РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

по дисциплине «Конструирование программ и языки программирования»

на тему: **«Алгоритмы фильтрации жидкости через пористое тело»**

Исполнитель:
Руководитель:

Дата проверки: _____
Дата допуска к защите: _____
Дата защиты: _____

Оценка работы: _____

Подписи членов комиссии

по защите курсовой работы: _____

СОДЕРЖАННИЕ

ВВЕДЕНИЕ

Фильтрация – это механический способ, с помощь которого жидкость содержащая в себе посторонние вещества, пропускается сквозь другое тело, которое может удерживать их в себе.

Теория фильтрации – один из разделов гидродинамики, посвященный исследованию движения жидкостей через пористые среды, т. е. тела, пронизанные системой сообщающихся между собой пустот (пор). Пористыми являются многие природные тела: горные породы, грунты, кожа, кость, мягкие ткани животных, древесина, а также искусственные материалы: строительные (кирпич, бетон), искусственная кожа, керамика, металлические детали. Пористой является почва, из чего можно сделать вывод: огромную роль играют пористые среды в жизни людей. Для всех этих материалов характерна способность накапливать и позволять двигаться жидкости под действием внешних сил.

В последние десятилетия к охране грунтовых вод от загрязнения отходами производства, удобрениями и прочими продуктами жизнедеятельности человечества уделяют большое внимание. Для решения этой проблемы создаются различные организации

Основными источниками энергии XXI века являются: нефть и газ. Добываются они из глубоко залегающих подземных пластов. Накопление нефти и газа в этих пористых пластах-коллекторах и основные технологии добычи следствия законов теории фильтрации и являются одним из главных источников ее задач. Важная количественная характеристика пористых тел -их пористость, определяемая отношением доли объема тела, приходящаяся на поры, на единицу объема материала. Обычно при этом не учитываются замкнутые изолированные поры и принимаются во внимание только соединенные между собой проточные поры. Они образуют сложную разветвленную сеть пор, благодаря которой мы можем наблюдать эффект перколяции.

В гидродинамике перколяцией называется явление протекания или непротекания жидкостей через пористые материалы.

Пористость – это безразмерная величина и она не зависит от размера частиц. Пористость основного количества материалов находится в пределах 0,1–0,4

1 АНАЛИТИЧЕСКИЙ ОБЗОР МЕТОДОВ И СРЕДСТВ РАЗРАБОТКИ ПРОГРАММНОГО КОМПЛЕКСА

1.1 Система непересекающихся множеств

Система непересекающихся множеств – структура данных, которая позволяет администрировать множество элементов, разбитое на непересекающиеся подмножества. При этом каждому подмножеству назначается его представитель – элемент этого подмножества.

Структура данных для непересекающихся множеств (*disition-set data structure*) поддерживает набор $S = \{S_1, S_2, ..., S_k\}$ непересекающихся множеств. Каждое множество идентифицируется представителем который представляет собой некоторый член множества. В одних приложениях не имеет значения, какой именно элемент множества используется в качестве представителя; главное, чтобы при запросе представителя множества дважды, без внесения изменений в множество между запросами, возвращался один и тот же элемент. В других приложениях может действовать предопределенное правило выбора представителя, например наименьшего члена множества (само собой разумеется, в предположении о возможности упорядочения элементов множества). В других реализациях динамических множеств, каждый элемент множества представляет некоторый объект *x*. Требуется обеспечить поддержку следующих операций:

– *Make-set*;

– *Union*;

– *Find-set*.

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(*x, y*) объединяет динамические множества, которые содержат *x* и *y* (обозначим их через $S_x S_y$), 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. Поскольку нам необходимо, чтобы все множества были непересекающимися. операция *Union* концептуально должна уничтожать множества S_x и S_y удаляя их из коллекции *S*.

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

1.1.1 Одно из многих применений структур данных для непересекающихся множеств – в задаче об определении связных компонентов

неориентированного графа. Так, на рис. 1.1 показан граф, состоящий из четырех связных компонентов.

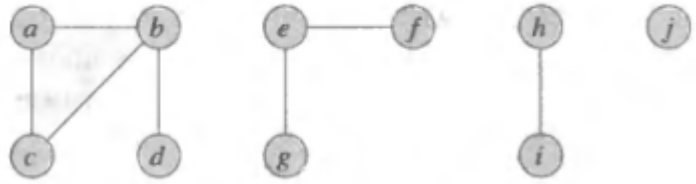


Рисунок 1.1 – Граф из четырех связных компонентов: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$ и $\{j\}$

Обработанные ребра	Набор непересекающихся множеств									
Исходные множества	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Рисунок 1.2 – Набор непересекающихся множеств после обработки каждого ребра

Процедура *Connected-Components* сначала помещает каждую вершину и в ее собственное множество. Затем для каждого ребра (u, v) выполняется объединение множеств, содержащих u и v. В соответствии с задачей 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 на одном и том же связном компоненте. На рис. 1.2 показан процесс вычисления непересекающихся множеств процедурой *Connected-Components*.

В 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 [1].

1.1.2 На рис.1.3 показан простейший способ реализации структуры данных для непересекающихся множеств: каждое множество представлено

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.

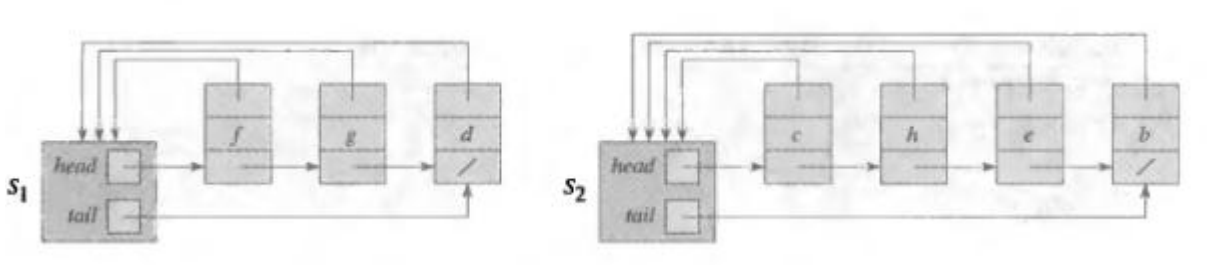


Рисунок 1.3 – Представление двух множеств в виде связных списков.

Множество S_1 содержит члены *d*, *f* и *g*, с представителем *f*, а множество S_2 содержит члены *b*, *c*, *e* и *h*, с представителем *c*. Каждый объект в списке содержит член множества, указатель на следующий объект в списке и указатель на объект множества. Каждый объект множества имеет указатели *head* и *tail* на первый и последний объекты соответственно.

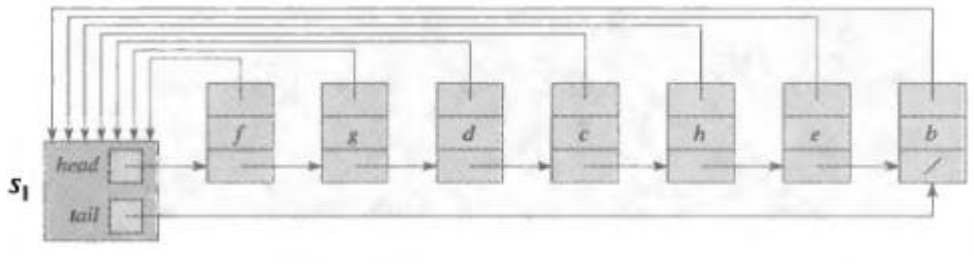


Рисунок 1.4 – Результат операции *Union*(*g*, *e*), которая добавляет связанный список, содержащий *e*, к связанному списку, содержащему *g*. Представителем полученного в результате множества является *f*. Объект множества S_2 для списка *e* уничтожается

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*. Например, на рис. 3 вызов *Find-Set*(*g*) вернет *f*.

1.1.3 В более быстрой реализации непересекающихся множеств представим множества в виде корневых деревьев, каждый узел которых содержит один член множества, а каждое дерево представляет одно множество.

В лесу непересекающихся множеств (*disition-set forest*) (рис. 1.5) каждый член указывает только на родительский узел. Корень каждого дерева содержит представителя и является родительским узлом для самого себя.

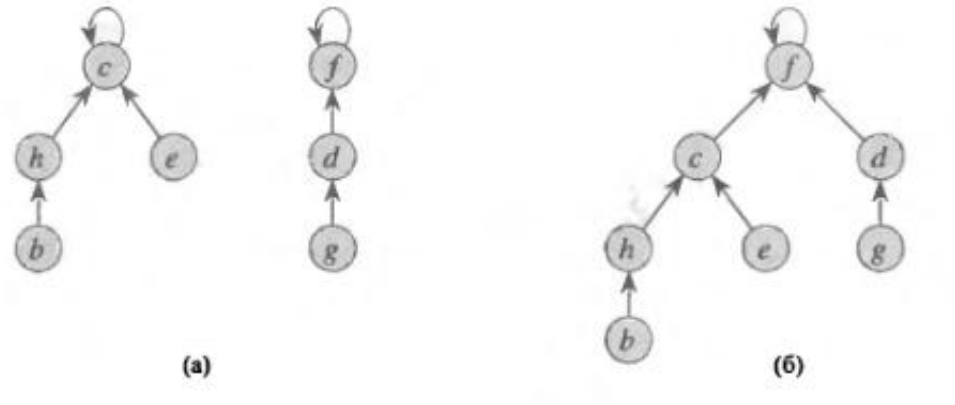


Рисунок 1.5 – Лес непересекающихся множеств.(а) Два дерева представляют два множества, показанных на рис. 1.3 – 1.4. Дерево слева представляет множество $\{b, c, e, h\}$, где с является представителем ,а дерево справа представляет множество $\{d, f, g\}$ с представителем *f*. (б) Результат выполнения *Union*(*e*,*g*)

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 (*find path*). Операция *Union*, показанная на рис.1.5,(б), состоит в том, что корень одного дерева указывает на корень другого.

