

Г. О. Евстропов,

Национальный исследовательский университет «Высшая школа экономики», г. Москва

РАЗРАБОТКА ТЕСТОВЫХ ПРИМЕРОВ ДЛЯ АВТОМАТИЧЕСКОЙ ПРОВЕРКИ РЕШЕНИЙ ЗАДАЧ В УЧЕБНЫХ КУРСАХ И СОРЕВНОВАНИЯХ ПО ПРОГРАММИРОВАНИЮ*

Аннотация

При автоматическом тестировании программ, решающих различные задачи теории потоков, использование тестовых примеров, построенных почти полностью случайно, не позволяет корректно оценить асимптотическую сложность реализованного алгоритма. Время работы большинства алгоритмов поиска потока на таких тестах будет существенно ниже верхней асимптотической оценки, а для продвинутых алгоритмов и вовсе будет расти линейно или почти линейно. В данной работе рассматривается построение тестовых примеров, на которых время работы наиболее распространенной реализации алгоритма Форда—Фалкерсона будет экспоненциально зависеть от размера теста.

Ключевые слова: теория потоков, стоимостные потоки, автоматическое тестирование, соревнования по программированию.

Введение

Теория потоков предлагает эффективные алгоритмы решения различных задач комбинаторной оптимизации, которые могут быть сведены к поиску максимального потока в графе, минимального разреза или потока минимальной стоимости. Задачи такого рода возникают при оптимизации в транспортных сетях, при организации производства и в других областях.

Теория потоков изучается в различных учебных заведениях в курсах алгоритмов и комбинаторной оптимизации. При этом, как правило, частью такого курса является набор задач в системах с автоматической проверкой решений. Использование подобных систем позволяет существенно уменьшить затраты времени преподавателя на проверку корректности выполнения обучающимися задания, повысить качество этой проверки. Также в последние несколько лет задачи на теорию потоков получают все более широкое

распространение на школьных и студенческих соревнованиях по информатике и программированию.

Вышесказанное делает важным вопрос качественной подготовки тестовых примеров. Ввиду того что построение тестов, на которых алгоритмы поиска потока будут достигать своих верхних асимптотических оценок, является задачей не менее сложной, чем разработка этих алгоритмов, при подготовке задач для систем с автоматической проверкой решений тестовые наборы далеко не всегда содержат примеры такого рода, зачастую они состоят только из графов, сгенерированных практически случайно, и некоторых популярных конструкций.

В данной статье приводится принцип построения тестовых примеров, на практике демонстрирующих, что алгоритм Форда—Фалкерсона поиска максимального потока в сети работает за время, экспоненциально зависящее от размера входных данных. Сначала рассматривается пример, позволяющий добиться

* Подготовка данной статьи финансировалась в рамках программы государственной поддержки ведущих университетов Российской Федерации «5–100».

Контактная информация

Евстропов Глеб Олегович, стажер-исследователь лаборатории теоретической информатики факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики», г. Москва; *адрес:* 125319, г. Москва, Кочновский пр., д. 3; *телефон:* (495) 772-95-90, доб. 22913; *e-mail:* glebshp@yandex.ru, gevstropov@hse.ru

G. O. Evstropov,

National Research University High School of Economics

CREATING TEST EXAMPLES FOR AUTOMATIC CHECKING THE SOLUTIONS OF TASKS IN TRAINING COURSES AND COMPETITIONS IN PROGRAMMING

Abstract

We consider automated testing (without manual code inspection) for problems that require implementation of flow algorithms. If only graphs that were generated almost at random are used, it is impossible to estimate whether the implemented algorithm has right complexity or not. For almost all algorithms real running time on such test cases will grow significantly below theoretical bounds, while for advanced algorithms it might even grow quasilinear. In this work we consider how to build a test cases that will force popular implementation of Ford—Fulkerson algorithm match the expected complexity with running time depending exponentially from the size of the test.

Keywords: flow theory, cost flows, automated testing, programming competitions.

такого результата для наиболее часто встречающейся учебной реализации алгоритма Форда—Фалкерсона. Далее приводится улучшение этого примера, которое позволяет сохранить экспоненциальный рост времени работы алгоритма даже в случае, если исходящие из каждой вершины ребра рассматриваются не в фиксированном, а в случайном порядке.

