УДК 004.896

ОБ ОДНОМ ВОПРОСЕ РЕАЛИЗАЦИИ АЛГОРИТМА ПЛАНИРОВАНИЯ ТРАЕКТОРИИ A*

С.А. Дергачев (sadergachev@edu.hse.ru)
МИЭМ им. Тихонова НИУ ВШЭ, Москва

К.С. Яковлев (kyakovlev@hse.ru)
ФИЦ ИУ РАН, Москва
НИУ ВШЭ, Москва
МФТИ, Долгопрудный

Аннотация. В работе рассматривается задача планирования траектории агента (мобильного робота, беспилотного транспортного средства и др.) в статической среде. В мобильной робототехнике требуется получать решение этой задачи за ограниченное время, что может быть затруднительным в условиях использования малопроизводительного аппаратного обеспечения. В работе исследуется вопрос организации хранения результатов промежуточных вычислений при планировании траектории с помощью алгоритма A* для повышения его быстродействия. Предлагаются различные варианты реализации алгоритма, приводятся результаты экспериментального исследования.[1]

Ключевые слова: планирование траектории, A*, эвристический поиск.

Введение

Для повышения автономности беспилотных транспортных средств необходима разработка интеллектуальных систем управления, важным компонентом которых является модуль автоматического планирования траектории [Макаров и др., 2015]. Зачастую планирование траектории сводится к поиску пути на графе, вершины которого соответствуют некоторым положениям объекта в пространстве, а ребра – простейшим траекториям между ними. Для решения задачи в такой постановке зачастую используются эвристические алгоритмы семейства A* [Hart et al., 1968] такие как Theta* [Daniel et al., 2010], LIAN [Yakovlev, 2015] и др. Многие алгоритмы этого семейства обладают важными теоретическими

[1] Работа выполнена при частичной поддержке РФФИ (проекты №№ 17-07-00281, 18-37-20032)

свойствами (полнота и оптимальность), но на практике важна также скорость работы, особенно – в контексте применения на маломощных вычислителях, широко используемых в мобильной робототехнике, таких как, например, Raspbery Pi [Андрейчук и др., 2016]. Таким образом, актуальной является задача повышения быстродействия алгоритмов планирования за счет оптимизаций при их реализации. Именно этой проблеме и посвящена настоящая работа.

Для обеспечения корректного функционирования алгоритмам эвристического поиска семейства A* требуется хранить большой объем результатов промежуточных вычислений и обеспечивать быстрый поиск элементов, при этом индексация по фиксированному ключу невозможна, т.к. в различных ситуациях требуется поиск по различным характеристикам. Таким образом, для определения наиболее эффективного (с точки зрения быстродействия) способа организации хранения данных необходимо: а) программно реализовать несколько вариантов метода, совпадающих алгоритмически, но отличающихся используемыми структурами хранения данных (контейнерами); б) провести экспериментальное исследование полученных реализаций. Именно эти задачи и были решены в ходе данной работы.

1 Постановка задачи планирования

Рассмотрим задачу планирования траектории мобильного робота на плоскости как задачу поиска пути на графе особой структуры – графе регулярной декомпозиции, метрическом топологическом графе (ГРД/МТ-граф) [Яковлев и др., 2013]. Этот граф может быть представлен в виде матрицы меток «0» и «1» («свободно», «заблокировано») или таблицы, состоящей из заблокированных (непроходимых) и свободных (проходимых) ячеек квадратной формы. Упомянутая таблица получается наложением регулярной сетки на рабочую область робота и закрашиванием тех ячеек, в которых содержится хотя бы одна непроходимая для робота точка. Центры ячеек – вершины графа. Ребра соответствуют парам смежных проходимых вершин. Вес ребра – эвклидово расстояние между вершинами (центрами клеток). Задача состоит в поиске кратчайшего пути на таком графе.