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:

1. Изначально, пусть T – пустое множество ребер (представляющее собой остовной лес, в который каждая вершина входит в качестве отдельного дерева).

2. 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.

3. 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 $O(E \log V)$ 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

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 Крускала 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. Инициализация множества A занимает время $O(1)$, а время, необходимое для сортировки множества равно $O(E \times \log E)$.

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

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 рёбер 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 рёбер 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 связен, 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).

Тривиальная реализация: алгоритмы за $O(n \times m)$ и $O(n^2 + m \log n)$ 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 $O(n \times m)$, 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 $O(n^2 + m)$, но предварительно потребуется выполнить сортировку всех рёбер за $O(m \times \log n)$, что в худшем случае (для плотных графов) даёт асимптотику $O(n^2 \times \log n)$. Ниже мы рассмотрим два немного других алгоритма: для плотных и для разреженных графов, получив в итоге заметно лучшую асимптотику.

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: Перколяция

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 Пусть есть прямоугольная решетка, состоящая из 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 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 блокированных 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.

1.Физика (исторически первый источник) – задачи просачивания жидкостей в пористой среде; некоторые задачи, связанные с проводимостью.

2.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 *pc* фильтрации – доля занятых узлов, при которой возникает фильтрационный кластер. Для бесконечной квадратной решетки величина *pc* определена: *pc*=0,5 для задачи связей; *pc* ≈ 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* новых занятых узлов, к которым существует путь из исходного узла. Если это число меньше единицы, то вероятность найти связный путь заданной длины убывает экспоненциально по мере увеличения этой длины. Если же $(z-1)p>1$, то существует положительная вероятность того, что в графе существуют пути сколь угодно большой длины (бесконечные кластеры). Таким образом, порог фильтрации *pc* определяется из уравнения $(z-1)p_c=1$. 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^{-t}. Для квадратной решетки *t*=187/91=2,(054945). Степенной закон $n(s)=cs^{-t}$ означает, что отношение числа кластеров одного размера к числу кластеров другого размера зависит не от их размеров *s*, а лишь от отношения размеров[3].

2 РАЗРАБОТКА АЛГОРИТМА ПРОГРАММНОГО КОМПЛЕКСА

Решение любой задачи осуществляется по определенному плану, называемому алгоритмом.

Алгоритм – понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение указанной цели или на решение поставленной задачи. К алгоритму относится, например, кулинарный рецепт приготовления какого-либо блюда. Алгоритмом является также инструкция по включению какого-либо прибора, например, компьютера. Таким образом, алгоритм – последовательность предписывающих правил, понятно и точно указывающих исполнителю, какую последовательность действий он должен исполнить по переработке исходных данных в искомые результаты. Рассмотрим основные свойства, которыми должен обладать создаваемый алгоритм.

Поставим следующую задачу. Допустим, имеются элементы *N* видов (для простоты, здесь и далее – числами от 0 до *N-1*). Некоторые группы чисел объединены в множества. Также можем добавить в структуру новый элемент, он тем самым образует множество размера один из самого себя. И наконец, периодически некоторые два множества потребуется сливать в одно.

Определимся сначала, в каком виде будет храниться вся информация. Множества элементов, образующих дерево, будет храниться в виде массива. Значение элемента массива – это представитель (лидер) множества. При реализации это означает, что мы заводим массив, в котором для каждого элемента мы храним ссылку на его предка в дерева. Для корней деревьев считать, что их предок – они сами.

Реализация операции поиска лидера (*find_set*(*v*)) проста: мы поднимаемся по предкам от вершины *v*, пока не дойдём до корня, т.е. пока ссылка на предка не ведёт в себя. Эту операцию удобнее реализовать рекурсивно. На рисунке 2.1 представлена графическая схема алгоритма.

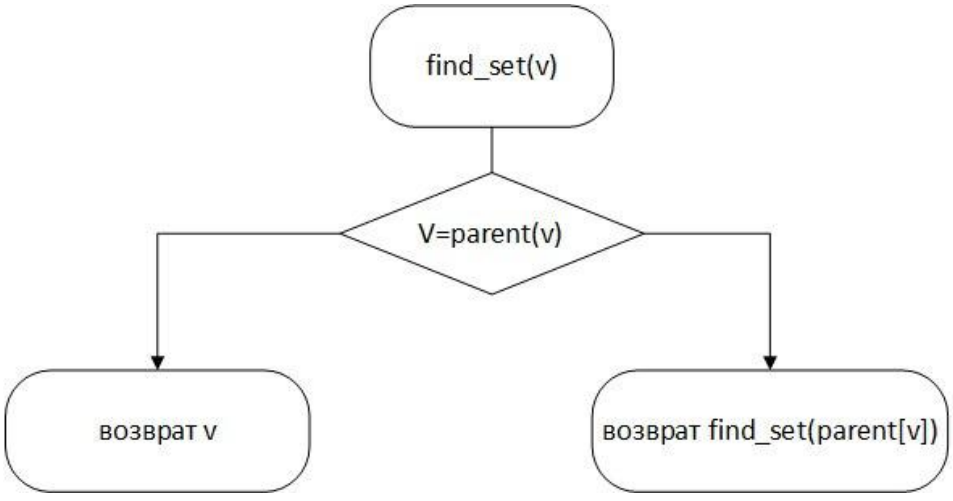


Рисунок 2.1 – Графическая схема алгоритма поиска родителя

Чтобы объединить два множества (операция *union_sets*(*a*,*b*)), сначала необходимо найти лидеров множества, в котором находится *a*, и множества, в котором находится *b*. Если лидеры совпали, то ничего не делаем – это значит, что множества и так уже были объединены. В противном случае можно просто указать, что предок вершины *b* равен *a* (или наоборот) – тем самым присоединив одно дерево к другому. На рисунке 2.1 представлена графическая схема алгоритма.

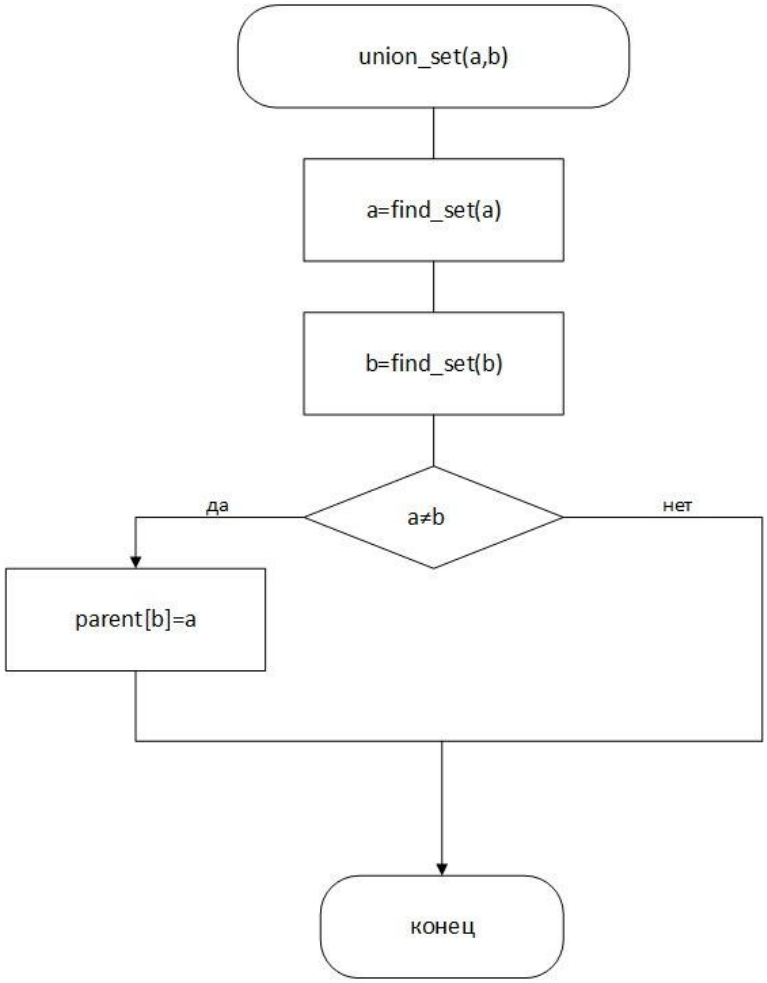


Рисунок 2.2 – Графическая схема алгоритма объединения элементов

Простейшая реализация операции *Union* при использовании связанных списков требует гораздо больше времени, чем процедуры *Make-Set* и *Find-Set*. Как показано на рис. 1.4 выполняется процедура *Union*(*x*,*y*), добавляя список с элементом *x*: в конец списка, содержащего *y*. Представитель списка *x*: становится представителем результирующего списка. Используя 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*. На рис. 1.3, например, операция *Union*(*g,e*) требует обновления указателей объектов *b*, *c*, *e* и *h*.