Задача о максимальном потоке и алгоритмы ее решения

Сформулируем задачу о максимальном потоке.

Пусть дана сеть (G, c, s, t) , где G — орграф с множеством вершин $V = V(G)$ и множеством ребер $E = E(G)$. Обозначим количество вершин в графе $n = |V(G)|$, а количество ребер $m = |E(G)|$. На множестве упорядоченных пар вершин определена неотрицательная вещественнозначная функция пропускных способностей $c: V \times V \rightarrow R$, при этом $(u, v) > 0$, только если $(u, v) \in E(G)$. Из множества вершин выбраны источник s и сток t . Поток в G называется функция $f: V \times V \rightarrow R$, удовлетворяющая следующим трем условиям:

1) *ограничение пропускной способности*: $f(u, v) \leq c(u, v)$ для всех u, v , принадлежащих V ;

2) *антисимметричность*: $f(u, v) = -f(v, u)$ для всех $u, v \in V$;

3) *сохранение потока*: для всех $u \in V \setminus \{s, t\}$

$$\sum_{v \in V} f(u, v) = 0.$$

Величиной потока называется суммарный поток, выходящий из источника, т. е.:

$$|f| = \sum_{v \in V} f(s, v).$$

В классической формулировке задачи о максимальном потоке ставится задача отыскания потока $f(u, v)$ максимальной величины в сети (G, c, s, t) .

Первый комбинаторный алгоритм нахождения максимального потока был опубликован в 1956 году Л. Р. Фордом и Д. Р. Фалкерсоном [5]. Более сильные результаты были опубликованы в 1970 году советским математиком Е. Диницем [3] и независимо в 1972 году Д. Эдмондсом и Р. Карпом [4]. Оба результата представляли комбинаторный сильно полиномиальный алгоритм поиска максимального потока, основанный на насыщении кратчайшего пути. Алгоритм Эдмондса—Карпа имеет сложность $O(m^2n)$, а алгоритм Диница, совершенствуя эту технику, достигает сложности $O(n^2m)$. Наилучший сильно полиномиальный теоретический результат для общего случая задачи о максимальном потоке принадлежит Д. Орлину [6], работа была опубликована в 2012 году, и в ней описан алгоритм нахождения максимального потока, имеющий сложность $O(nm)$.

Тестовый пример для алгоритма Форда—Фалкерсона

Рассмотрим для начала известный пример, использующийся для демонстрации неполиномиальности алгоритма Форда—Фалкерсона поиска максимального потока [1] (рис. 1).

Действительно, если на каждом шаге выбирать для насыщения путь, использующий центральное ребро,

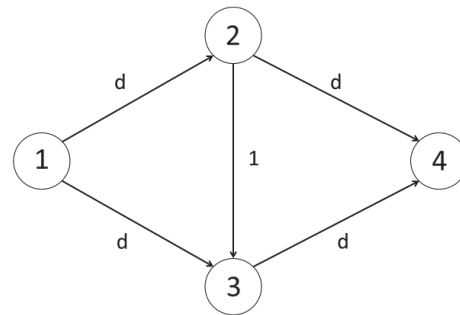


Рис. 1. Классический пример для алгоритма Форда—Фалкерсона

то алгоритм завершится только спустя $2d$ итераций. Поскольку для записи во входных данных числа d требуется только $\log_2 d$ бит, то время работы алгоритма экспоненциально растет при увеличении входных данных. Однако такое обоснование сильно опирается на произвольность выбора пути между истоком и стоком, каждый раз выбирая наихудший возможный путь.

Приведем листинг программы на языке C++ (рис. 2), содержащей естественную и простую реализацию алгоритма Форда—Фалкерсона. Мы постарались не использовать никаких особенностей языка C++, чтобы код был понятен и тем, кто с данным языком не знаком.

При использовании в качестве входных данных теста, приведенного на рисунке 1, указанная на рисунке 2 реализация алгоритма Форда—Фалкерсона найдет максимальный поток, сделав только три итерации основного цикла. На первой итерации будет насыщен путь $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, затем путь $1 \rightarrow 2 \rightarrow 4$, и, наконец, насыщение пути $1 \rightarrow 3 \rightarrow 4$ завершит поиск максимального потока. Таким образом, данный тест не демонстрирует неполиномиальность с практической точки зрения даже для наиболее естественной реализации алгоритма Форда—Фалкерсона, не содержащей никаких эвристических ускорений.