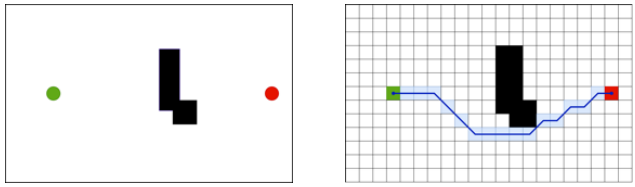

Рис. 1. Пример представления пространства в качестве графа особой структуры

2 Алгоритм A* и вопросы его реализации

Алгоритм A* является эвристической модификацией алгоритма Дейкстры [Dijkstra, 1959]. Для поиска пути осуществляется итеративный обход графа, при этом используется эвристика, влияющая на то, какую вершину обрабатывать на очередном шаге. Алгоритм выбирает для обработки вершину минимизирующую функцию $f(v) = g(v) + h(v)$, где $g(v)$ – это вес пути из начальной вершины в вершину v, известный к текущей итерации, а $h(v)$ – это оценка веса пути из v в целевую вершину. С алгоритмической точки зрения можно считать, что h-значения всех вершин известны (или вычисляются за константное время, которым можно пренебречь). g-значения, наоборот, изначально неизвестны для всех вершин, кроме начальной, а на уровне псевдокода полагают эти значения равными бесконечности.

После того как на текущей итерации выбрана вершина минизирующая $f(v)$, например v', происходит её раскрытие (expansion), в ходе которого производится перебор всех смежных с v' вершин – v'', и сравнение $g(v'')$ и $g(v') + w(v', v'')$, где w – вес соответствующего ребра. Если $g(v') + w(v', v'')$ меньше, чем $g(v'')$, то есть найден более короткий путь к v'', то, во-первых этот факт фиксируется, а именно $g(v'')$ полагается равным $g(v') + w(v', v'')$, во-вторых для вершины v'' запоминается ссылка на v', то есть теперь вершина v'' является родительской для вершины v''. С использованием этих ссылок происходит восстановление пути после окончания расчёта g-значений. Это окончание происходит, когда к раскрытию выбирается целевая вершина.

Псевдокод алгоритма A* представлен на Рис.2. Предполагается, что для организации обхода графа используются два контейнера – OPEN и CLOSE. OPEN – контейнер, содержащий вершины-кандидаты, из которых выбирается вершина для раскрытия на текущей итерации алгоритма. Изначально это список содержит лишь начальную вершину и пополняется за счет добавления в него вершин, смежных с раскрываемой. CLOSE – контейнер для раскрытых вершин. Заметим, что при использовании эвристики, удовлетворяющей условиям монотонности, не требуется перераскрытие вершин для нахождения кратчайшего пути [Pearl, 1984], таким образом CLOSE может быть использован так, как указано в строке 17 псевдокода: если вершина смежная с раскрываемой содержится в CLOSE, то перерасчёт её g-значения не требуется (строки 18-20 пропускаются).

Алгоритм 1 Алгоритм A^*

```
1: Open := {}
2: Close := {}
3: g(v_start) := 0
4: for v ∈ V do
5:     g(v) := inf
6:     parent(v) := NIL
7: end for
8: Open := Open ∪ {v_start}
9: while Open ≠ ∅ do
10:    v' = argmin_{v ∈ Open}(F(v))
11:    Open := Open\{v'}
12:    Close := Close ∪ {v'}
13:    if v' = v_goal then
14:        break
15:    end if
16:    for v'' ∈ successors(v') do
17:        if v'' ∉ Close and g(v'') > g(v') + w(v', v'') then
18:            g(v'') := g(v') + w(v', v'')
19:            parent(v'') := v'
20:            Open := Open ∪ {v''}
21:        end if
22:    end for
23: end while
24: if v_goal ∉ Close then
25:    print("Path not found")
26: else
27:    GetPathFromParents()
28: end if
```