Действительно, нетрудно построить последовательность из *m* операций над *n* объектами, которая требует $\theta(n^2)$ времени. Предположим, что есть объекты $x_1, x_2, ..., x_n$. Выполняем последовательность из *n* операций *Make-Set*, за которой следует последовательность из *n* - 1 операций *Union*, показанных на рис. 1.4, так что $m = 2n - 1$. На выполнение *n* операций *Make-Set* тратим время $\theta(n)$. Поскольку *i*-я операция *Union* обновляет *i* объектов, общее количество объектов, обновленных всеми *n*-1 операциями *Union*, равно:

$$\sum_{i=1}^{n-1} i = \theta(n^2) \qquad (2.1)$$

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 $\theta(n)$.

Для любого элемента множества представитель всегда одинаковый. Поэтому чтобы проверить принадлежность элементов *x* и *y* одному множеству достаточно сравнить *find(x)* и *find(y)*. На рисунке 2.3 представлен процесс сравнения и объединения элементов.

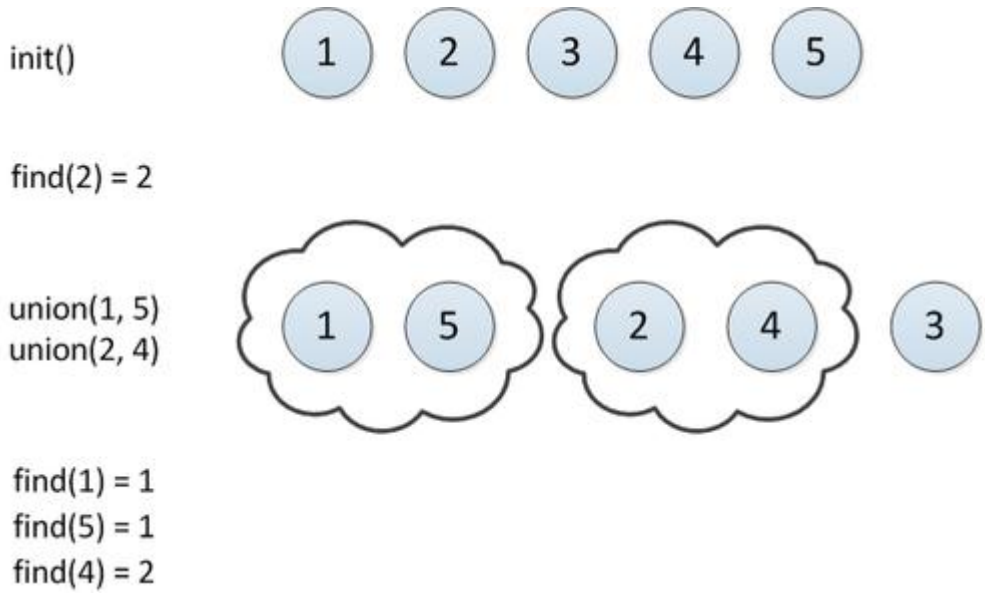


Рисунок 2.3 – Процесс сравнения и объединения элементов

Иногда в конкретных приложениях системы непересекающихся множеств всплывает требование поддерживать расстояние до лидера (т.е. длину пути в рёбрах в дереве от текущей вершины до корня дерева). В этом случае расстояние до корня просто равнялось бы числу рекурсивных вызовов, которые сделала функция *find_set*.[4]

2.1.1 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 находится 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 [lGk] 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 [lGn] 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 – $O \times (m - n \times \lg n)$.

3 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 одномерном массиве.

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 программного комплекса

Для работы с программой необходимо открыть файл *body.exe*. После появления консольного приложения пользователю нужно выбрать способ ввода данных. На рисунке 3.1 показано пользовательское меню.

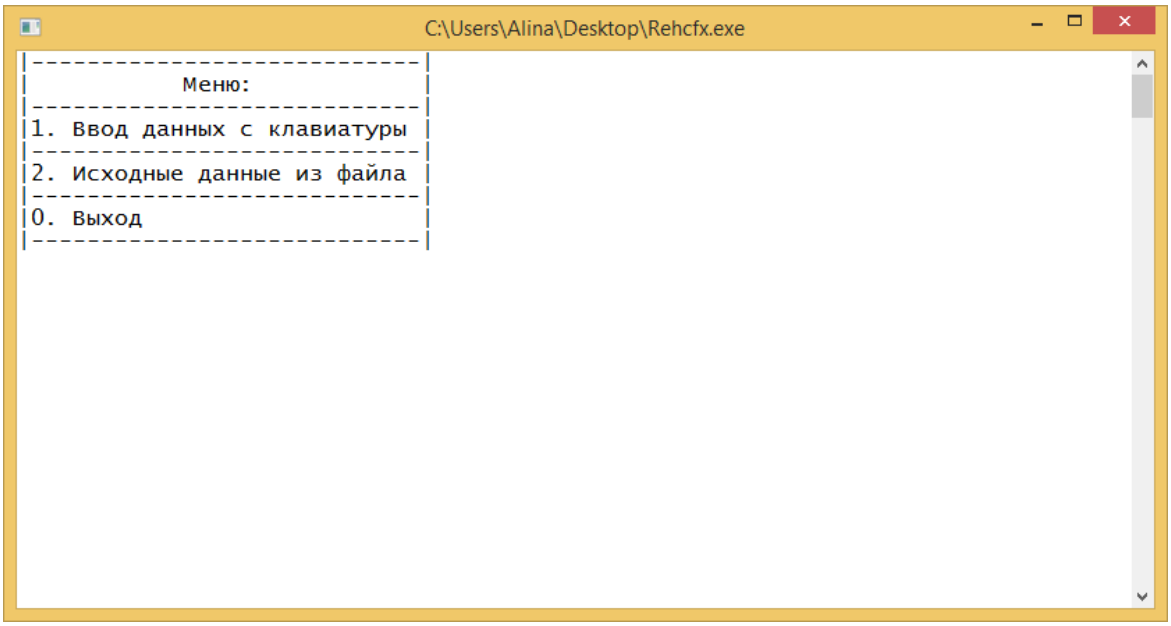


Рисунок 3.1 – Пользовательское меню

Пользователю необходимо ввести номер соответствующего пункта и нажать клавишу *Enter*. В случае выбора первого пункта пользователю будет предложено ввести размеры тела, после чего необходимо будет заполнить тело

порами (число один означает пору, число ноль означает отсутствие поры). На рисунке 3.2 показан процесс ввода.

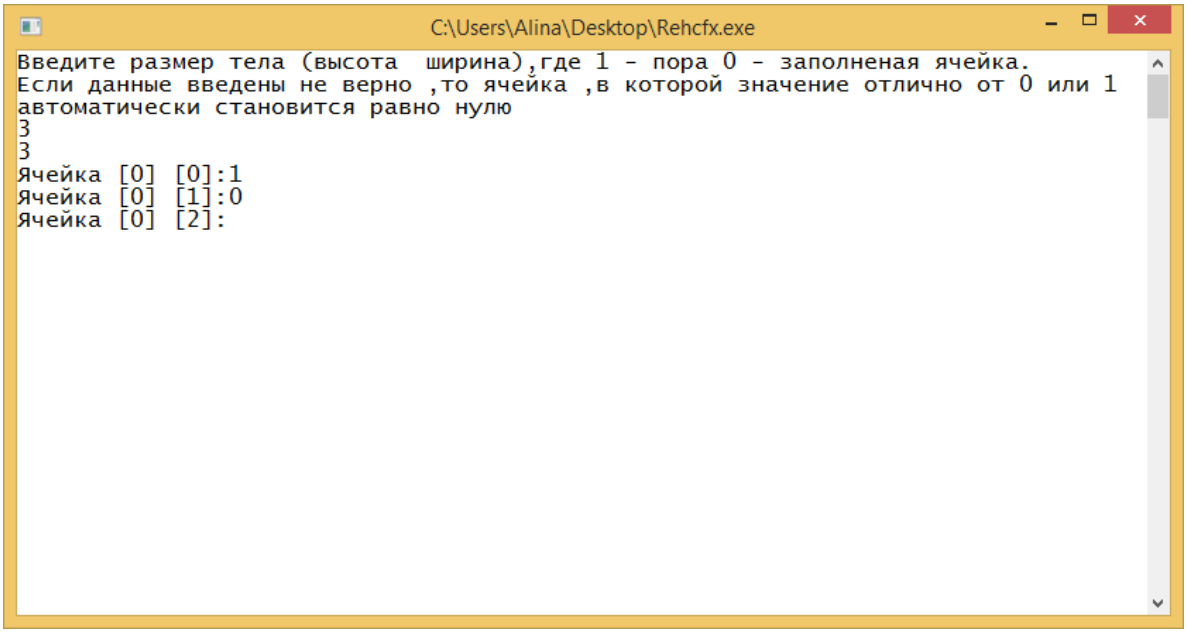


Рисунок 3.2 – Ручной ввод данных

В случае ввода пользователем чисел отличных от нуля и единицы, число будет автоматически изменено на ноль.

Если будет выбран ввод данных из файла (пункт два), то размеры тела и координаты пор будут считаны с текстового файла (*input.txt*).На рисунке 3.3 представлен пример текстового файла.

