

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
по дисциплине «Основы алгоритмизации и программирования»
на тему: «Подпрограммы на языке Си»

Исполнитель: студент гр.

Руководитель: преподаватель

Дата проверки: _____

Дата допуска к защите: _____

Дата защиты: _____

Оценка работы: _____

Подписи членов комиссии по защите курсовой работы:

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1. Аналитический обзор методов	4
1.1 Система непересекающихся множеств	4
1.2 Алгоритм Борувки	5
1.3 Алгоритм Краскала	6
1.3.1 Общие сведения	6
1.3.2 Корректность	7
1.4 Алгоритм Прима	7
1.4.1 Общие сведения	7
1.4.2 Тривиальная реализация: алгоритмы за $O \times (n \times m)$ и $O \times (n^2 + m \times \log n)$	8
1.4.3 Случай плотных графов: алгоритм за $O(n^2)$	9
1.5 Метод Монте–Карло: Фильтрация	9
1.5.1 Фильтрация или задача о протекании	9
2. РАЗРАБОТКА АЛГОРИТМА ПРОГРАММНОГО КОМПЛЕКСА ..	12
2.1 Общие сведения	12
2.2 Приложение структур данных для непересекающихся множеств	14
2.3 Представление непересекающихся множеств с помощью связанных списков	14
2.4 Простая реализация объединения	15
2.5 Весовая эвристика	15
2.6 Леса непересекающихся множеств	17
2.7 Эвристики для снижения времени работы	17
3. Описание разработанного программного обеспечения	18
3.1 Постановка задачи и исходные данные	18
3.3 Верификация программного обеспечения	20

ВВЕДЕНИЕ

В процессе жизнедеятельности, зачастую, мы сталкиваемся с таким явлением, как фильтрация жидкости. Песок, уголь, шунгит – все это природные фильтры. Песок, с легкостью устранил из жидкости частицы, которые нерастворимы в ней. Уголь обладает бактерицидными свойствами. Т.е. он может очистить воду от органических загрязнителей, а также сделать воду более приятной на вкус. Фильтры окружают нас повсеместно. Фильтры мы используем в быту. У многих из нас, на кухне, есть небольшой фильтр, в котором всегда есть чистая вода. Вода, также, проходит обработку, прежде чем попасть в кран. Если задуматься, то роль фильтров в жизни человека невозможно преувеличить. Однако мало кто задумывается о том, что же все-таки такое, эти фильтры. Как известно, жидкость не может проходить через твердые тела. Именно поэтому, встречая на своем пути твердые породы, водные потоки меняют свое направление. Однако люди заметили, что существует группа твердых пород, которые пропускают жидкость через себя. В процессе жизнедеятельности они обнаружили, что вода, пропущенная через некоторые породы, становится чище. Так люди, сами того не подозревая, ввели такое понятие как фильтрация. А породы, которые пропускали жидкости через себя, назвали фильтрами. Позднее, когда наука получила широкое распространение и вышла на достаточно высокий уровень, было установлено, что это явление связано с пористостью тел. Это означает, что внутри тел имеются пустые пространства, поры, которые заполнены либо жидкостью, либо (чаще всего) газом. Поры могут быть соединены, создавая сеть канальцев. Именно по этим канальцам жидкость проходит через тело. Но возникает проблема. Эти канальцы спрятаны внутри тела и недоступны для человеческого глаза. И как же понять, сможет жидкость пройти через тело, или же она попросту «обойдет» наше тело стороной. Проводить испытания с каждой породой, довольно ненадежный и нерациональный способ. Ко всему этому, далеко не каждое тело сможет пропустить жидкость через себя. Это связано с тем, что поры могут быть не связаны между собой. Либо может быть множество канальцев, которые не связаны. Попросту говоря, один такой каналец может быть «тупиком». Возникает необходимость в предварительном изучении тела. Можно просматривать под микроскопом каждый образец и давать заключение. Однако не стоит забывать о человеческом факторе. Человек может ошибиться. Хорошо изучив данный вопрос, я решил разработать алгоритм, который позволит просчитать возможность фильтрации при помощи вычислительной техники. Имея в

наличии информацию о размерах и расположении пор, данный алгоритм сможет рассчитать, будет ли жидкость проходить через заданное тело.

1. Аналитический обзор методов

1.1 Система непересекающихся множеств

О двух множествах, не имеющих ни родного общего элемента, говорят, что они не пересекаются. Если имеется несколько множеств, то говорят, что они попарно непересекающиеся, если не пересекаются никакие два из них. Легко доказать, что если счетное множество разбить на попарно непересекающиеся множества, то множество этих множеств будет или конечным, или счетным. Действительно, если мы каждому из множеств Z , на которые мы разбили счетное множество u_1, u_2, u_3, \dots , соотнесем самый первый член этой последовательности, принадлежащий Z , то, поскольку наши множества не пересекаются, каждому множеству Z будет соответствовать другой член нашей последовательности, и мы сможем расположить наши множества в виде конечной или бесконечной последовательности в соответствии с расположением последовательности u_1, u_2, u_3, \dots этих элементов. Два счетных множества называют почти непересекающимися, если они имеют только конечное (или равное нулю) число общих элементов. Докажем, что множество всех натуральных чисел можно разбить на несчетное число множеств, любые два из которых являются почти непересекающимися. Обозначим, с этой целью, для каждого действительного числа x через $E(x)$ наибольшее целое число $\leq x$, а для каждого действительного положительного числа x обозначим через $Z(x)$ множество всех чисел (очевидно, натуральных) $2^n[2E(px)+1]$, где $n=1,2,3, \dots$. Покажем теперь, что для $0 < x < y$ множества $Z(x)$ и $Z(y)$ имеют только конечное (или равное нулю) число общих элементов. Действительно, если при каких-нибудь натуральных P и q имеем $2^P[2 \times E \times (p \times x) + 1] = 2^q[2 \times E \times (p \times y) + 1]$, то отсюда легко получаем, что должно быть $P=q$, а, следовательно, $E(px) = E(qy)$ $E(Px) = E(qy)$, что, ввиду условия $0 < x < y$, дает, как легко заметить, $P_y - P_x < 1$ и $p < \frac{1}{y-x}$. Таким образом, множества $Z(x)$ и $Z(y)$ имеют менее чем $\frac{1}{y-x}$, следовательно, конечное число общих элементов. Система непересекающихся множеств – специфическая структура данных, содержащая информацию о наборе множеств, которая позволяет объединять множества и отвечать на вопрос, принадлежат ли указанные элементы к одному множеству. Назначение системы непересекающихся множеств позволяют понять такие метафоры, как задача о постройке дорог между городами, где периодически требуется отвечать на запросы о достижимости одного города из другого, и задача о

добавлении друзей в социальной сети, где требуется узнавать, связаны ли два человека цепочкой общих друзей. Изначально рассматриваются N различных элементов, каждый из которых представляет собой самостоятельное множество. Далее любые два множества допустимо объединять, при этом все элементы обоих множеств, становятся элементами результирующего множества. Очевидно, что из начального состояния (N одноэлементных множеств) через $(N-1)$ слияние будет получено состояние, при котором все элементы объединены в одно множество. Как можно будет убедиться в дальнейшем, реализации системы непересекающихся множеств достаточно просто дополнить операцией добавления нового одноэлементного множества (то есть добавления $(N+1)$ -го элемента, формирующего самостоятельное множество). Любое множество может быть уникальным образом идентифицировано с помощью одного из своих элементов: два множества не могут содержать один и тот же элемент по определению (этот факт отражён словом «непересекающихся» в названии структуры данных). Такой элемент называется представителем множества.