Рис. 2. Псевдокод алгоритма A*

Указанный на Рис. 2 псевдокод, являясь алгоритмически корректным, опирается на ряд допущений, реализация которых на практике сопряжена с определенными вопросами. Во-первых, строки 4-7 неявно предполагают генерацию структур данных, необходимых для поиска и соответствующих *каждой* вершине графа. Назовём такую структуру Node. Очевидно, что хранение в памяти данных обо всех вершинах графа для конкретной задачи избыточно (часто в процессе поиска рассматривается лишь малый процент от всех вершин). Во-вторых, строка 10 подразумевает поиск Node с минимальным *f*-значением среди всех Node, принадлежащих OPEN. На практике это означает, что для быстрого выполнения этого шага необходима индексация OPEN по ключу *f*. С другой стороны, в строке 17 неявно подразумевается поиск в OPEN вершины, смежной с текущей раскрываемой. Для быстрого выполнения этого шага необходима индексация по идентификатору вершины. В рассматриваемом случае, когда речь идёт о поиске пути на MT-графе, этот идентификатор есть пара индексов вершины *i* и *j* (или их свертка до одного целого числа). Таким образом, на практике к контейнеру OPEN выдвигаются противоречивые требования – с одной стороны быстрый поиск по идентификатору, с другой – быстрый поиск минимума по *f*-значению. Псевдокод более приближенный к практической реализации, акцентирующий внимание на указанных вопросах представлен на Рис. 3.

Теперь в процессе выполнения граф поиска (множество структур Node, соответствующих всем вершинам исходного графа) не хранится целиком в оперативной памяти, и новые элементы пространства поиска, Node, генерируются по мере надобности. Каждый элемент инкапсулирует идентификатор вершины (координаты клетки MT-графа), её *h*-значение, *g*-значение, индексы вершины *i* и *j*, а также ссылку на родительскую вершину (parent). Операции поиска/добавления в OPEN и CLOSE теперь прописаны явно.

Алгоритм 3 Алгоритм *A** (модификация)

```
1: Open := new OpenStructure()
2: Close := new CloseStructure()
3: N_start := new Node(v_start)
4: N_goal := new Node(v_goal)
5: N_start.g := 0
6: Open.AddOrUpdate(N_start)
7: while Open.Size ≠ 0 do
8:     N' = Open.GetOptimal()
9:     Close.Add(N')
10:    if N' = N_goal then
11:        return GetPathFromParents()
12:    end if
13:    for N" in N'.GetSuccessors() do
14:        if not Close.Contain(N") then
15:            N".g := N'.g + w(N', N")
16:            N".parent() := N'
17:            Open.AddOrUpdate(N")
18:        end if
19:    end for
20: end while
21: return PathNotFound()
```

Рис. 3. Псевдокод, описывающий программную реализацию алгоритма A*

Таким образом, при практической реализации алгоритма A* (или *любого* другого алгоритма этого семейства) возникает важный вопрос – каким образом организовать хранение и упорядочивание элементов поиска (Node) в OPEN и CLOSE? От CLOSE требуется лишь быстрый поиск по идентификатору, то есть для программной реализации можно однозначно рекомендовать использование хэш-таблицы, вставка и поиск по которой осуществляются за $O(1)$. Требования же к контейнеру OPEN, как было сказано, противоречивы. С одной стороны необходим быстрый поиск минимума по f (строка 8), с другой – быстрый поиск по идентификатору для добавления/обновления (строка 17). Удовлетворить обеим требованиям одновременно невозможно, поэтому предлагается реализовать OPEN с использованием различных контейнеров данных и провести экспериментальное исследование.

Рассмотрим подробнее контейнеры, которые были выбраны для сравнения.

Линейный список (list) – структура, хранящая упорядоченный набор данных с последовательным доступом к элементам. Хранение вершин в списке было реализовано двумя способами: с сохранением дубликатов (т.е. вершин с совпадающими индексами *i* и *j*) и без сохранения (w/ duplicates и w/o duplicates соответственно). Хранение дубликатов не влияет на теоретические свойства алгоритма, но, очевидно, ускоряет добавление элемента (поиск по идентификатору вершины просто не производится). Для быстрого нахождения вершины с минимальным *f*-значением, элементы списка упорядочены по нему. При совпадении *f*-значений вершин дальнейшее упорядочивание производилось по g-значению, индексу *i*, индексу *j*. Таким образом, можно говорить о составном ключе (*f, g, i, j*), который однозначно задает линейный порядок на множестве элементов поиска. Аналогичная система упорядочивания вершин использовалась и для программных реализаций с другими контейнерами.

