Сравнение эффективности алгоритмов поиска на невзвешенных графах на примере поиска пути на карте с препятствиями

Виногродский Серафим

5 июня 2022 г.

Содержание

1 Введение

Поиск пути между двумя вершинами некоторого графа — это задача, которая очень часто встречается в различных сферах приложения информатики. В связи с этим она имеет множество вариантов решения, каждое из которых имеет свои преимущества и недостатки. В этой работе мы рассмотрим три наиболее распространённых и простых из этих решений:

- поиск в глубину,
- поиск в ширину,
- поиск с двух сторон.

Помимо непосредственного описания принципов работы каждого из этих алгоритмов и их реализации на языке C++, мы так же проанализируем эффективность каждого из них. Делать мы это будем как с теоретической точки зрения, используя асимптотическую оценку для времени выполнения и потребления памяти, так и с практической — сравним скорость выполнения каждого из алгоритмов на примере поиска пути на карте с препятствиями.

2 Описание реализованных алгоритмов поиска

2.1 Поиск в глубину (Depth-first search, DFS)

2.1.1 Описание алгоритма

В основе реализации алгоритма поиска в ширину лежит использование стека для определения очерёдности посещения элементов графа. Для его реализации на C++ требуется 3 вспомогательные структуры:

(1) Множество посещённых элементов графа.

(2) Стек, определяющий очерёдность посещения.

(3) Словарь родителей.

```
auto visited      = set<cell_t>({_start_pos});    // (1)
auto search_stack = stack<cell_t>({_start_pos}); // (2)
auto parents      = map<cell_t, cell_t>();        // (3)
```

В общем случае структура parents является необязательной, поскольку она используется только для восстановления пути до найденного элемента, что не всегда необходимо. Поскольку в данной работе алгоритм DFS используется

именно для поиска пути, а не для нахождения элемента, избавиться от структуры parents невозможно.

Далее, пока стек search_stack не пуст, из него изымается верхний элемент next. Если next подходит под условия поиска, то из словаря родителей строится путь до найденного элемента, после чего этот путь возвращается как результат поиска:

```
while (!search_stack.empty()) {
    cell_t next = search_stack.top();
    if (next == _dest_pos) {
        return build_path(parents, _start_pos, _dest_pos);
    }
    searh_stack.pop();

    // <...>
}
```

В ином случае, то есть если искомая вершина ещё не достигнута, все до этого не посещённые дети вершины next поочерёдно добавляются в стек search_stack, после чего поиск продолжается уже для следующей вершины (если таковая имеется):

```
while (!search_stack.empty()) {
    // <...>

    for (cell_t child : neighbors(next)) {
        if (!visited.contains(child)) {
            parents.insert({child, next});
            visited.insert(child);
            search_stack.push(child);
        }
    }
}
```

Следствием использования стека для определения очерёдности посещения является то, что дети только что посещённых узлов обрабатываются раньше детей узлов, посещённых до этого. Таким образом алгоритм DFS как бы старается забраться как можно глубже в структуру дерева (что вполне соответствует названию алгоритма), вместо того, что бы последовательно обрабатывать все узлы все большей и большей глубины, как это делает алгоритм BSF, который будет описан далее.

Стоит так же отметить, что приведённая здесь реализация может быть оптимизирована путём использования одной только структуры parents вместо пары parents и visited, но подобный подход, пусть и не значительно, но уложил бы логику реализации алгоритма, требуя отдельной обработки определённых частных случаев. Поскольку данная работа фокусируется на сравнительной характеристике алгоритмов, а не на предельной их оптимизации, в ней подобная

реализация подробно не рассматривается.

2.1.2 Оценка сложности

Пусть в связном графе, по которому производится поиск, содержится V вершин и E рёбер. В худшем случае алгоритм DFS обходит все вершины и рёбра графа. Время обработки каждой из вершин составляет $O(1)$, поскольку оно включает в себя лишь работу с очередью и проверку на равенство вершины искомой вершине. Время обработки каждого из рёбер составляет $O(\log V)$, поскольку оно включает в себя работу со словарями и множествами из не более чем V элементов. Таким образом, верхняя оценка времени выполнения алгоритма составляет $O(V + E \cdot \log V)$.

Каждая вершина попадает в $\mathrm{visited}$, $\mathrm{parents}$ и stack не более одного раза, так что верхняя оценка потребления памяти составляет $O(V)$.

2.2 Поиск в ширину (Breadth-first search, BFS)

2.2.1 Описание алгоритма

В общем случае, реализация алгоритма поиска в ширину полностью идентична таковому для поиска в глубину. Единственное отличие состоит в том, что для определения очерёдности посещения вместо стека используется очередь. Как следствие, алгоритм BSF гарантирует, что все вершины заданной глубины будут посещены раньше всех вершин большей глубины, так что в процессе поиска алгоритм BFS, в отличие от DSF, как бы разрастается вширь, постепенно покрывая все большую и большую окрестность вершины, из которой начинается поиск (отсюда и название.)