1.2 Алгоритм Борувки

Алгоритм Борувки—это алгоритм нахождения минимального остовного дерева в графе. Впервые был опубликован в 1926 году Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии. Несколько раз был переоткрыт, например Флореком, Перкалом и Соллином. Последний, кроме того, был единственным западным ученым из этого списка, и поэтому алгоритм часто называют алгоритм Соллина, особенно в литературе по параллельным вычислениям. Работа алгоритма состоит из нескольких итераций, каждая из которых состоит в последовательном добавлении ребер к остовному лесу графа, до тех пор, пока лес не превратится в дерево, то есть, лес, состоящий из одной компоненты связности. В псевдокоде, алгоритм можно описать так:

Изначально, пусть T — пустое множество ребер (представляющее собой остовный лес, в который каждая вершина входит в качестве отдельного дерева). Пока T не является деревом (что эквивалентно условию: пока число ребер в T меньше, чем $V-1$, где V —число вершин в графе):

- Для каждой компоненты связности (то есть, дерева в остовном лесе) в подграфе с ребрами T , найдем самое дешевое ребро, связывающее эту компоненту с некоторой другой компонентой связности. (Предполагается, что веса ребер различны, или как-то дополнительно

упорядочены так, чтобы всегда можно было найти единственное ребро с минимальным весом).

- Добавим все найденные ребра в множество T .

Полученное множество ребер T является минимальным остовным деревом входного графа.

Сложность данного алгоритма состоит в следующем. На каждой итерации число деревьев в остовном лесу уменьшается по крайней мере в два раза, поэтому всего алгоритм совершает не более $O(\log V)$ итераций. Каждая итерация может быть реализована со сложностью $O(E)$, поэтому общее время работы алгоритма составляет $O(E \times \log V)$ времени (здесь V и E — число вершин и ребер в графе, соответственно). Однако для некоторых видов графов, в частности, планарных, оно может быть уменьшено до $O(E)$. Существует также рандомизированный алгоритм построения минимального остовного дерева, основанный на алгоритме Борувки, работающий в среднем за линейное время.

1.3 Алгоритм Краскала

1.3.1 Общие сведения

В начале 19 века геометр из Берлина Якоб Штейнер поставил задачу, как соединить три деревни так, чтобы их протяженность была самой короткой. Впоследствии он обобщил эту задачу: требуется найти на плоскости такую точку, чтобы расстояние от нее до n других точек было наименьшим. В 20 веке продолжилась работа над этой темой. Было решено взять несколько точек и соединить их таким образом, чтобы расстояние между ними было самым коротким. Это все является частным случаем изучаемой проблемы. Алгоритм Краскала относится к "жадным" алгоритмам (их еще называют градиентными). Суть таковых – самый большой выигрыш на каждом шаге. Не всегда "жадные" алгоритмы дают наилучшее решение поставленной задачи. Существует теория, показывающая, что при их применении к определенным задачам они дают оптимальное решение. Это теория матроидов. Алгоритм Краскала относится к таким задачам. Алгоритм Краскала—эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Также алгоритм используется для нахождения некоторых приближений для задачи Штейнера. Алгоритм впервые описан Джозефом Краскалом в 1956 году. Суть данного алгоритма заключается в следующем. Вначале текущее множество ребер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех ребер, добавление которых к уже имеющемуся множеству не вызовет появления в нем цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких

ребер больше нет. Алгоритм завершен. Подграф данного графа, содержащий все его вершины и найденное множество ребер, является его остовным деревом минимального веса. Время работы алгоритма Краскала для графа $G = (V, E)$ зависит от реализации структуры данных для непересекающихся множеств. Будем считать, что лес непересекающихся множеств реализован с эвристиками объединения по рангу и сжатия пути, поскольку асимптотически это наиболее быстрая известная реализация.

1.3.2 Корректность

Пусть T – каркас исходного графа, построенный с помощью алгоритма Краскала, а S – его произвольный каркас. Нужно доказать, что $w(T)$ не превосходит $w(S)$. Пусть M – множество ребер S , P – множество ребер T . Если S не равно T , то найдется ребро eT каркаса T , не принадлежащее S . Присоединим eT к S . Образуется цикл, назовем его C . Удалим из C любое ребро eS , принадлежащее S . Получится новый каркас, потому что и ребер, и вершин в нем столько же. Его вес не превосходит $w(S)$, так как $w(eT)$ не больше $w(eS)$ в силу алгоритма Краскала. Эту операцию (замену ребер S на ребра T) будем повторять до тех пор, пока не получим T . Вес каждого последующего полученного каркаса не больше веса предыдущего, откуда следует, что $w(T)$ не больше, чем $w(S)$.

1.4 Алгоритм Прима

1.4.1 Общие сведения

Алгоритм Прима — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые был открыт в 1930 году чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 году, и, независимо от них, Э. Дейкстрой в 1959 году. Сам алгоритм имеет очень простой вид. Искомый минимальный остов строится постепенно, добавлением в него ребер по одному. Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно). Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов. После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких ребер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины (или, что то же самое $n-1$ ребро). В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связан, то остов найден не будет (количество

выбранных рёбер останется меньше $n-1$). Пусть граф G был связным, т.е. ответ существует. Обозначим через T остов, найденный алгоритмом Прима, а через S — минимальный остов. Очевидно, что T действительно является остовом (т.е. поддеревом графа G). Покажем, что веса S и T совпадают. Рассмотрим первый момент времени, когда в T происходило добавление ребра, не входящего в оптимальный остов S . Обозначим это ребро через e , концы его — через a и b , а множество входящих на тот момент в остов вершин — через V (согласно алгоритму, $a \in V$, $b \notin V$, либо наоборот). В оптимальном остове S вершины a и b соединяются каким-то путём P ; найдём в этом пути любое ребро G , один конец которого лежит в V , а другой — нет. Поскольку алгоритм Прима выбрал ребро e вместо ребра G , то это значит, что вес ребра G больше либо равен весу ребра e . Удалим теперь из S ребро G , и добавим ребро e . По только что сказанному, вес остова в результате не мог увеличиться (уменьшиться он тоже не мог, поскольку S было оптимальным). Кроме того, S не перестало быть остовом (в том, что связность не нарушилась, нетрудно убедиться: мы замкнули путь P в цикл, и потом удалили из этого цикла одно ребро). Итак, мы показали, что можно выбрать оптимальный остов S таким образом, что он будет включать ребро e . Повторяя эту процедуру необходимое число раз, мы получаем, что можно выбрать оптимальный остов S так, чтобы он совпадал с T . Следовательно, вес построенного алгоритмом Прима T минимален, что и требовалось доказать. Время работы алгоритма существенно зависит от того, каким образом мы производим поиск очередного минимального ребра среди подходящих рёбер. Здесь могут быть разные подходы, приводящие к разным асимптотикам и разным реализациям.