Упорядоченное множество (set) – структура, хранящая упорядоченный набор данных, которая исключает хранение эквивалентных элементов. Аналогично списку используется в вариантах w/ duplicates и w/o duplicates (за исключением элементов с полностью ссыпающим набором свойств (f, g, i, j), т.к. они будут эквивалентны).

Очередь с приоритетами (priority queue) – структура данных, поддерживающая операции добавления нового элемента и нахождения элемента с максимальным приоритетом. Внутреннее устройство этого контейнера не предусматривает доступ к элементам, за исключением элемента с наибольшим приоритетом, а значит невозможно произвести операцию поиска и удаления дубликатов вершин, таким образом очередь с приоритетами реализована только в варианте w/duplicates.

Динамический массив (vector), элементами которого являются экземпляры вышеописанных контейнеров (например: vector of lists w/ duplicates). Индексация массива осуществлялась по *i* индексам вершин. То есть каждый элемент массива хранит экземпляры определенного контейнера, в котором содержаться лишь вершины с одинаковым *i*. Такая индексация удобна при использовании MT-графов, т.к. значение *i* меняется от 0 до заранее известного предела (высоты карты в клетках). Благодаря такой организации, скорость доступа к вершинам по идентификатору существенно повышалась, но несколько замедлялся поиск элемента с наименьшим *f*.

Таким образом, для дальнейших исследований была подготовлена реализация алгоритма A*, с возможностью выбора одного из 10 вышеописанных контейнеров.

3 Экспериментальное исследование

Для проведения экспериментальных исследований алгоритм A* был реализован на языке C++ в соответствие с указанным псевдокодом (см. рис. 2). В качестве контейнеров OPEN были использованы реализации выбранных структур из библиотеки стандартных шаблонов (STL) C++. Эксперименты проводились на двух типах карт – синтетических (пустых, искусственно сгенерированных, с изменяющимся размером) и реальных (взятых из открытых коллекций данных или построенных по реальным картам городской местности). Исследование с использованием синтетических карт производилось на компьютере на базе Intel Core i7-3615QM @2.3 ГГц, 8 Гб ОЗУ под управлением Windows 10 (64-bit). Тестирование на реальных картах производилось на компьютере с Windows 7 (32-bit) на базе Intel Core 2 Duo Q8300 @2.5Ghz, 2 Гб ОЗУ. Исходный код реализации алгоритма доступен на https://github.com/PathPlanning/AStar-DCO.

3.1. Тестирование на синтетических картах

Для проведения экспериментальных исследований генерировались пустые карты разного размера с стартом и финишем расположенными в диагонально противоположенных углах карты. Размер карт менялся от 5000 x 5000[2] до 20 000 x 20 000 с шагом в 100 единиц. При подобном подходе количество итераций алгоритма, требуемых для выполнения задания, растет линейно. На каждой итерации при этом проверяется одно и тоже число смежных вершин, т.к. отсутствуют препятствия. Таким образом, такой подход позволяет линейно увеличивать сложность заданий, т.к. общее время выполнения зависит в основном от времени, которое требуется на работу с контейнером OPEN (добавление, поиск по идентификатору, поиски минимума по *f*-значению).

Одиночный эксперимент состоял в запуске алгоритма A* с различными реализациями контейнера OPEN для отдельного задания. Поскольку на время выполнения оказывают влияние процессы, запущенные на ЭВМ параллельно этой задаче, одно и тоже задание для каждого контейнера запускалось 5 раз, а потом бралось среднее арифметическое времени выполнения нескольких запусков. График, демонстрирующий зависимость этих времен от типа используемого контейнера OPEN приведен ниже (Рис. 4).

[2] Для карт с размером менее 5 000 x 5 000 время работы алгоритма было меньше 10^{-3} и не поддавалось точному измерению, поэтому такие задания не рассматривались.