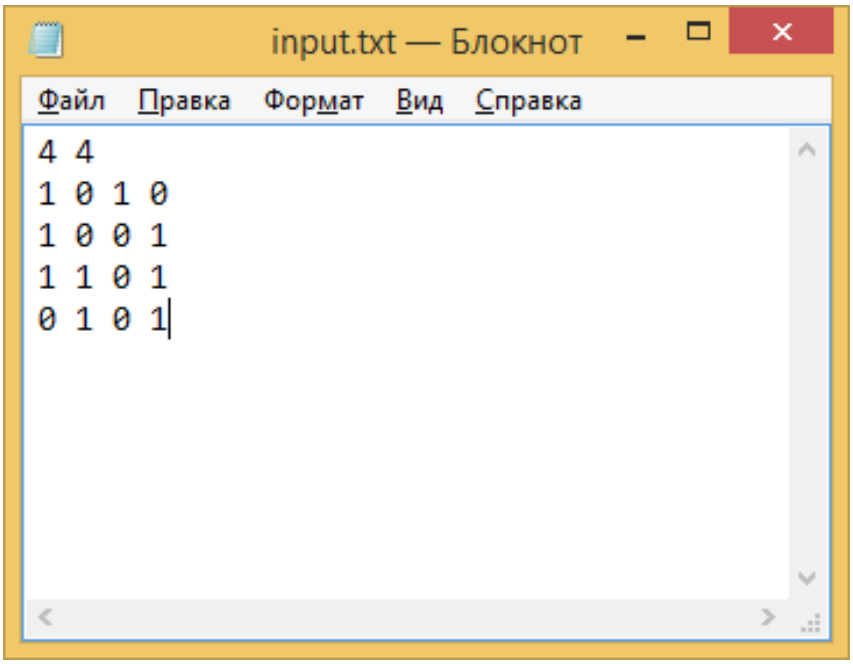


Рисунок 3.3 – Пример текстового файла

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**
ConsoleColor	*enum*	Перечисление цветов для графического отображения результата
width	*int*	Ширина тела
height	*int*	Высота тела
num	*int*	Номер индекса элемента массива
select	*int*	Значение выбора пользователя
n	*int*	Размер массива array
array	*int*[]	Массив вершин
matrix	*int*[][]	Исходная матрица
file	*FILE*	Файл с исходными значениями
i	*int*	Счетчик цикла
j	*int*	Счетчик цикла
p	*int*	Ссылка на вершину
q	*int*	Ссылка на вершину
l	*int*	Индекс элемента массива
m	*int*	Счетчик индекса элемента массива

3.3 Верификация программного обеспечения

Для верификации разработанного программного комплекса составим несколько вариантов тестовых данных. Исходные данные представлены в таблице 3.2 –3.6. Пути протекания выделены жирным шрифтом.