Вопрос завершения работы алгоритма Форда—Фалкерсона при реализации обхода в глубину с использованием фиксированного порядка рассмотрения ребер в каждой вершине анализируется в работе Б. Дина, М. Гозманса и Н. Имморлица [2]. Опишем здесь подробно процесс построения серии примеров, на которых указанный выше код сделает экспоненциально растущее от количества вершин число шагов. Определим рекурсивно множество графов G_k для всех целых неотрицательных значений k .

1. G_0 будет состоять из истока s_0 и стока t_0 , соединенных ребром пропускной способности 1. Вершина s_0 имеет номер 1, а вершина t_0 — номер 2.

2. Граф G_k получается добавлением к графу G_{k-1} новых вершин s_k и t_k , которые получают номера $2k+1$ и $2k+2$ соответственно. Из вершины s_k проводятся ребра в s_{k-1} и t_{k-1} пропускной способности 2^{k-1} . Ребра такой же пропускной способности проводятся из s_{k-1} и t_{k-1} в t_k .

Приведем рисунки, показывающие графы G_0 и G_1 (рис. 3, 4).

В графе G_1 размер максимального потока равен 2. При этом сначала алгоритм насытит путь $3 \rightarrow 1 \rightarrow 2 \rightarrow 4$, а затем — путь $3 \rightarrow 2 \rightarrow 1 \rightarrow 4$. Таким образом, ребро из 1 в 2 выступит в роли «переключа-

```

// Функция обхода в глубину принимает два
// параметра, v - номер текущей вершины
// и can_push - минимальную остаточную пропускную
// способность ребра вдоль пути,
// который привел обход в глубину в вершину v.
int dfs(int v, int can_push) {
    // Если обход графа дошел до стока, то насытить
    // путь, увеличив поток вдоль него на величину
    // can_push.
    if (v == sink) {
        return can_push;
    }
    // Пометить вершину как посещенную.
    visited[v] = true;
    // Перебираем возможную следующую вершину пути
    // до стока.
    for (int u = 1; u <= n; u = u + 1) {
        // Не рассматриваем уже посещенные вершины.
        if (visited[u] == true)
            continue;
        // Не рассматриваем насыщенные ребра.
        if (flow[v][u] == capacity[v][u])
            continue;
        // Величина can_push во вложенном вызове
        // не должна превосходить остаточной
        // пропускной способности ребра из v в u.
        int f = dfs(u, min(can_push,
            capacity[v][u] - flow[v][u]));
        // Если f > 0, то процедура обхода в глубину
        // нашла путь до стока из данной вершины.
        // Увеличим поток вдоль ребра из v в u
        // и уменьшим поток вдоль обратного ребра.
        if (f > 0) {
            flow[v][u] = flow[v][u] + f;
            flow[u][v] = flow[u][v] - f;
            return f;
        }
    }
    // Если выполнение процедуры дошло до этого
    // момента, то найти пути из вершины v до стока
    // не удалось. Вернем 0.
    return 0;
}

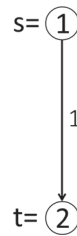
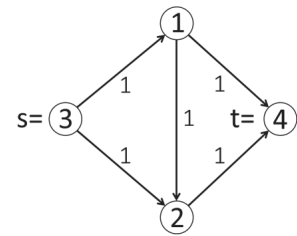
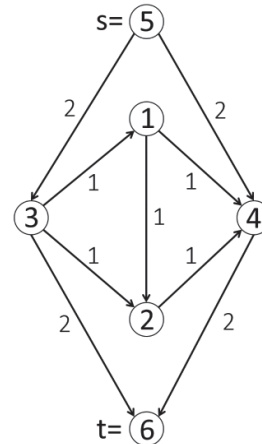
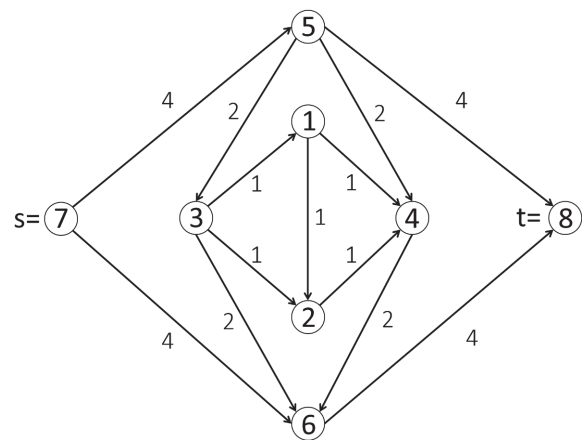
// source - стартовая вершина графа,
// sink - конечная.
int FordFulkerson() {
    int maxflow = 0, last = 0;
    do {
        // Убираем из всех вершин пометки о посещении.
        for (int i = 1; i <= n; i = i + 1)
            visited[i] = false;
        // Ищем увеличивающий поток путь
        // из source в sink
        last = dfs(source);
        maxflow = maxflow + last;
        // Если увеличить поток не удалось,
        // то проверка ниже завершит выполнение
        // основного цикла алгоритма
    } while (last > 0);
    return maxflow;
}