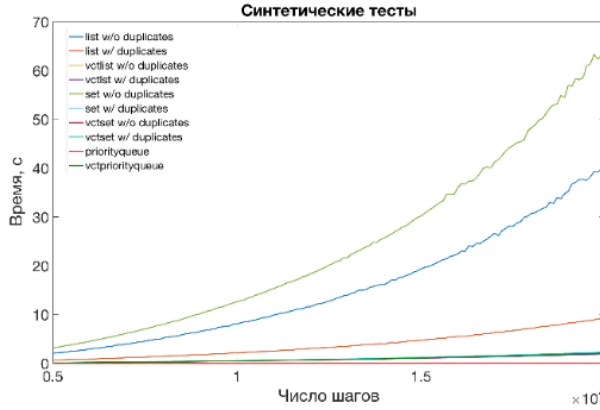

Рис. 4. График зависимости времени выполнения теста от количества шагов для всех контейнеров

Можно видеть, что наименьшее быстродействие показывают контейнеры: set без дубликатов, list без дубликатов и list с дубликатами. Для контейнеров с большим быстродействием был построен следующий график (Рис. 5). На нем видно, что priority queue и set с дубликатами лидируют по быстродействию Следующими по скорости являются векторы list с дубликатами и без дубликатов. Их результаты почти полностью совпадают, но заметно отстают от set с дубликатами и priority queue.

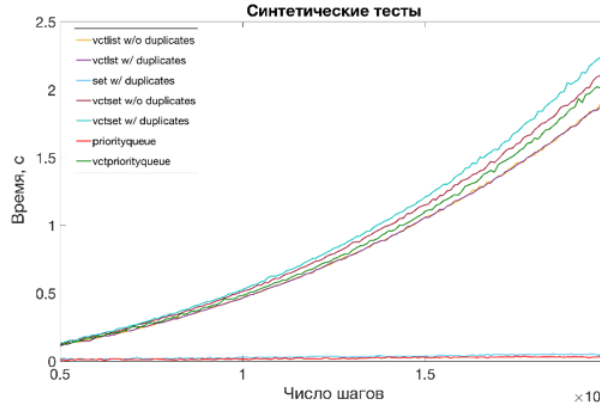

Рис. 5. График зависимости времени выполнения теста от количества шагов для 7 первых контейнеров

3.2 Тестирование на реальных картах

Во второй части эксперимента были использованы три набора из 36, 75 и 100 карт соответственно. Первые два набора – Warcraft и Baldur's Gate – состоят из карт размером 512 на 512 ячеек, использующихся в игровой индустрии [Sturtevant, 2012]. Для них существуют банк заданий, для каждого из которых известна длина кратчайшего пути. Задания для каждой карты были отсортированы по этой длине и было выбрано по 100 заданий с самыми длинными путями. Третий набор состоял из фрагментов карты Москвы, полученных из открытой гео-информационной базы данных OpenStreetMaps (www.openstreetmap.org) размером 501 на 501 ячейку. Меняя расположение ячеек начала и конца пути для каждого фрагмента было составлено по 100 различных заданий. Ячейки начала и конца выбирались таким образом, чтобы они располагались на противоположенных сторонах карты с отступом 10 ячеек от края. Стороны карты и неопределенные координаты выбирались случайным образом. Для каждого задания было проведено по 6 запусков реализации алгоритма и было взято среднее значение времени выполнения.

При анализе результатов было обнаружено, что использование различных контейнеров по-разному влияет на быстродействие алгоритма в зависимости от числа итераций основного цикла. Так, усредняя по всем заданиям получается график, представленный на рис. 7, а при усреднении по заданиям, для решения которых требуется 4 000 шагов и менее, получается результаты представлены на рис. 6. Основное отличие в том, что для таких заданий варьируется время работы для «средних» контейнеров (т.е. не самых быстрых и не самых медленных) в зависимости от набора карт. При усреднении по всем заданиям картина более однородная.

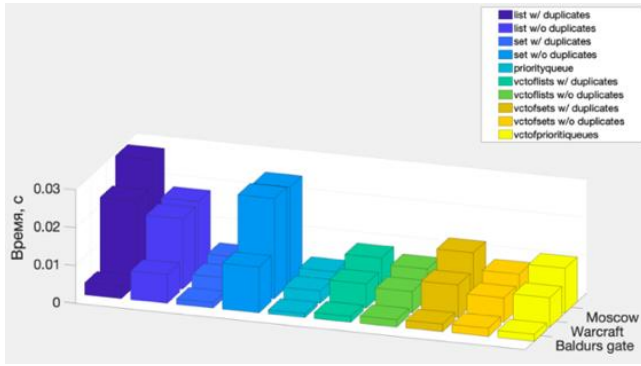

Рис. 6. Среднее время работа на заданиях с малым (<4 000) числом шагов.