Фактическая реализация этого алгоритма в данном проекте, однако, в значительной степени отличается от алгоритма DFS. Вместо прямой реализации аналогичной той, что была приведения выше для DFS, алгоритм опирается на использование функции, реализующей обобщённый шаг поиска в ширину, используемый в дальнейшем алгоритмом поиска в ширину с двух сторон. Этот подход будет более подробно описан в части 2.3.3.

2.2.2 Оценка сложности

Очереди асимптотически полностью аналогичны стекам, так что верхняя оценка и времени выполнения, и потребления памяти алгоритмом BFS совпадает с оценками для DFS (см. часть 2.1.2):

- время выполнения — $O(V + E \cdot \log V)$,
- потребление памяти — $O(V)$.

2.3 Поиск в ширину с двух сторон (Bidirectional search)

2.3.1 Описание алгоритма

Суть алгоритма поиска с двух сторон, как нетрудно следует из названия, состоит в том, что поиск пути начинается сразу с двух сторон — и сточки начала, и с точки конца пути. Окончание поиска происходит в тот момент, когда два поиска пересекаются в некоторой вершине, через которую, соответственно, и будет проходить искомый путь.

Реализация алгоритма двустороннего поиска в целом аналогична реализации алгоритма BFS. Сначала инициализируются используемые структуры — отдельно для поиска в прямом направлении и отдельно для поиска в обратном направлении:

```
set<cell_t>          visited_forward({_start_pos});
queue<cell_t>        forward_search_queue({_start_pos});
map<cell_t, cell_t> parents;

set<cell_t>          visited_backward({_dest_pos});
queue<cell_t>        backward_search_queue({_dest_pos});
map<cell_t, cell_t> children;
```

Так же создаются две вспомогательные переменные:

(1) Переменная-флаг, отвечающая за окончание поиска.

(2) Переменная, куда будет сохранена вершина — пересечение поисков.

```
bool is_done = false; // (1)
cell_t intersection;  // (2)
```

Далее, пока и очередь прямого поиска, и очередь обратного поиска не пусты, выполняется по одному шагу поиска в каждом направлении:

```
while (!forward_search_queue.empty() &&
       !backward_search_queue.empty()) {
    bfs_step(
        &visited_forward, &visited_backward,
        &forward_search_queue, &parents,
        &is_done, &intersection
    );
    bfs_step(
        &visited_backward, &visited_forward,
        &backward_search_queue, &children,
        &is_done, &intersection
    );

    // <...>
}
```

В случае, если какой-то из поисков сигнализирует об завершении поиска, путём установки флагу is_done значения true, строятся и соединятся две части найденного пути, отвечающие каждому из направлений поиска:

```
while ( <...> ) {
    // <...>

    if (is_done) {
        vector<cell_t> res_path =
            build_path(parents, _start_pos, intersection);
        res_path.pop_back();
        res_path +=
            build_reversed_path(
                children, _dest_pos, intersection
            );
        return res_path;
    }
}
```

Если же очереди окажутся пустыми до окончания поиска, это будет означать, что пути межу двумя вершинами графа не существует, так что в качестве результата возвращается пустой путь:

```
return {};
```

Значительная часть логики алгоритма скрыта в функции bfs_step, инкапсулирующей один шаг поиска в ширину (см. части 2.1.1 и 2.2.1):

```
void bfs_step( <...> ) {
    if (search_queue->empty() || *is_done) return;
    cell_t next = search_queue->front();
    if (visited_backward->contains(next)) {
        *is_done = true;
        if (intersection) *intersection = next;
        return;
    }
    search_queue->pop();

    for (cell_t child : neighbors(next)) {
        if (!visited_forward->contains(child)) {
            parents->insert({child, next});
            visited_forward->insert(child);
            search_queue->push(child);
        }
    }
}
```

2.3.2 Оценка сложности

Верхняя оценка для времени работы и потребления памяти в худшем случае для алгоритма двустороннего поиска совпадает с оценками для DFS и BFS:

- время выполнения — $O(V + E \cdot \log V)$,
- потребление памяти — $O(V)$.

Это связано с тем, что в худшем случае алгоритму так же приходится обходить все ребра и вершины графа, совершая над ними те же операции, что и для алгоритма BFS.