1.4.2 Тривиальная реализация: алгоритмы за $O \times (n \times m)$ и $O \times (n^2 + m \times \log n)$

Если искать каждый раз ребро простым просмотром среди всех возможных вариантов, то асимптотически будет требоваться просмотр $O(m)$ рёбер, чтобы найти среди всех допустимых ребро с наименьшим весом. Суммарная асимптотика алгоритма составит в таком случае $O(nm)$, что в худшем случае есть $O(n^3)$, — слишком медленный алгоритм. Этот алгоритм можно улучшить, если просматривать каждый раз не все рёбра, а только по одному ребру из каждой уже выбранной вершины. Для этого, например, можно отсортировать рёбра из каждой вершины в порядке возрастания весов, и хранить указатель на первое допустимое ребро (напомним, допустимы только те рёбра, которые ведут в множество ещё не выбранных вершин). Тогда, если пересчитывать эти указатели при каждом добавлении ребра в остов, суммарная асимптотика алгоритма будет $O \times (n^2 + m)$, но предварительно потребуется выполнить сортировку всех рёбер за $O \times (m \times \log n)$, что в худшем случае (для плотных графов) даёт асимптотику $O \times (n^2 \times \log n)$. Ниже мы рассмотрим два

немного других алгоритма: для плотных и для разреженных графов, получив в итоге заметно лучшую асимптотику.

1.4.3 Случай плотных графов: алгоритм за $O(n^2)$

Подойдём к вопросу поиска наименьшего ребра с другой стороны: для каждой ещё не выбранной будем хранить минимальное ребро, ведущее в уже выбранную вершину. Тогда, чтобы на текущем шаге произвести выбор минимального ребра, надо просто просмотреть эти минимальные рёбра у каждой не выбранной ещё вершины — асимптотика составит $O(n)$. Но теперь при добавлении в остов очередного ребра и вершины эти указатели надо пересчитывать. Заметим, что эти указатели могут только уменьшаться, т.е. у каждой не просмотренной ещё вершины надо либо оставить её указатель без изменения, либо присвоить ему вес ребра в только что добавленную вершину. Следовательно, эту фазу можно сделать также за $O(n)$. Таким образом, мы получили вариант алгоритма Прима с асимптотикой $O(n^2)$. В частности, такая реализация особенно удобна для решения так называемой евклидовой задачи о минимальном остове: когда даны n точек на плоскости, расстояние между которыми измеряется по стандартной евклидовой метрике, и требуется найти остов минимального веса, соединяющий их все (причём добавлять новые вершины где-либо в других местах запрещается). Эта задача решается описанным здесь алгоритмом за $O(n^2)$ времени и $O(n)$ памяти, чего не получится добиться алгоритмом Крускала.

1.5 Метод Монте–Карло: Фильтрация

Общее название группы численных методов, основанных на получении большого числа реализаций стохастического (случайного) процесса, который формируется таким образом, чтобы его вероятностные характеристики совпадали с аналогичными величинами решаемой задачи. Используется для решения задач в областях физики, математики, экономики, оптимизации, теории управления и др.

1.5.1 Фильтрация или задача о протекании

Пусть есть прямоугольная решетка, состоящая из $N \times N$ свободных и занятых узлов. Пусть вероятность того, что узел занят $-P$, и, соответственно, вероятность того, что узел свободен $(1-P)$. Будем называть кластером соседние занятые узлы и стягивающим кластером (фильтрационным кластером) такой кластер, который начинается на одной границе и заканчивается на противоположной границе решетки. Образование стягивающего кластера называется фильтрацией. Вероятность его образования в конкретной решетке $P_c(p)$ является предметом исследования. В теории фильтрации в основном рассматриваются два типа решеточных задач – задача узлов и задача связей. В задаче связей, рассматривая некоторую решётку (сетку), определяют долю

связей, при которой образуется стягивающий кластер, т.е. кластер, соединяющий две противоположные стороны системы. В задаче узлов блокируют узлы и определяют долю заблокированных узлов при образовании бесконечного кластера. Блокировать узел означает перерезать все входящие в узел связи. Идея метода Монте–Карло:

- Задать значение вероятности занятости узла P .
- Разыграть состояние каждого узла при заданном значении P , тем самым построив конкретную реализацию решетки.
- Определить, есть ли в этой реализации хотя бы один стягивающий кластер. Если есть, то увеличить счетчик числа стягивающих кластеров M_c на единицу.
- Повторить пункты 2–3 M раз.
- Получить оценку вероятности образования стягивающего кластера $P_c \approx M_c / M$.
- Повторить 1–5 для ряда значений вероятности P в интервале от 0 до 1.
- Два источника задач фильтрации.
- Физика (исторически первый источник) – задачи просачивания жидкостей в пористой среде; некоторые задачи, связанные с проводимостью.
- Математика – случайные графы.

Прямая задача: Какова доля p занятых элементов решетки, при которой возникает путь от верхнего края до нижнего? Два варианта постановки задачи: какова доля узлов (задача узлов) или какова доля связей (задача связей). Связный подграф называется кластером. Кластер, в котором есть путь от верхней до нижней границы решетки, называется фильтрационным. В бесконечной решетке фильтрационный кластер бесконечен и единственен. Порог P_c фильтрации – доля занятых узлов, при которой возникает фильтрационный кластер. Для бесконечной квадратной решетки величина P_c определена: $P_c=0,5$ для задачи связей; $P_c \approx 0,59275$ для задачи узлов. Обратная задача: какую долю узлов (или связей) надо удалить (блокировать), чтобы фильтрационный кластер распался на несвязные части. Дерево Кэли – это дерево, у которых степени всех вершин равны Z . Выберем произвольный узел и перейдем из него в один из смежных Z узлов. Из него выходит $Z-1$ ребер к другим $Z-1$ узлам, каждый из которых занят с вероятностью P . Значит, существует в среднем $(z-1) \times p$ новых занятых узлов, к которым существует путь из исходного узла. Если это число меньше 1, то вероятность найти связный путь заданной длины убывает экспоненциально по мере увеличения этой длины. Если же $(z-1) \times p > 1$, то существует положительная вероятность того, что в графе существуют пути сколь угодно большой длины (бесконечные кластеры). Таким образом, порог фильтрации P_c определяется из уравнения $(z-1) \times p_c = 1$. При достижении порога фильтрации по узлам занятые узлы бесконечной решетки образуют кластеры всех размеров из связанных между

собой узлов. Распределение кластеров по размерам следует степенному закону: число $n(S)$ кластеров, содержащих S занятых узлов, пропорционально $s^{-\tau}$. Для квадратной решетки $\tau = 187/91 = 2,054945$. Степенной закон $n(S) = CS^{-\tau}$ означает, что отношение числа кластеров одного размера к числу кластеров другого размера зависит не от их размеров S , а лишь от отношения размеров. Таким образом, фильтрационные кластеры самоподобны, или независимы от масштаба, на интервале от шага решетки до размера всей решетки.