Таблица 3.2 – Пример 1

1	0	**1**	**1**
1	0	**1**	**1**
1	0	0	**1**
1	0	0	**1**

На рисунке 3.4 показан результат работы программы

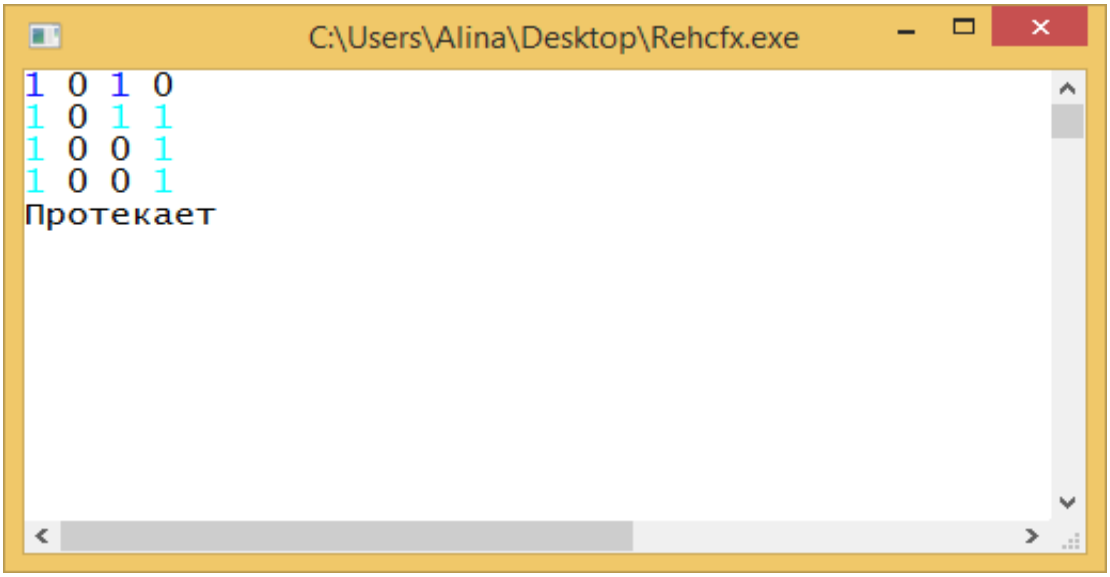


Рисунок 3.4 – Результат работы программы

Таблица 3.3 – Пример 2

1	**1**	**1**	**1**
1	**1**	**1**	**1**
1	**1**	**1**	**1**
0	0	0	0

На рисунке 3.5 показан результат работы программы

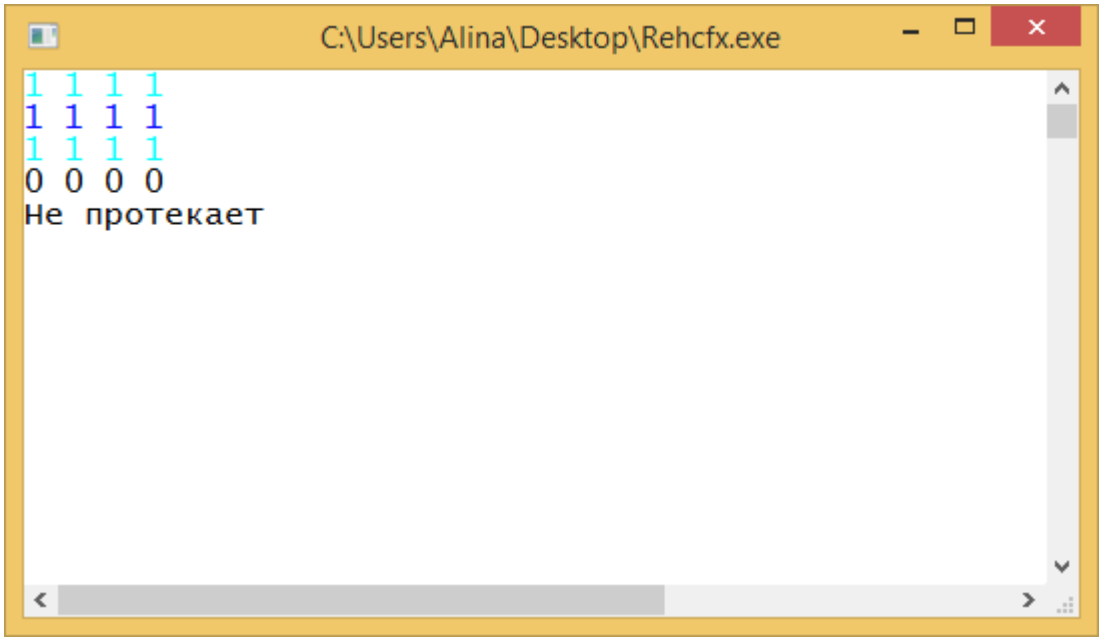


Рисунок 3.5 – Результат работы программы

Таблица 3.4 – Пример 3

1	0	0	0
1	**1**	0	0
0	**1**	**1**	0
0	0	**1**	**1**

На рисунке 3.6 показан результат работы программы

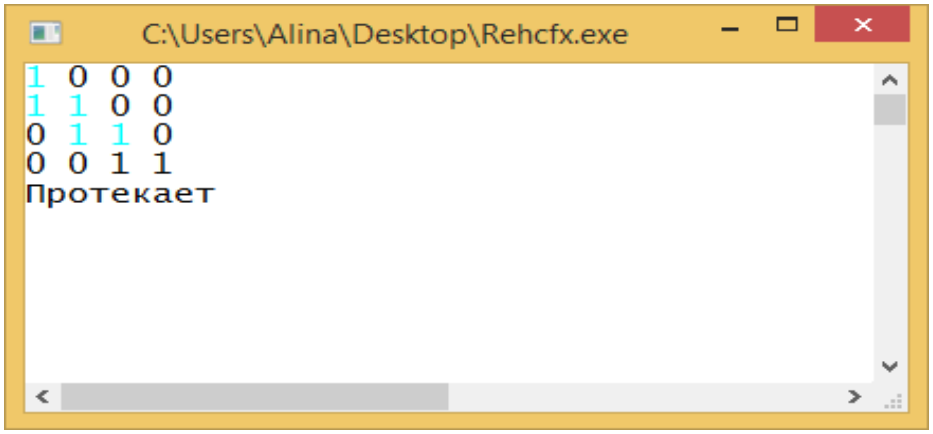


Рисунок 3.6 – Результат работы программы

Таблица 3.5 – Пример 4

1	0	0	0
1	**1**	0	0
0	**1**	**1**	0
0	**1**	0	**1**

На рисунке 3.7 показан результат работы программы

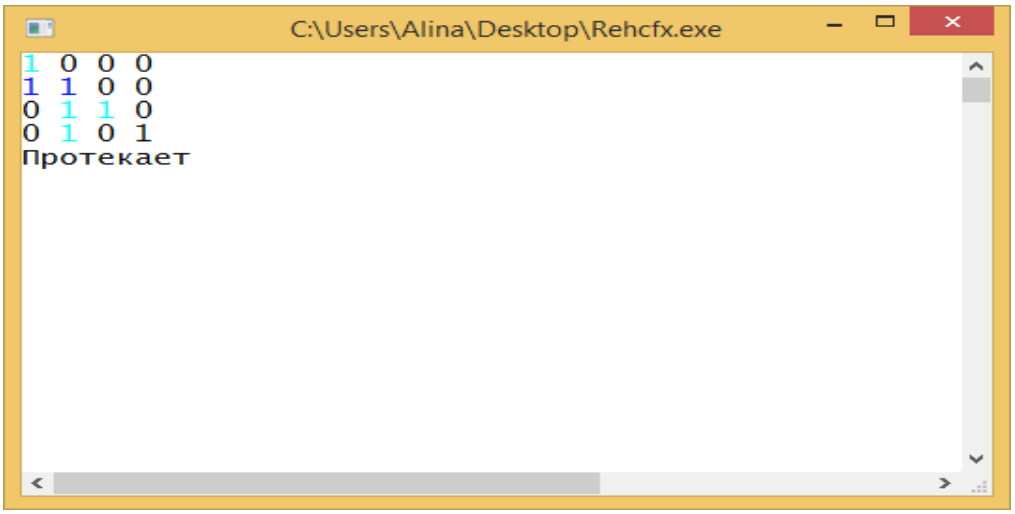


Рисунок 3.7 – Результат работы программы

Таблица 3.6 – Пример 5

1	0	0	0	0	0	0
1	**1**	0	0	0	**1**	**1**
0	**1**	**1**	0	**1**	**1**	0
0	0	**1**	**1**	**1**	0	0
0	0	0	**1**	**1**	0	0
0	0	0	**1**	0	0	0
0	0	0	**1**	0	0	0

На рисунке 3.8 показан результат работы программы

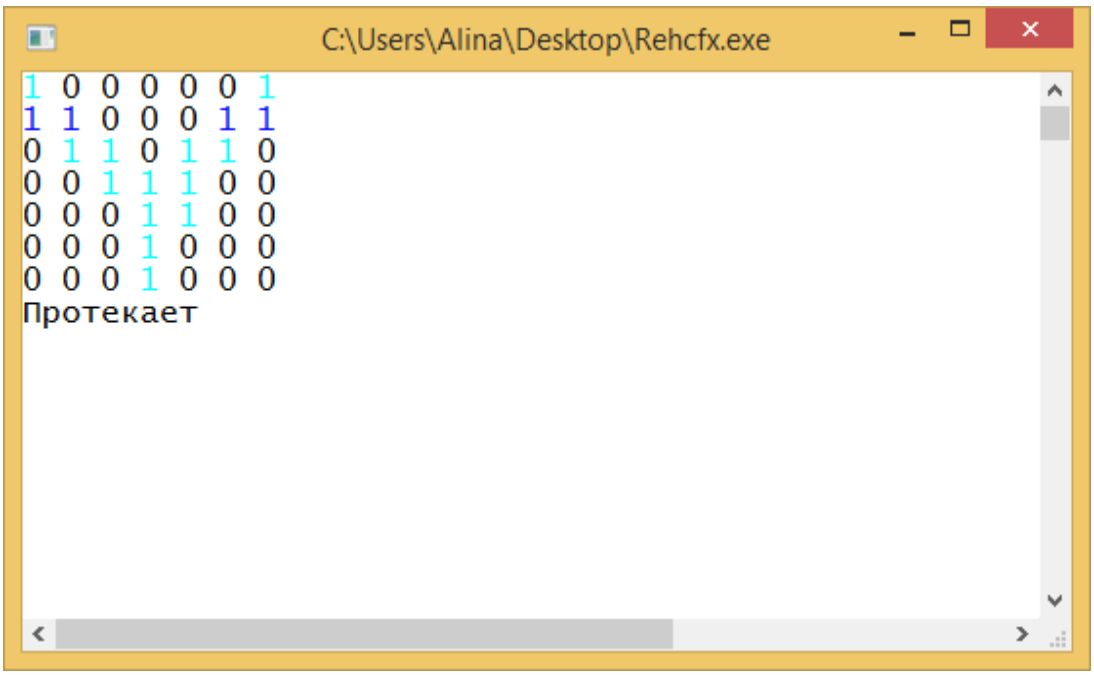


Рисунок 3.8 – Результат работы программы

ЗАКЛЮЧЕНИЕ

В ходе выполнении 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. Кормен Т., Лейзерсон Ч., Ривест Р. и др. Алгоритмы. Построение и анализ: Учебное пособие - М.:Издательский дом «Вильямс»,2013.
2. Седжвик Р. Фундаментальные алгоритмы C++. 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. Герберт Ш. C++ Базовый курс: полное руководство: Научно-популярное издание / Герберт Шидлт – Москва: Издательский дом "Вильямс", 2011. – 1056 с.

ПРИЛОЖЕНИЕ А
(обязательное)
Руководство Пользователя

1. Введение

Разработанная программа позволяет просчитать возможность фильтрации жидкости через пористое тело. Она может быть полезна для студентов, изучающих алгоритмы фильтрации. Программа проста в использовании, подсказки в ходе использования доступны и понятны. Так же программа имеет графическое отображение результата.

2. Запуск

Для работы с программой необходимо открыть файл *body.exe*. После появления консольного приложения пользователю нужно выбрать способ ввода данных. После вывода результата можно закрыть программу выбором соответствующего пункта меню.

3. Непредвиденные ситуации

При возникновении любых непредвиденных ситуаций рекомендуется перезапустить приложение «*body*». Если после повторного использования программы возникают ошибки, то следует сообщить об ошибке разработчику данной программы для ее устранения.

ПРИЛОЖЕНИЕ Б
(обязательное)
Код программы

```
#include <stdio.h>
#include <iostream>
#include<conio.h>
#include <windows.h>//подключение библиотек

int Find(int *,int);//нахождение элемента
void Union(int *,int,int,int);//объединение элементов

enum ConsoleColor //Перечисление цветов
{
	Black = 0,
	Blue = 1,
	Green = 2,
	Cyan = 3,
	Red = 4,
	Magenta = 5,
	Brown = 6,
	LightGray = 7,
	DarkGray = 8,
	LightBlue = 9,
	LightGreen = 10,
	LightCyan = 11,
	LightRed = 12,
	LightMagenta = 13,
	Yellow = 14,
	White = 15
};

int main()
{
	HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE); //получение дескриптора
	SetConsoleTextAttribute(hConsole, (WORD) ((White << 4) | Black)); //установка цвета фона и шрифта
   setlocale(LC_ALL, "Russian");//установка языка
   int width = 0;//ширина
   int height = 0;//высота
   int num=0,//номер индекса элемента массива
		select;//значение выбора пользователя
   int n =width * height + 2;//размер массива array
   int *array =new int[n];//массив вершин
   int **matrix= new int *[width];//исходная матрица
```