```

Рис. 2. Листинг программы

теля», два раза пропустив поток в разные стороны, и вернется в исходное состояние. При построении графа G_k в роли такого «переключателя» выступает уже весь граф G_{k-1} . Для лучшего понимания устройства конструкции приведем также рисунки графов G_2 и G_3 (рис. 5, 6).

Величина максимального потока в графе G_k равна 2^k , при этом каждая итерация указанной выше реали-

Рис. 3. Граф G_0 Рис. 4. Граф G_1 Рис. 5. Граф G_2 Рис. 6. Граф G_3

зации алгоритма Форда—Фалкерсона будет находить путь с остаточной пропускной способностью ровно 1. Количество вершин в графе G_k равняется $2k + 2$. Таким образом, время работы алгоритма на тестах такой структуры есть $O(2^{\frac{n}{2}}n)$. Однако данный тест сильно опирается на тот факт, что нам известно, в каком именно порядке будут просматриваться исходящие из каждой вершины ребра. Соответственно, при внесении небольших изменений в код реализации алгоритма приведенный тест утратит свою эффективность.

Будем теперь считать, что при каждом входе в вершину выбирается некоторая случайная перестановка инцидентных ей ребер, после чего они просматриваются в соответствующем этой перестановке порядке. Мы бы хотели модифицировать описанную конструкцию графа, чтобы «заставить» алгоритм в каждом конкретном случае выбрать нужный нам путь с достаточно высокой вероятностью.

Пусть в графе G_k в вершину v входили ребра из вершин a и b , а выходили ребра в вершины c и d (рис. 7).

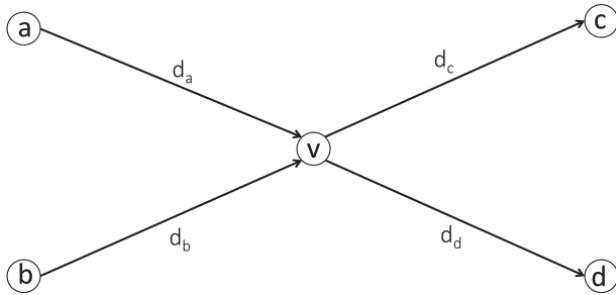


Рис. 7. Фрагмент графа G_k до преобразования

Зафиксируем некоторое значение p , после чего применим следующее преобразование к каждой вершине исходного графа G_k :

1. Создадим новые вершины $c', v_a, v_1, v_2, \dots, v_{2p}$.
2. Удалим из графа ребра (a, v) , (v, c) и (v, d) .
3. Проведем из a в v_a ребро пропускной способности d_a , из v_a в v — ребро бесконечной пропускной способности.
4. Последовательно соединим вершины $v, v_1, v_2, \dots, v_{2p}$ ребрами бесконечно большой пропускной способности. Из вершины v_{2p} проведем в d ребро пропускной способности d_d .
5. Из вершин $v_1, v_2, \dots, v_p$ проведем ребра в вершину v_a с бесконечными пропускными способностями.
6. Наконец, из вершин $v_{p+1}, v_{p+2}, \dots, v_{2p}$ проведем ребра в вершину c' с бесконечными пропускными способностями.