РАЗРАБОТКА АЛГОРИТМА ПРОГРАММНОГО КОМПЛЕКСА

2.1 Общие сведения

Система непересекающихся множеств – структура данных, которая позволяет администрировать множество элементов, разбитое на непересекающиеся подмножества. При этом каждому подмножеству назначается его представитель – элемент этого подмножества. Абстрактная структура данных определяется множеством трёх операций: *Union*, *Find*, *MakeSet*. Применяется для хранения компонент связности в графах, в частности, алгоритму Краскала необходима подобная структура данных для эффективной реализации. Пусть S конечное множество, разбитое на непересекающиеся классы X_i :

$S = X_0 \cup X_1 \cup X_2 \cup \dots \cup X_k : X_i \cap X_j = \emptyset \quad \forall i, j \in \{0, 1, \dots, k\}, i \neq j$. Каждому классу X_i назначается представитель $r_i \in X_i$. Соответствующая система непересекающихся множеств поддерживает следующие операции:

- *Make-Set*(x) создает новое множество, состоящее из одного члена (который, соответственно, является его представителем) x . Поскольку множества непересекающиеся, требуется, чтобы x не входил ни в какое иное множество.
- *Union-Set*(x, y) объединяет динамические множества, которые содержат x и y (обозначим их через S_x и S_y), в новое множество. Предполагается, что до выполнения операции указанные множества не пересекались. Представителем образованного в результате множества является произвольный элемент $S_x \cup S_y$, хотя многие реализации операции *Union* выбирают новым представителем представителя множества S_x или S_y , удаляя их из коллекции S .

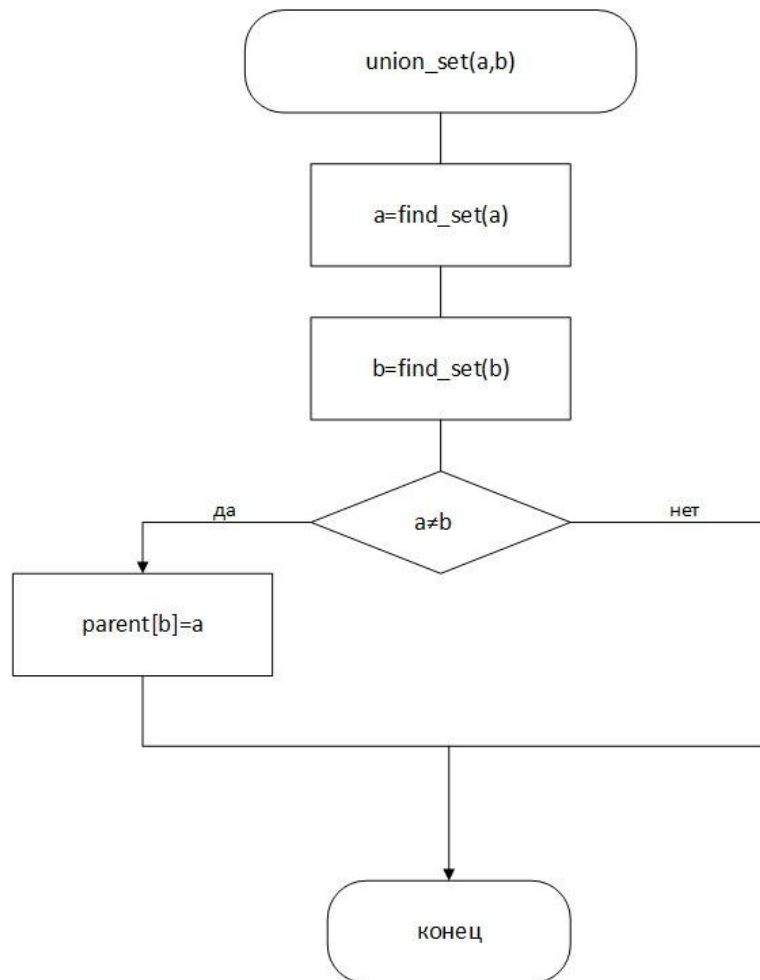


Рисунок 2.1–Графическая схема алгоритма объединения элементов

– *Find-Set(x)* возвращает указатель на представителя (единственного) множества, в котором содержится элемент x .

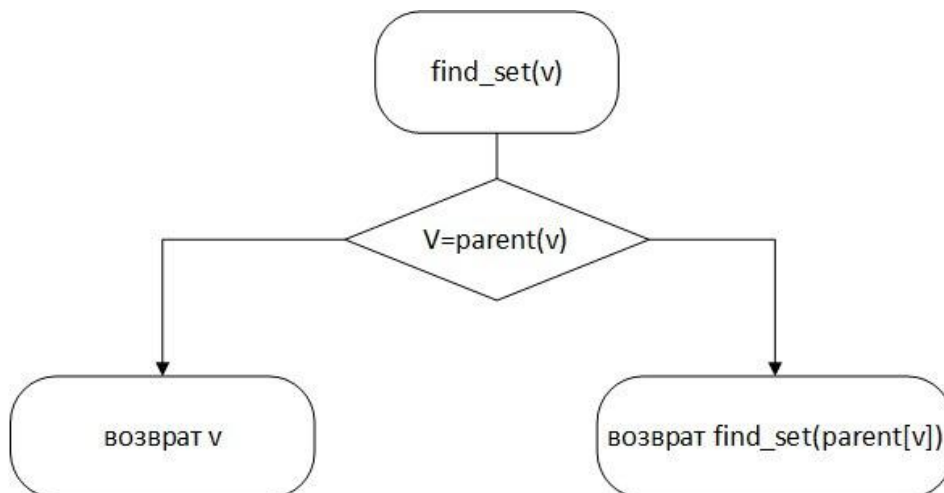


Рисунок 2.2–Графическая схема поиска родителя

Количества операций *Make-Set* n и общего количества операций *Make-Set*, *Union* и *Find-Set* m . Поскольку множества непересекающиеся, каждая операция *Union* уменьшает количество множеств на 1. Следовательно, после $n-1$ операций *Union* останется только одно множество, так что количество операций *Union* не может превышать $n-1$. Также следует заметить, что поскольку в общее количество операций m включены операции *Make-Set*, то $m \geq n$. Предполагается также, что n операций *Make-Set* являются первыми n выполненными операциями.

2.2 Приложение структур данных для непересекающихся множеств

Одно из многих применений структур данных для непересекающихся множеств—в задаче об определении связных компонентов неориентированного графа. Процедура *Connected-Components*, приведенная ниже, использует операции над непересекающимися множествами для вычисления связных компонентов графа. После того как процедура *Connected-Components* разобьет множество вершин графа на непересекающиеся множества, процедура *Same-Component* в состоянии определить принадлежат ли две данные вершины одному и тому же связному компоненту. Множество вершин G обозначим как $G.V$, а множество ребер—как $G.E$.

Процедура *Connected-Components* сначала помещает каждую вершину u в ее собственное множество. Затем для каждого ребра (u, V) выполняется объединение множеств, содержащих u и V . После обработки всех ребер две вершины будут находиться в одном связном компоненте тогда и только тогда, когда соответствующие объекты находятся в одном множестве. Таким образом, процедура *Connected-Components* вычисляет множества так, что процедура *Same-Component* может определить, находятся ли две вершины в одном и том же связном компоненте. В реальной реализации описанного алгоритма представления графа и структуры непересекающихся множеств требуют наличия взаимных ссылок, т.е. объект, представляющий вершину, должен содержать указатель на соответствующий объект в непересекающемся множестве и наоборот.