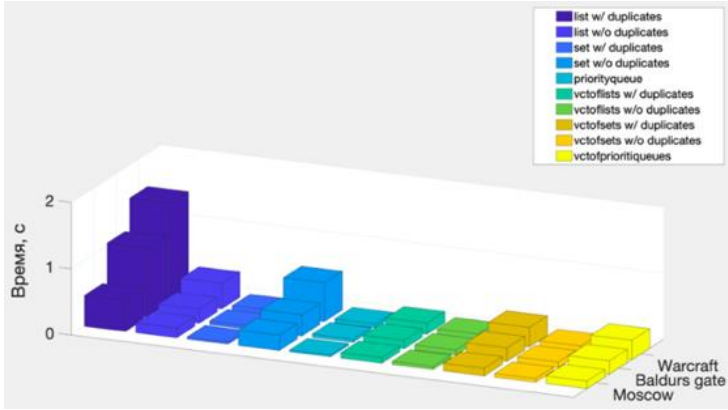

Рис. 7. Время работы, усредненное по всем заданиям.

3.3 Выводы

По результатам экспериментального исследования можно сделать следующие выводы. Во-первых, использование различных контейнеров существенно влияет на производительность (скорость работы) алгоритма. Таким образом, очень важно с практической точки зрения использовать подходящие контейнеры данных при реализации алгоритма планирования. Во-вторых, контейнеры с дубликатами всегда быстрее, чем контейнеры без них. Однако, очевидно, что их использование ведёт к дополнительным затратам памяти. Поэтому в приложениях, где критична как скорость работы, так и затраты памяти, рекомендуется использовать вектор списков или вектор множеств.

Заключение

В работе был рассмотрен ряд вопросов, касающихся практической реализации алгоритмов эвристического поиска семейства A*, обычно использующихся в мобильной робототехнике для планирования траектории. Основное внимание было уделено вопросу выбора программного контейнера для хранения результатов промежуточных вычислений. Была продемонстрирована критическая зависимость быстродействия алгоритма от выбранного способа хранения (контейнера). В результате проведения масштабных экспериментальных исследований различных контейнеров даны рекомендации по их использованию для создания высокоэффективных (в смысле скорости работы) реализация алгоритмов планирования.

Список литературы

[Андрейчук и др., 2016] Андрейчук А. А. Боковой А. В. Яковлев К. С. Оценка быстродействия некоторых алгоритмов планирования траектории на широко используемой в робототехнике платформе Raspberry PI // Экстремальная робототехника – 2016 – №1 – С. 184-189.

[Макаров и др., 2015] Макаров Д.А., Панов А.И., Яковлев К.С. Архитектура многоуровневой интеллектуальной системы управления беспилотными летательными аппаратами // Искусственный интеллект и принятие решений, 3, 2015. С.18-32.

[Яковлев и др., 2013] Яковлев К.С., Баскин Е.С. Графовые модели в задаче планирования траектории на плоскости // искусственный интеллект и принятие решений, 1, 2013. С.5-12.

[Daniel et al., 2010] Daniel, K., Nash, A., Koenig, S., and Felner, A. Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research. 2010. No 39, P.533-579.

[Dijkstra, 1959] Dijkstra, E.W. A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 1959. Pp 269–271.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics. 1968. No 4(2), P.100-107.

[Pearl, 1984] Pearl J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984

[Sturtevant, 2012] Sturtevant N. R. Benchmarks for grid-based pathfinding //IEEE Transactions on Computational Intelligence and AI in Games. – 2012. – V. 4. – №. 2. – pp. 144-148.

[Yakovlev, 2015] Konstantin Yakovlev, Egor Baskin, and Ivan Hramoin. Grid-based angle-constrained path planning. In Proceedings of The 38th Annual German Conference on Artificial Intelligence (KI'2015), pages 208-221, 2015. Springer International Publishing.