Отметим, что для данной конструкции важно, что из вершины v изначально выходят только два ребра. На рисунке 8 приведен пример преобразования для $p = 2$.

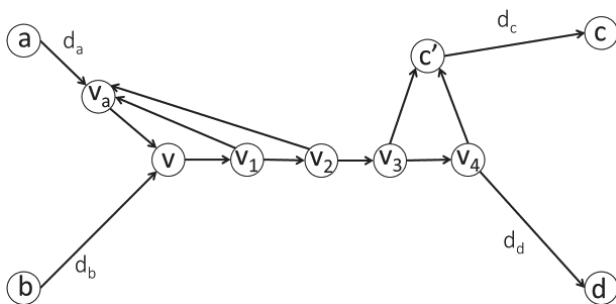


Рис. 8. $p = 2$, все неподписанные пропускные способности бесконечные

Поясним смысл проделанных преобразований. Для приведенной ранее конструкции было важно, чтобы при выборе первого ребра для поиска пути до стока рассматривалось сначала обратное ребро, ведущее в «переключатель» меньшего размера, а затем — ребро, идущее в нужное состояние «переключателя» текущего уровня. Цепочка вершин $v_1, v_2, \dots, v_p$ отвечает за перенаправление поиска на обратное ребро в вершину a , а цепочка $v_{p+1}, v_{p+2}, \dots, v_{2p}$ перенаправляет поиск в вершину c . В обоих случаях

вероятность пройти весь путь и не перейти в v_a (или c') составляет только $\frac{1}{2^p}$.

Можно выбрать число p достаточно большим, чтобы вероятность пойти в изначальной конструкции не по нужному ребру была достаточно малой. Например, если $p = 3k$, то вероятность хотя бы раз свернуть в неправильную сторону при поиске хотя бы одного из 2^k путей не превосходит $\frac{2^k k}{2^{3k}} \leq \frac{1}{2^k}$. Таким образом, математическое ожидание времени работы на графе из n вершин будет не менее $O(2^{\frac{n}{4 \log n}} n)$, что по-прежнему означает экспоненциальный рост времени выполнения.

Заключение

Добавление в наборы входных данных для задач с автоматической проверкой решений тестов, построенных с использованием конструкций, описанных в данной работе, позволит лучше отделять по времени работы решения, реализующие алгоритм Форда—Фалкерсона, от решений, использующих более эффективные алгоритмы, такие как алгоритм Эдмондса—Карпа, алгоритм Диница и др. Это будет полезно как в учебных задачах по теории потоков, так и на школьных и студенческих олимпиадах по программированию.

Отдельно стоит отметить, что во многих случаях наиболее сложной частью учебных и соревновательных задач на теорию потоков является не реализация алгоритма нахождения потока, а правильный выбор сети и ее построение по входным данным. Часто это сильно ограничивает возможные структуры графов и значения пропускных способностей, которые могут быть использованы в качестве тестовых данных. В таком случае разработке тестовых примеров следует отдельно проработать вопрос, как тесты различных структур, в том числе приведенный в данной статье, могут быть выражены в терминах исходной задачи.

Список использованных источников

1. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. 2-е изд. / пер. с англ. М.: Вильямс, 2009.
2. Dean B. C., Goemans M. X., Immerlica N. Finite termination of “augmenting path” algorithms in the presence of irrational problem data // Algorithms — ESA 2006. Azar Y., Erlebach T. (eds). Lecture Notes in Computer Science. 2006. V. 4168. Berlin, Heidelberg: Springer, 2006.
3. Dinitz Ye. Algorithm for solution of a problem of maximum flow in a network with power estimation // Doklady Akademii nauk SSSR. 1970. № 11.
4. Edmonds J., Karp R. M. Theoretical improvements in algorithmic efficiency for network flow problems // Journal of the ACM. Association for Computing Machinery. 1972. № 19 (2). doi:10.1145/321694.321699.
5. Ford L. R., Fulkerson D. R. Maximal flow through a network // Canadian Journal of Mathematics. 1956. № 8. doi:10.4153/CJM-1956-045-5.
6. Orlin J. B. Max flows in $O(nm)$ Time, or better // STOC '13 Proceedings of the forty-fifth annual ACM symposium on Theory of computing (Palo Alto, California, USA — June 01–04, 2013). New York, USA: ACM, 2013.