2.3 Представление непересекающихся множеств с помощью связанных списков

Каждое множество представляется своим связанным списком. Объект каждого множества имеет атрибуты *head*, указывающий на первый объект списка, и *Tail*, указывающий на последний объект списка. Каждый объект списка содержит член множества, указатель на следующий объект в списке и

указатель на объект множества. Объекты в пределах каждого списка могут располагаться в любом порядке. Представителем множества является член в первом объекте списка.

При использовании такого представления в виде связанных списков процедуры *Make-Set* и *Find-Set* легко реализуются и время их работы равно $O(1)$. Процедура *Make-Set(x)* создает новый связанный список с единственным объектом x , а процедура *Find-Set(x)* просто следует по указателю на объект множества и возвращает член объекта, на который указывает *head*.

2.4 Простая реализация объединения

Простейшая реализация операции *Union* при использовании связанных списков требует гораздо больше времени, чем процедуры *Make-Set* и *Find-Set*. Процедура *Union(x,y)* выполняется, добавляя список с элементом x в конец списка, содержащего y . Представитель списка x становится представителем результирующего списка. Мы используем указатель *Tail* для списка с элементом x для того, чтобы быстро определить, куда следует добавить список, содержащий y . Поскольку все члены списка y объединяются со списком x , можно уничтожить объект множества для списка y . К сожалению, требуется обновить указатель на объект множества для каждого объекта, исходно входящего в список y , что требует времени, линейно зависящего от длины списка y .

Действительно, нетрудно построить последовательность из m операций над n объектами, которая требует $\Theta(n^2)$ времени. Предположим, что у нас есть объекты x_1, x_2, \dots, x_n . Мы выполняем последовательность из n операций *Make-Set*, за которой следует последовательность из $n-1$ операций *Union*, так что $m=2n-1$. На выполнение n операций *Make-Set* мы тратим время $\Theta(n)$. Поскольку i -я операция *Union* обновляет i объектов, в общее количество объектов, обновленных всеми $n-1$ операциями *Union*, равно $\sum_{i=1}^{n-1} i = \theta \times (n^2)$. Общее количество операций равно $2n-1$, так что каждая операция в среднем требует для выполнения времени $\Theta(n)$. Таким образом, амортизированное время выполнения операции составляет $\Theta(n)$.

2.5 Весовая эвристика

В наихудшем случае представленная реализация процедуры *Union* требует в среднем $\Theta(n)$ времени на один вызов, поскольку может оказаться, что мы присоединяем длинный список к короткому и должны при этом обновить поля указателей на представителя всех членов длинного списка. Предположим

теперь, что каждый список включает также поле длины списка (которое легко поддерживается) и что мы всегда добавляем меньший список к большему (при одинаковых длинах порядок добавления не имеет значения). При такой простейшей весовой эвристике одна операция *Union* может потребовать время $\Omega(n)$ членов. Однако, как показывает следующая теорема, последовательность из m операций *Make-Set*, требует для выполнения время $O \times (m + n \times \lg n)$.

Теорема: при использовании связанных списков для представления непересекающихся множеств и применении весовой эвристики последовательность из m операций *Make-Set*, *Union* и *Find-Set*, n из которых составляют операции *Make-Set*, требует для выполнения время $O \times (m + n \times \lg n)$.

Доказательство. Поскольку каждая операция *Union* объединяет два непересекающихся множества, всего выполняется не более $n-1$ операции *Union*. Теперь вычислим границу для общего времени, требуемого для выполнения этих операций *Union*. Начнем с вычисления верхней границы количества обновлений указателей на объект множества для каждого из n элементов. Рассмотрим конкретный объект x . Мы знаем, что всякий раз, когда происходит обновление указателя в объекте x , он должен находиться в меньшем из объединяемых множеств. Следовательно, когда происходит первое обновление указателя в объекте x , он должен находиться в меньшем из объединяемых множеств. Следовательно, когда происходит первое обновление указателя в объекте x , образованное в результате множество содержит не менее двух элементов. При следующем обновлении указателя на представителя в объекте x полученное после объединения множество содержит не менее четырех членов. Продолжая рассуждения, приходим к выводу о том, что для произвольного $k \leq n$, после того как указатель в объекте x был обновлен $\lceil \lg k \rceil$ раз, полученное в результате множество должно иметь не менее k элементов. Поскольку максимальное множество может иметь только n элементов, во всех операциях *Union* указатель каждого объекта может быть обновлен не более $\lceil \lg n \rceil$ раз. Мы должны также учесть обновления указателей *Tail* и длин списков, для выполнения которых при каждой операции *Union* требуется $\Theta(1)$ времени. Таким образом, общее время, необходимое для обновления n объектов, составляет $O \times (n \times \lg n)$.

Отсюда легко выводится время, необходимое для всей последовательности из m операций. Каждая операция *Make-Set* и *Find-Set* занимает время $O(1)$, а всего их — $O(m)$. Таким образом, полное время выполнения всей последовательности операций — $O \times (m + n \times \lg n)$.

2.6 Леса непересекающихся множеств

В более быстрой реализации непересекающихся множеств мы представляем множества в виде корневых деревьев, каждый узел которых содержит один член множества, а каждое дерево представляет одно множество. В лесу непересекающихся множеств каждый член указывает только на родительский узел. Корень каждого дерева содержит представителя и является родительским узлом для самого себя.

Три операции над непересекающимися множествами выполняются следующим образом. Операция *Make-Set* просто создает дерево с одним узлом. Поиск в операции *Find-Set* выполняется простым перемещением до корня дерева по указателям на родительские узлы. Посещенные узлы на этом простом пути составляют путь поиска. Операция *Union* состоит в том, что корень одного дерева указывает на корень другого.

2.7 Эвристики для снижения времени работы

Пока что мы не видим никаких особых преимуществ перед реализацией с использованием связанных списков: последовательность из $n-1$ операций *Union* может создать дерево, которое представляет собой линейную цепочку из n узлов. Однако мы можем воспользоваться двумя эвристиками и достичь практически линейного времени работы от общего количества операций m .

Первая эвристика, объединение по рангу, аналогична весовой эвристике при использовании связанных списков. Ее суть заключается в том, чтобы корень дерева с меньшим количеством узлов указывал на корень дерева с большим количеством узлов. Вместо явной поддержки размера поддерева для каждого узла можно воспользоваться подходом, который упростит анализ: использовать ранг каждого корня, который представляет собой верхнюю границу высоты узла. При выполнении операции *Union* корень с меньшим рангом должен указывать на корень с большим рангом.

Вторая эвристика, сжатие пути, также достаточно проста и эффективна. Она используется в процессе выполнения операции *Find-Set* и делает каждый узел указывающим непосредственно на корень. Сжатие пути не изменяет ранги узлов.

2. Описание разработанного программного обеспечения

3.1 Постановка задачи и исходные данные

Необходимо разработать алгоритм и программу определения возможности фильтрации жидкости через пористое тело с помощью системы непересекающихся множеств, организованной на дереве, сбалансированном по количеству элементов.