Однако, для среднего случая можно получить более благоприятную оценку времени выполнения, исходя из немного других соображений. Пусть b — средний коэффициент связности рассматриваемого графа, d — длина искомого пути. Тогда, из комбинаторных соображений, обработка алгоритмом BFS всех вершин каждого следующего уровня глубины в среднем составляет $O(b^d) \cdot O(\log b^d)$ (где $O(\log b^d)$ — время обработки каждой отдельной вершины), а итоговая оценка для всего алгоритма BFS составляет

$$O\left(\sum_{k=1}^{d} b^k \cdot \log b^k\right) = O(b^d \cdot \log b^d) = O(b^d \cdot d \log b).$$

Алгоритм же двустороннего поиска состоит из двух одновременных BFS, каждый из которых в среднем случае отвечает половине искомого пути и потому опускается на глубину примерно равную $d/2$. Таким образом общая оценка времени выполнения для алгоритма двустороннего поиска составляет

$$O(2 \cdot b^{\frac{d}{2}} \cdot \frac{d}{2} \log b) = O(b^{\frac{d}{2}} \cdot d \log b),$$

что в значительной степени лучше средней оценки для алгоритма BFS.

Отметим так же, что подобная оценка для алгоритма DFS совпадает с таковой для BFS и потому равна $O(b^d \cdot \log b^d)$.

2.3.3 Реализация алгоритма поиска в ширину

Алгоритм BFS можно реализовать, используя функцию bfs_step. Для этого используются те же основные структуры:

```
set<cell_t>          visited({_start_pos});
queue<cell_t>        search_queue({_start_pos});
map<cell_t, cell_t> parents;
```

Так же для такой реализации потребуются две вспомогательные переменные:

(1) Переменная-флаг, отвечающий за окончание поиска (см. часть 2.3.1).

(2) Множество, хранящее в себе только вершину _dest_pos, т.е. конец искомого пути. Это множество будет передаваться в bfs_step в качестве аргумента visited_backwards. Поиск останавливается, как только множества visited_forward и visited_backward пересекаются хотя бы по одному элементу, так что таким образом поиск будет остановлен, как только алгоритм поиска доберётся до _dest_pos.

```
bool                 is_done = false;                  // (1)
const set<cell_t> destination_set({_dest_pos}); // (2)
```

Далее, пока очередь search_queue непуста, выполняются шаги поиска:

```
while (!search_queue.empty()) {
    bfs_step(
        &visited, &destination_set,
        &search_queue, &parents,
        &is_done, nullptr
    );

    // <...>
}
```

В случае, если функция bfs_step сигнализирует об окончании поиска, устанавливая флагу is_done значение true, поиск завершается и найденный путь возвращается в качестве результата:

```
while (!search_queue.empty()) {
    // <...>

    if (is_done) {
        return build_path(parents, _start_pos, _dest_pos);
    }
}
```

Если же в какой-то момент очередь окажется пуста, это будет означать, что искомого пути не существует, так что в качестве результата вернётся пустой путь.

```
return {};
```

3 Сравнение эффективности

В части 2.3.2 мы выяснили, что, несмотря на то, что в худшем случае алгоритм поиска с двух сторон имеет асимптотику равную асимптотике алгоритма BFS,

в среднем случае двусторонний поиск является более эффективным. Перейдём же к основной цели данной работы — проверим так ли это.

Для этого будем многократно запускать поиск пути на карте фиксированного размера, каждый раз произвольным образом заполняя её проходимыми и непроходимыми клетками в соотношении $1 : 3$ и так же произвольно выбирая точки начала и конца пути. После n-го числа забегов выводится средний результат времени выполнения для каждого из алгоритмов.

Результаты тестов подтверждают теоретические предсказания. Так, для карты 60×120 и теста на 10000 забегах, вывод программы выглядит следующим образом:

```
Averate time:
   BFS   : 1638µs
   DFS   : 1774µs
  biBFS  : 1177µs
```

Отсюда получаем, что в среднем случае алгоритм двустороннего поиска оказывается примерно в 1.4 раза быстрее одностороннего алгоритма поиска в ширину, который, в свою очередь, по скорости выполнения сопоставим с алгоритмом DFS.

Аналогично, для карты 200×200 получаем следующее результаты:

```
Averate time:
   BFS   : 11468µs
   DFS   : 10319µs
  biBFS  : 7249µs
```

Разрыв по скорости выполнения получается даже больше — поиск в двух сторон работает примерно в 1.6 раза быстрее.

4 Выводы

Таким образом, из трёх проанализированных в этой работе алгоритмов наиболее эффективным, как и предсказали теоретические рассуждения, оказался алгоритм поиска с двух сторон. Не стоит, однако, думать, что двусторонний поиск всегда лучше одностороннего, поскольку у него есть и значительный недостаток: для его применимости требуется, чтобы точка конца пути была изначально известной, что накладывает значительные ограничения на зону его применимости.

Полный код код проекта доступен на GitHub[1].

[1] https://github.com/fimmind/maze_solver/blob/master/main.cpp