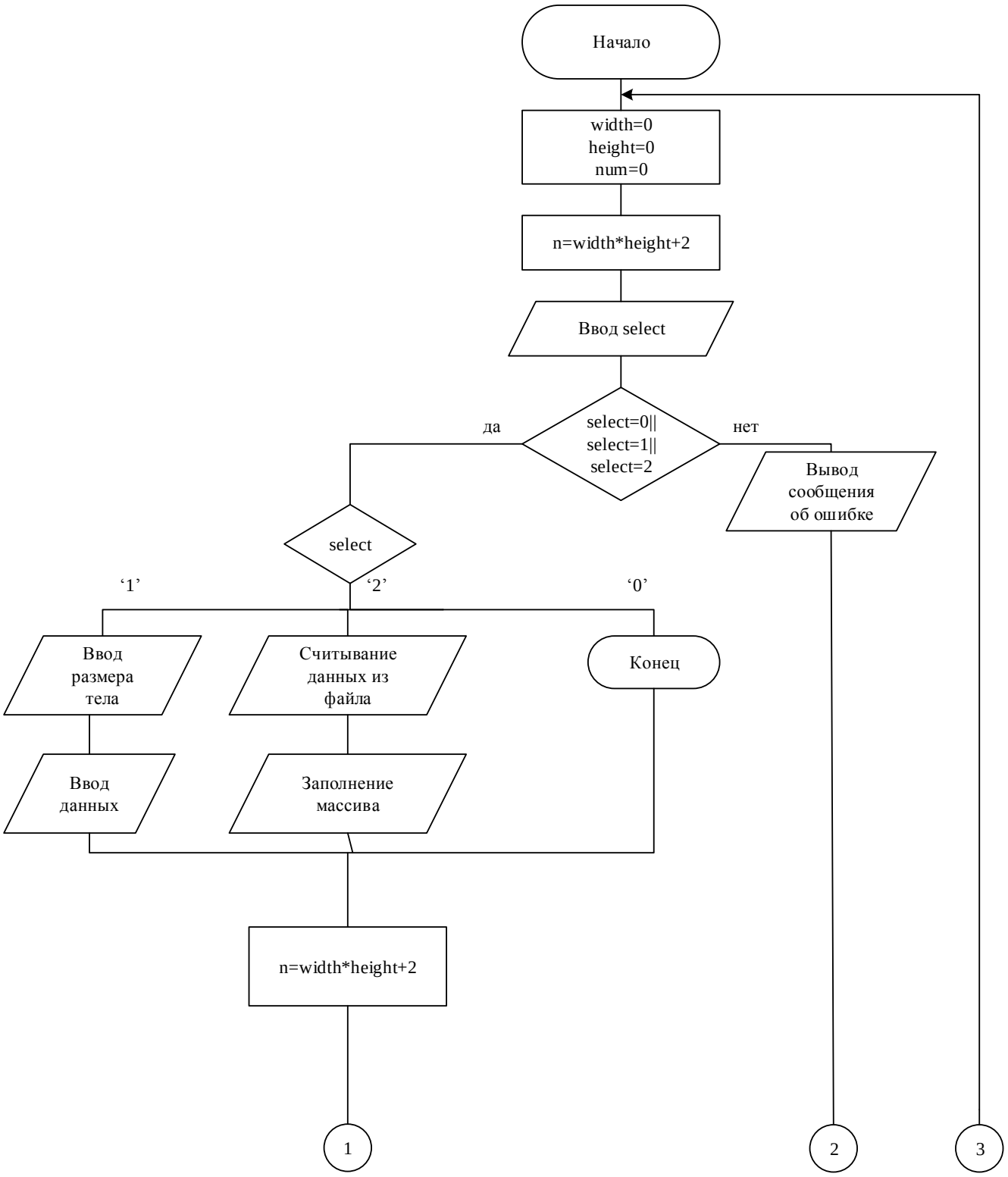
```
do
        {
        system("cls");
   puts("|---------------------------|");
        puts("|          Меню:           |");
   puts("|---------------------------|");
   puts("|1. Ввод данных с клавиатуры |");
   puts("|---------------------------|");
   puts("|2. Исходные данные из файла |");
   puts("|---------------------------|");
   puts("|0. Выход                   |");
   puts("|---------------------------|");

   scanf("%d", &select);
   if(select==1||select==2||select==0)
   {
     system("cls");
     switch(select)
               {
       case 1:
                                        {
                                                printf("Введите размер тела (высота  ширина),где 1 -
пора 0 - заполненая ячейка.\nЕсли данные введены не верно ,то ячейка ,в которой значение
отлично от 0 или 1  автоматически становится равно нулю\n");
                          scanf("%d %d", &height, &width);
                                for (int i =0;i < height;i++)
                                                        {
                                  matrix[i] = new int[width];
                                  for (int j =0;j < width; j++)
                                                                {
                                     printf("Ячейка [%d] [%d]:", i, j);
                                     scanf("%d", &matrix[i][j]);
                                  if(matrix[i][j] != 0 && matrix[i][j] != 1)
                                  matrix[i][j] = 0;
                                                        }
                                  }
                          }
       break;
       case 2:
                          {
          FILE *file=fopen("input.txt", "a+");
          fscanf(file, "%d %d", &height, &width);
          for (int i =0;i < height;i++)
                                {
            matrix[i] =new int[width];
            for (int j =0;j < width;j++)
                                        {
              fscanf(file, "%d", &matrix[i][j]);
              printf("%d ", matrix[i][j]);
             }
             printf("\n");
          }
```

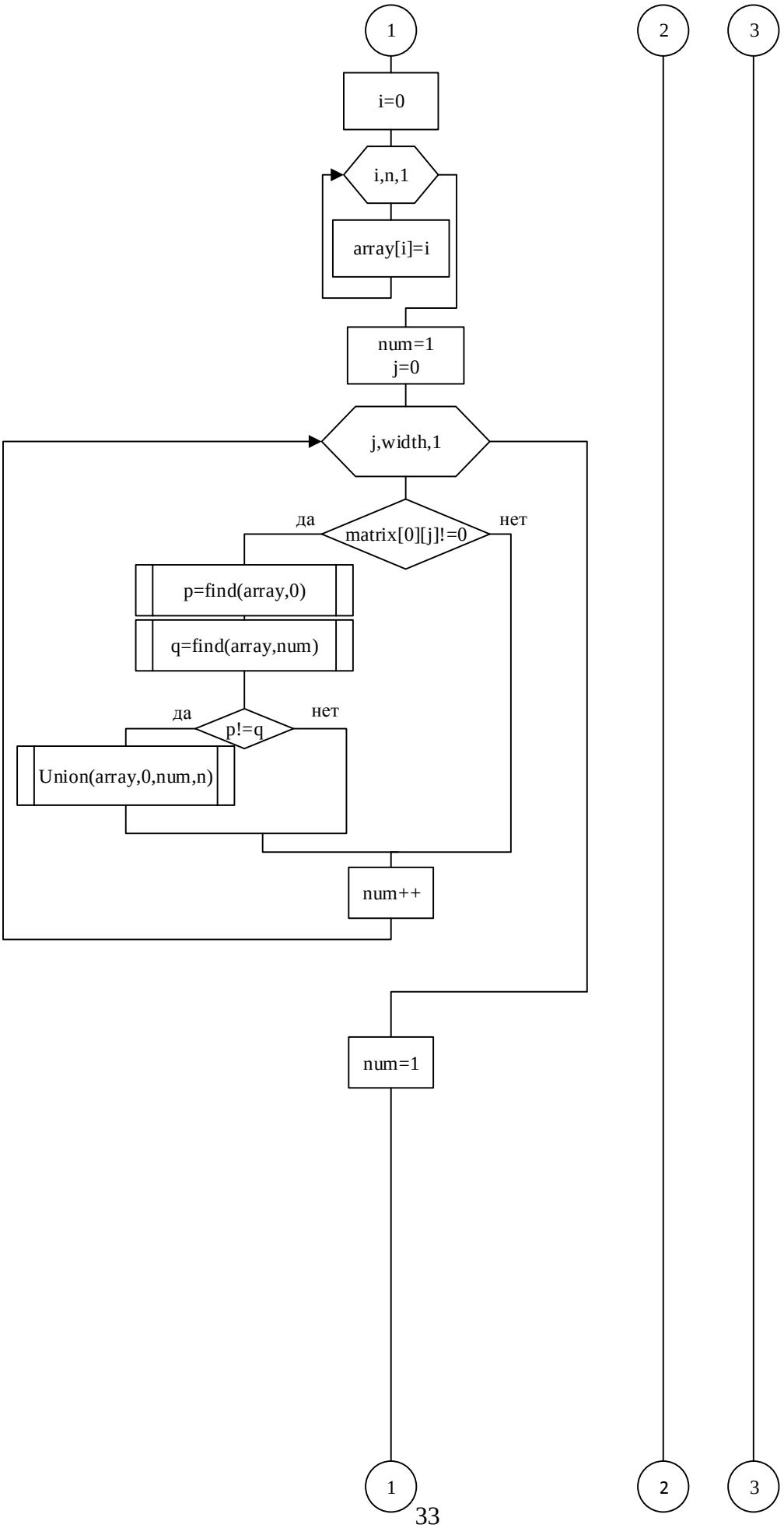
```
            fclose(file);
        }
        break;
        case 0:
                return 0;
                                break;

    }
    n=width*height+2;
  for (int i =0;i < n;i++)
        {
    array[i] =i;
  }
  num =1;
  for (int j =0;j < width;j++) //верх
  {
    if (matrix[0][j] != 0)
                {
        int p =Find(array, 0);
        int q =Find(array, num);
        if (p != q) Union(array, 0, num, n);
    }
    num++;
  }
  num =1;
  for (int i =0;i < height - 1;i++) //центр
  {
    for (int j =0;j < width;j++) {
        if (matrix[i][j] != 0) {
            if (j + 1 < width && matrix[i][j + 1] != 0) {
                int p =Find(array, num);
                int q =Find(array, num + 1);
                if (p != q) Union(array, p, q, n);
            }
            if (i + 1 < height && matrix[i + 1][j] != 0) {
                int p =Find(array, num);
                int q =Find(array, num + width);
                if (p != q) Union(array, p, q, n);
            }
        }
        num++;
    }
  }
  num =n - width - 1;
for (int j = 0; j < width; j++) //низ
        {
                if (matrix[height - 1][j] != 0)
                {
                        int p = Find(array, num);
                        int q = Find(array, n - 1);

                        if (p != q)
```