Исходными данными являются:

- форма и размеры тела фильтрации;
- форма и координаты пор тела фильтрации;
- метод построения связей пор тела фильтрации;

Тело фильтрации представляет собой матрицу размерности $m \times n$, где поры обозначены цифрой один.

Ввод данных осуществляется с помощью текстового файла либо с помощью ручного ввода

3.2 Описание разработанного комплекса

Для работы с программным комплексом необходимо запустить файл *percolation.exe*. После запуска программы, на экране появится консольное окно с главным меню.

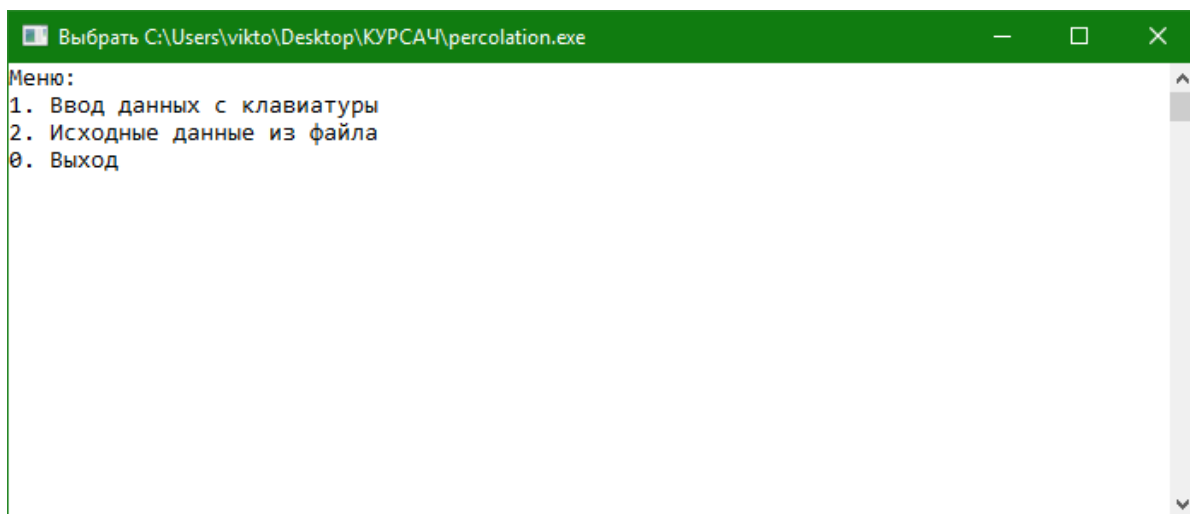


Рисунок 3.1 – Пользовательское меню

Для продолжения работы пользователю необходимо выбрать один из пунктов меню.

При выборе пункта меню «Ввод данных с клавиатуры» перед пользователем появится окно в котором необходимо ввести размеры матрицы (тела фильтрации) и значение матрицы (расположение пор).

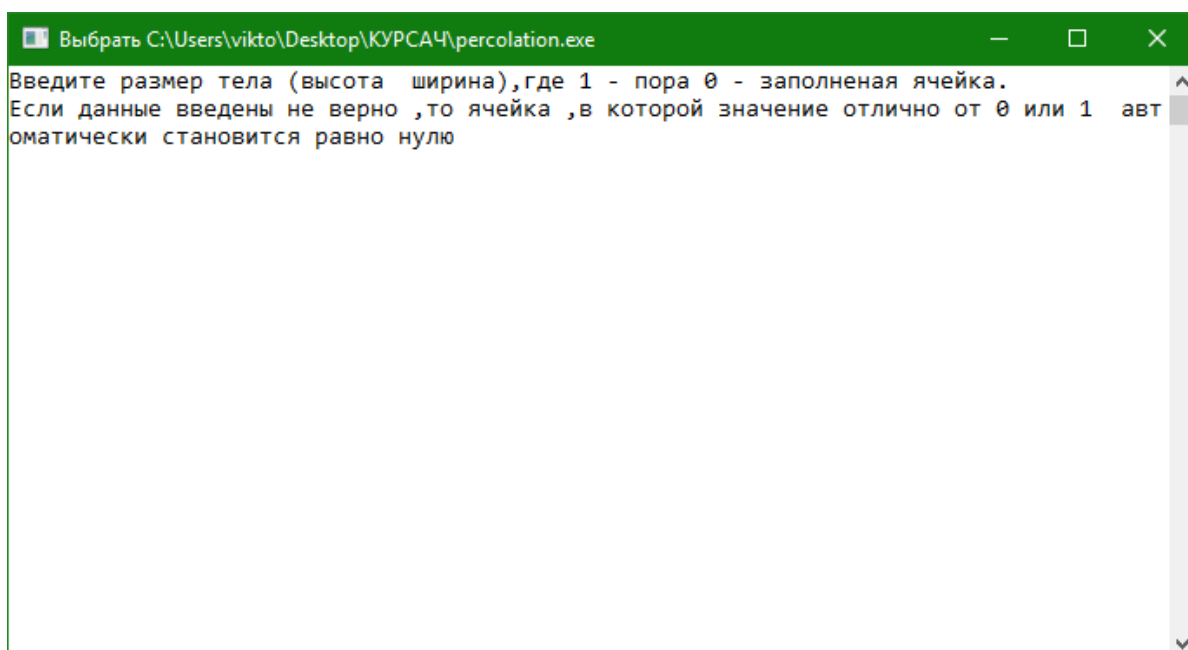


Рисунок 3.2 – Пункт меню «Ввод данных с клавиатуры»

При выборе пункта меню «Исходные данные из файла» размеры матрицы и координаты пор берутся из файла *input.txt* .

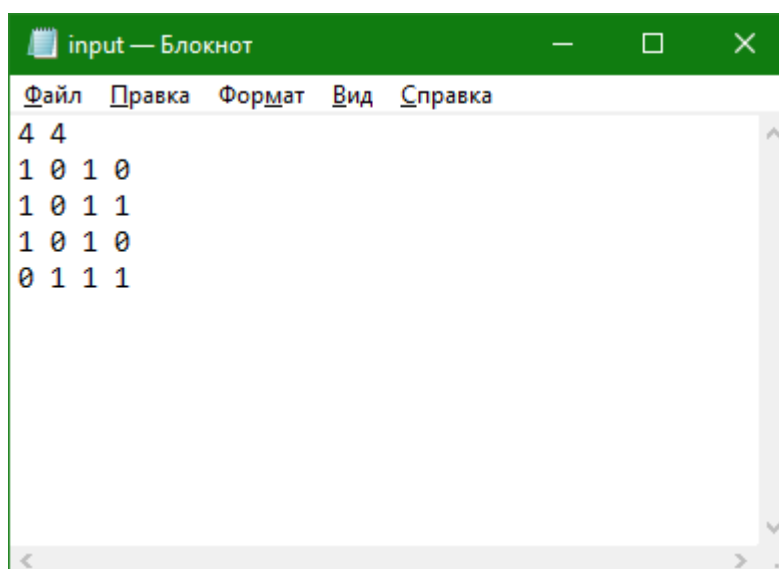


Рисунок 3.3 – Файл «input.txt»

После ввода исходных данных программа выдаст сообщение о возможности фильтрации пористого тела.

В таблице 3.1 приведены идентификаторы и типы данных, которые были использованы при разработке алгоритма программы.