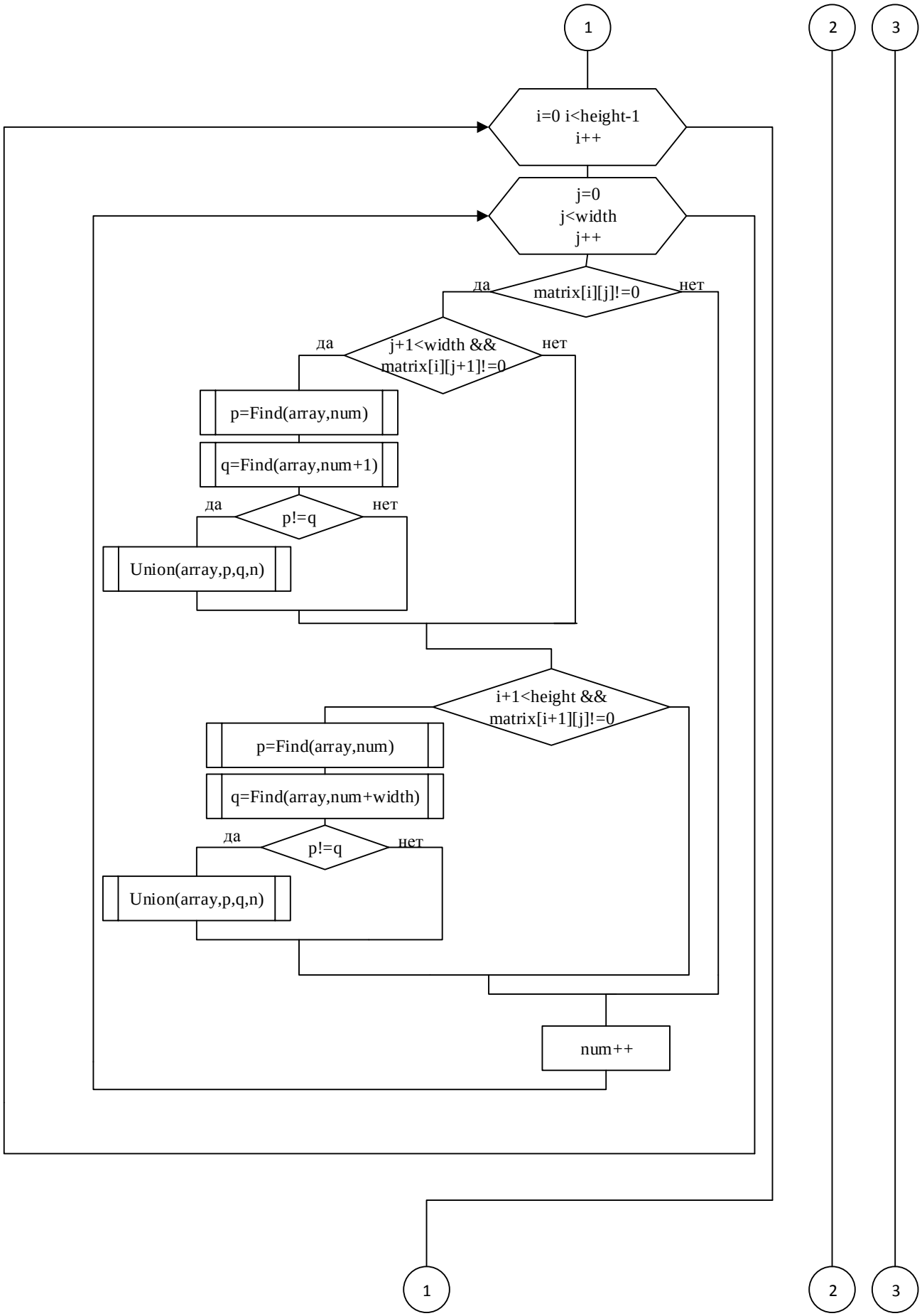
```
                        Union(array, num, n - 1, n);
                }

                if (array[0] == array[n - 1])
                        break;
                num++;
        }

   int m = 0;

   do
   {
        system("cls");
           int l = 1;
           for (int i = 0;i < height;i++)
                {
              for (int j =0;j < width;j++)
                        {
                                if(m > height)
                                m = 0;

                                if(j == m && array[l] == array[0])
                                {
                                        SetConsoleTextAttribute(hConsole, (WORD) ((White << 4) |
LightBlue));
                }
                else if(array[l] == array[0])
                {
                                        SetConsoleTextAttribute(hConsole, (WORD) ((White << 4) |
LightCyan));
                }
                printf("%d ", matrix[i][j]);
                l++;
                m++;
                SetConsoleTextAttribute(hConsole, (WORD) ((White << 4) | Black));
             }
                printf("\n");
           }

           if (array[0]==array[n - 1])
         printf("Протекает \n");
   else printf("Не протекает\n");

           Sleep(500);
        }while(!kbhit());}
        else
        printf("Данные введены не корректно\n");
   puts("Для продолжения нажмите любую клавишу...");
        getch();
   }
        while(select);
   fflush(stdin);
```

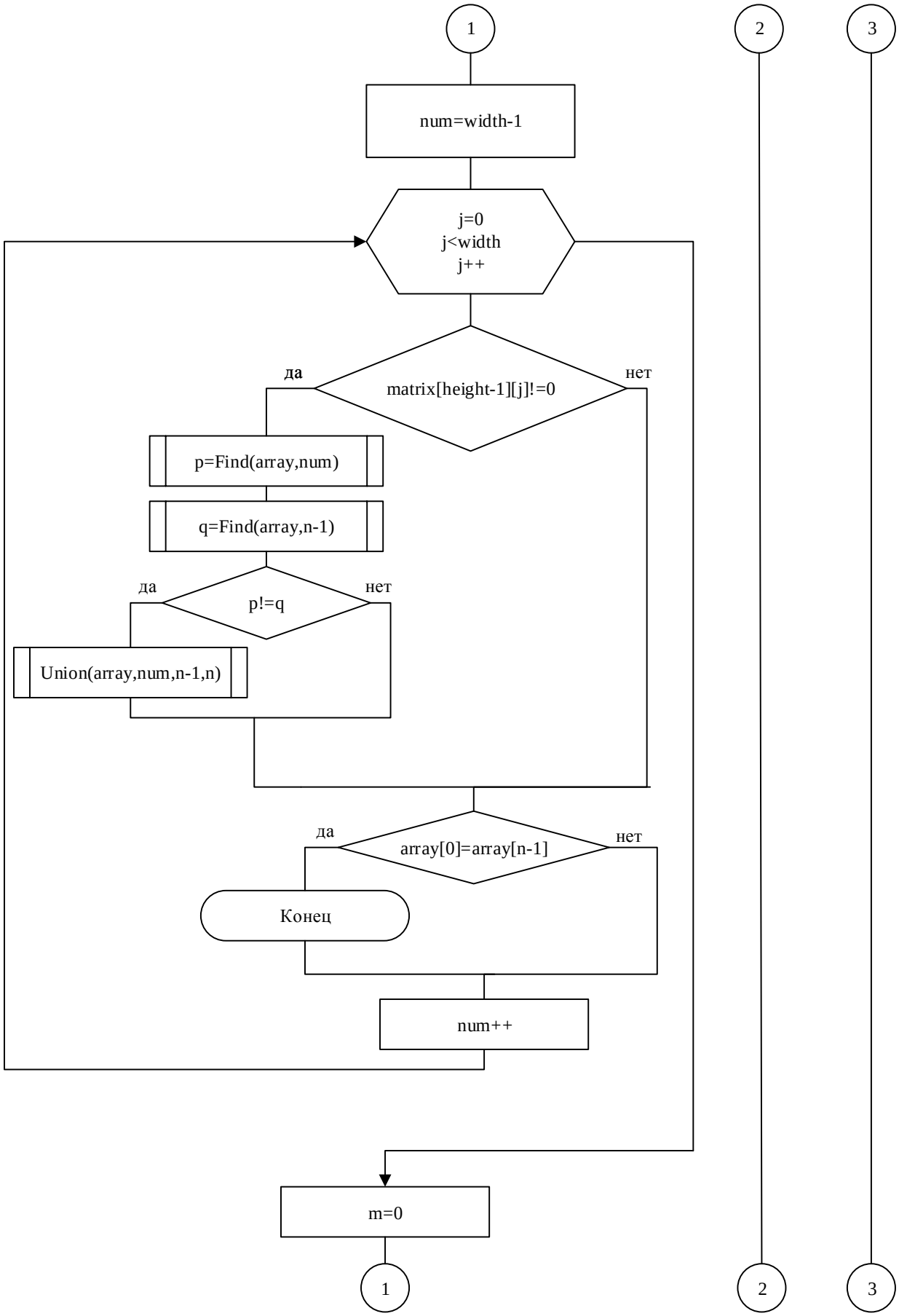
```
    getchar();
    return 0;
}
int Find(int *array,int num)
 {
    while (array[num] != num)
            num =array[num];
    return array[num];
}
void Union(int *array,int p,int q,int n)
 {
    while (array[p] != p)
            p=array[p];
    array[q]=array[p];
    for (int i =0;i < n;i++)
            {
        if (array[i]== q) array[i]=p;
     }
}
```