Таблица 3.1 – Идентификаторы и типы данных

Название переменной	Тип переменной	Комментарий
<i>consolecolorr</i>	<i>Enum</i>	Перечисление цветов для графического отображения результата
<i>width</i>	<i>Int</i>	Ширина тела
<i>height</i>	<i>Int</i>	Высота тела
<i>num</i>	<i>Int</i>	Номер индекса элемента массива
<i>rank</i>	<i>Int</i>	Массив весов вершин
<i>Select</i>	<i>Int</i>	Значение выбора пользователя
<i>N</i>	<i>Int</i>	Размер массива <i>array</i>
<i>array</i>	<i>int[]</i>	Массив вершин
<i>Matrix</i>	<i>int[][]</i>	Исходная матрица
<i>file</i>	<i>FILE</i>	Файл с исходными значениями
<i>I</i>	<i>Int</i>	Счетчик цикла
<i>J</i>	<i>Int</i>	Счетчик цикла
<i>P</i>	<i>Int</i>	Ссылка на вершину
<i>Q</i>	<i>Int</i>	Ссылка на вершину
<i>L</i>	<i>Int</i>	Индекс элемента массива
<i>M</i>	<i>Int</i>	Счетчик индекса элемента массива

2.3 Верификация программного обеспечения

Для верификации разработанного программного комплекса составим несколько вариантов исходных данных. Исходные данные для верификации представлены в таблице 3.2–3.7. Пути протекания жидкости выделены жирным шрифтом.

Таблица 3.2–Пример 1

1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

На рисунке 3.4 представлен результат выполнения программы.

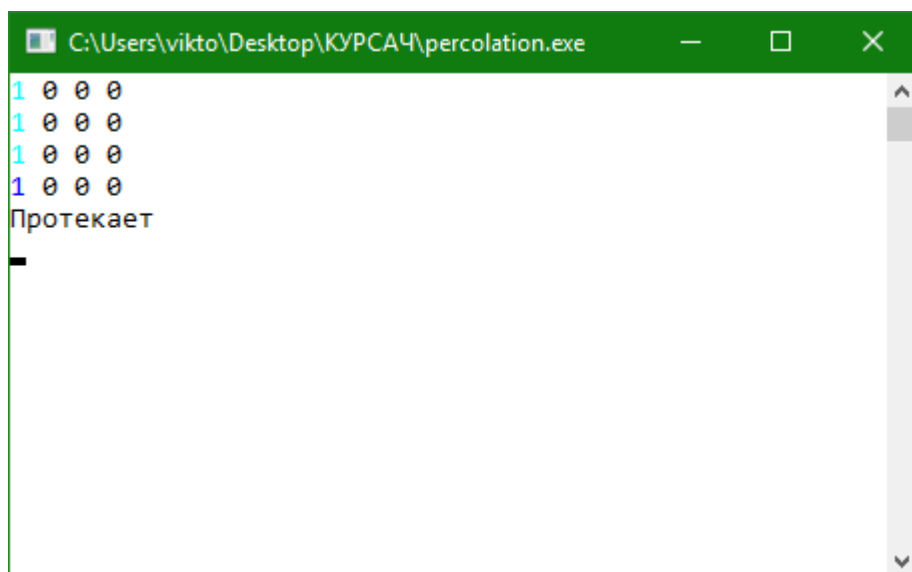


Рисунок 3.4–Результат выполнения программы

Таблица 3.3–Пример 2

1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	1

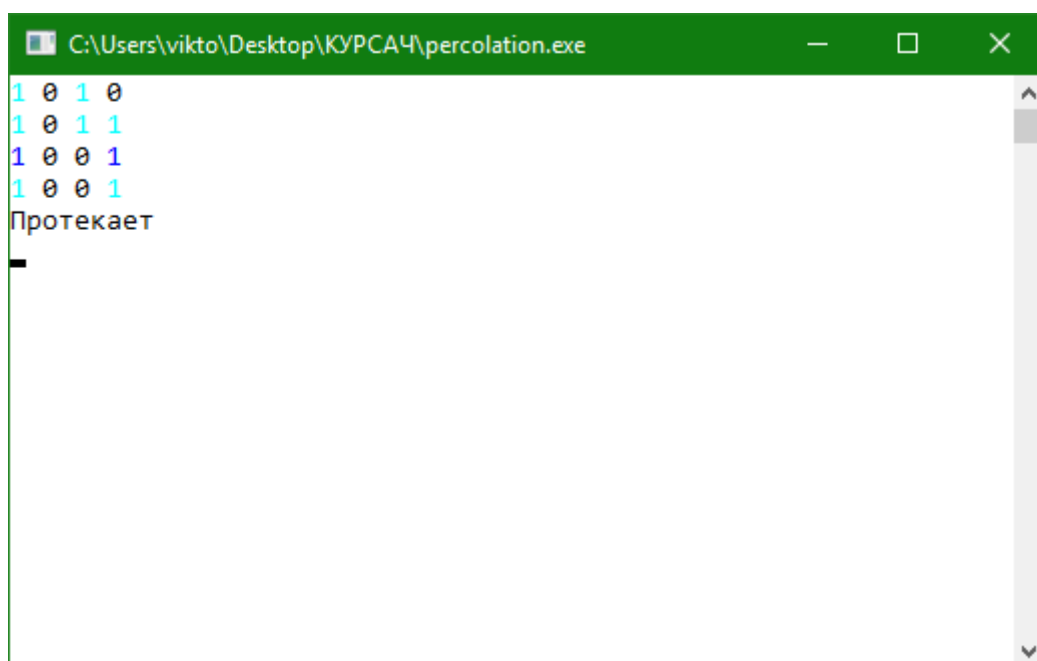


Рисунок 3.5–Результат выполнения программы

Таблица 3.4–Пример 3

0	1	0	0
0	1	1	1
0	0	0	1

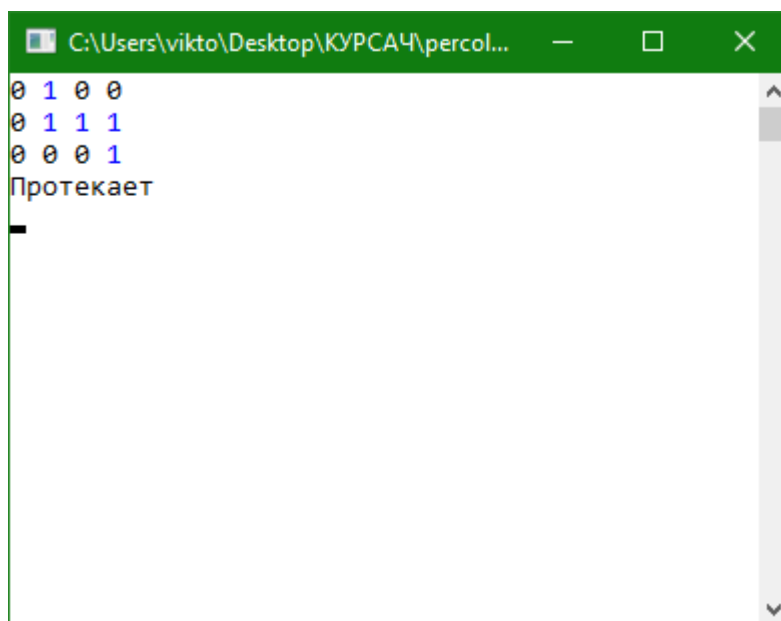


Рисунок 3.6–Результат выполнения программы

Таблица 3.5–Пример 4

0	0	0	1
0	1	1	1
1	1	0	0
1	0	0	0

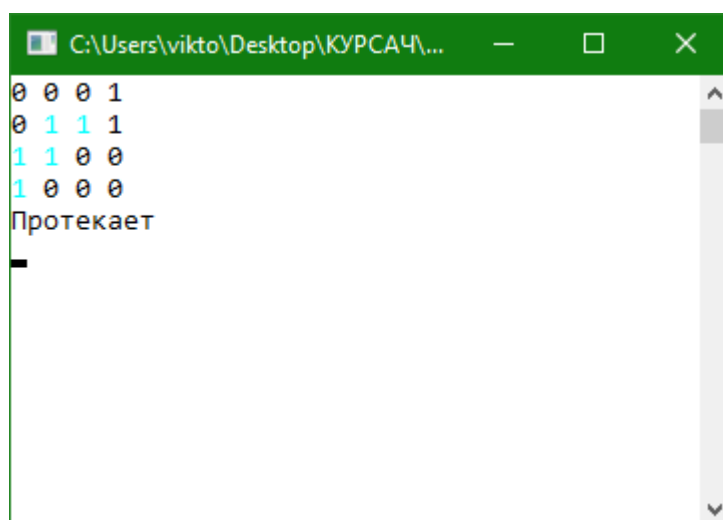


Рисунок 3.7–Результат выполнения программы

Таблица 3.6–Пример 5

1	0	0	1
1	0	0	1
1	1	1	1
0	0	0	0

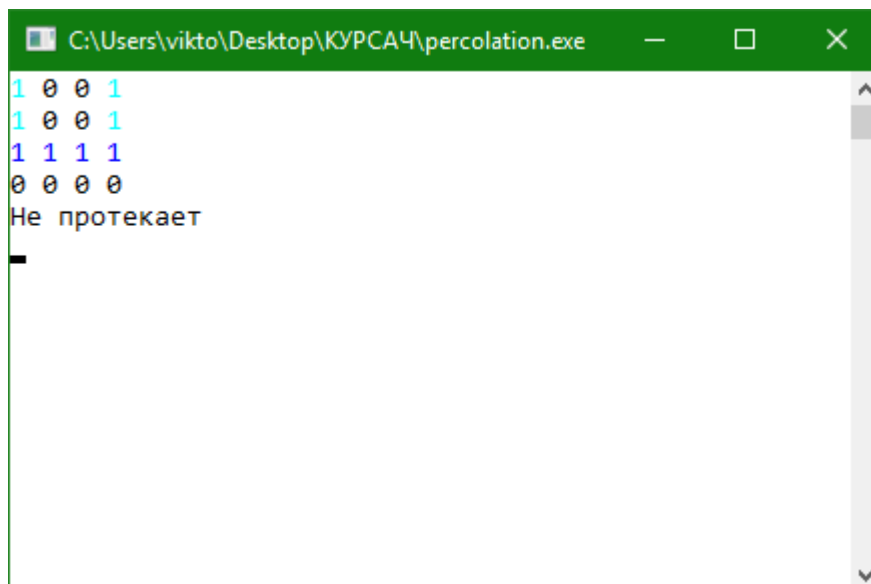


Рисунок 3.8–Результат выполнения программы

Таблица 3.7–Пример 6

1	1	0	0
1	1	0	0
1	1	1	1
0	0	0	1

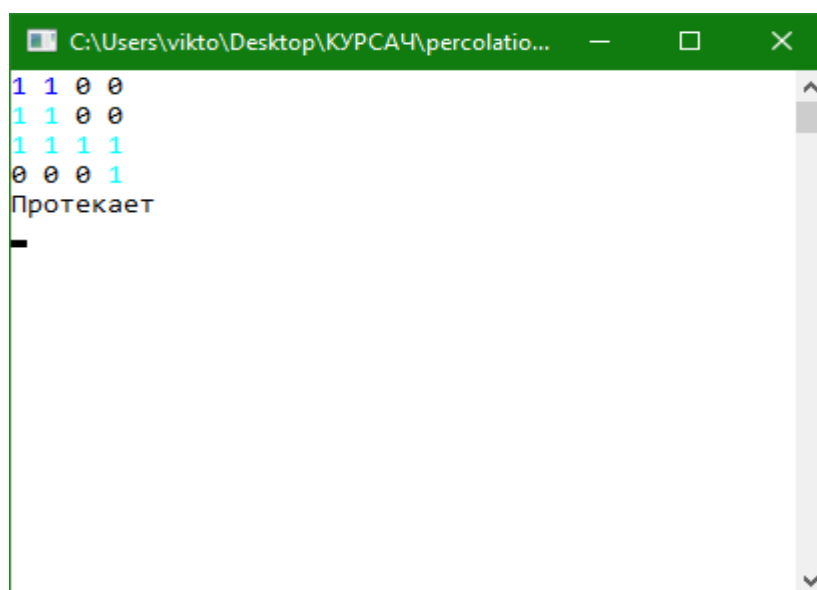


Рисунок 3.9–Результат выполнения программы

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы были разработаны алгоритм нахождения возможности фильтрации пористого тела через систему непересекающихся множеств, после чего была написана программа для реализации этой задачи на языке программирования Си в среде *Dev-C++*.

В процессе выполнения работы была изучена структура подпрограмм, механизмы передачи параметров в подпрограмму, передача одномерных массивов в функцию и вызова подпрограммы на выполнения. Были разработаны блок-схемы алгоритмов, написаны соответствующие программы и разработаны тесты для отладки данных программ.

Для написания курсовой работы были использованы методические и учебные пособия, учебники современных и иностранных авторов.

Данная курсовая работа даёт возможность глубже изучить пройденный материал, позволяет закрепить навыки решения поставленной задачи и научиться поиску необходимой для этого информации, а также помогла освоить на практике все теоретические знания работы с подпрограммами и функциями.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест и др. Алгоритмы. Построение и анализ: Учебное пособие-М.:Издательский дом «Вильямс»,2013.
2. Седжвик Р. Фундаментальные алгоритмы С++.Алгоритмы на графах: Пер. с англ. / Роберт Седжвик. – СПб:ООО «ДиаСофтЮП», 2006. – 496 с.
3. Гулд Х.,Табочник Я. Компьютерное моделирование в физике: часть 2 1999. – 320 с.
4. Habrahabr[Электронный ресурс] / Система непересекающихся множеств – Режим доступа : <https://habrahabr.ru/post/104772> – Дата доступа: 20.05.2017
5. Кравченко О.А. Основы алгоритмизации и программирования: курс лекций для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / Кравченко О.А. – Гомель: ГГТУ им. П.О. Сухого, 2010 – 112 с.
6. Герберт Ш. С++ Базовый курс: полное руководство : Научно-популярное издание / Герберт Шидлт – Москва: Издательский дом "Вильямс", 2011. – 1056 с.