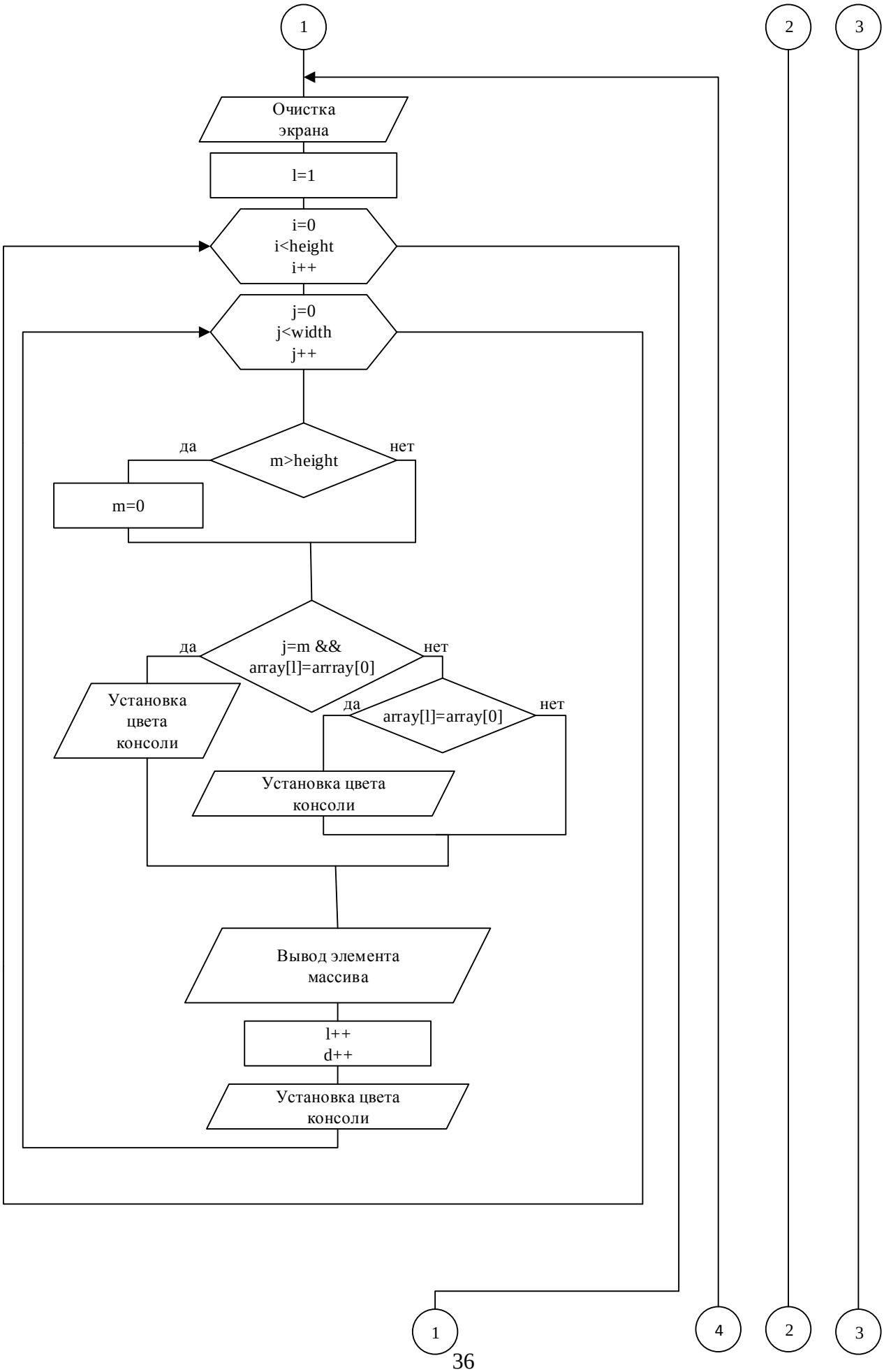
ПРИЛОЖЕНИЕ В
(обязательное)
Графические схемы алгоритмов




1
2
3
i=0
i,n,1
array[i]=i
num=1
j=0
j,width,1
да
matrix[0][j]!=0
нет
p=find(array,0)
q=find(array,num)
да
p!=q
нет
Union(array,0,num,n)
num++
num=1
1
2
3


1
2
3
i=0 i<height-1
i++
j=0
j<width
j++
да
matrix[i][j]!=0
нет
да
j+1<width &&
matrix[i][j+1]!=0
нет
p=Find(array,num)
q=Find(array,num+1)
да
p!=q
нет
Union(array,p,q,n)
i+1<height &&
matrix[i+1][j]!=0
p=Find(array,num)
q=Find(array,num+width)
да
p!=q
нет
Union(array,p,q,n)
num++
1
2
3


1
2
3
num=width-1
j=0
j<width
j++
да
matrix[height-1][j]!=0
нет
p=Find(array,num)
q=Find(array,n-1)
да
p!=q
нет
Union(array,num,n-1,n)
да
array[0]=array[n-1]
нет
Конец
num++
m=0
1
2
3


1
2
3
Очистка экрана
l=1
i=0
i<height
i++
j=0
j<width
j++
да
m>height
нет
m=0
да
j=m &&
array[l]=array[0]
нет
Установка цвета консоли
да
array[l]=array[0]
нет
Установка цвета консоли
Вывод элемента массива
l++
d++
Установка цвета консоли
1
4
2
3

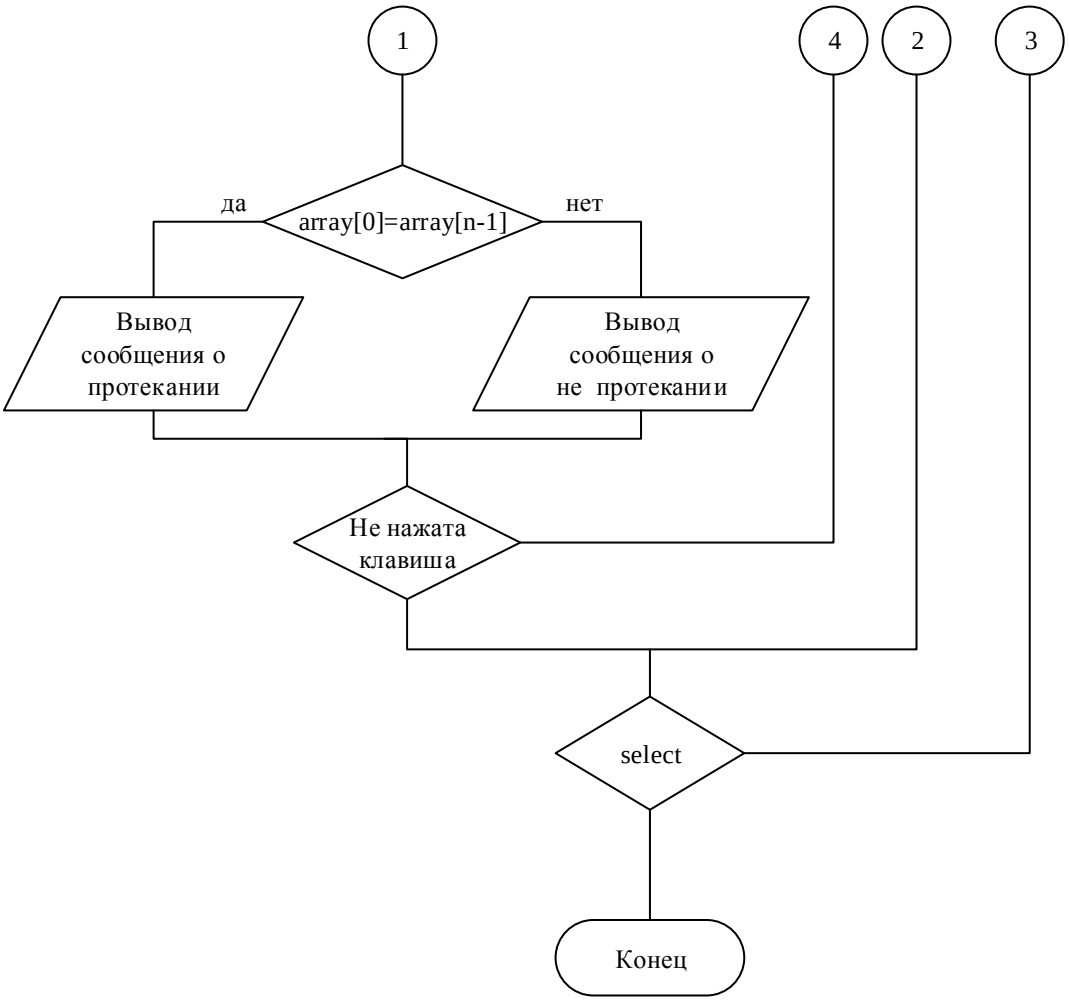


Рисунок В.1 – Графическая схема алгоритма

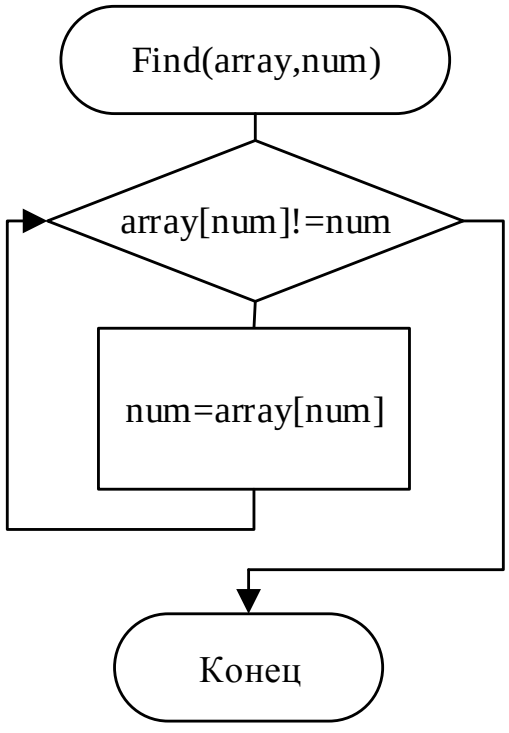


Рисунок В.2 – Графическая схема подпрограммы *Find*

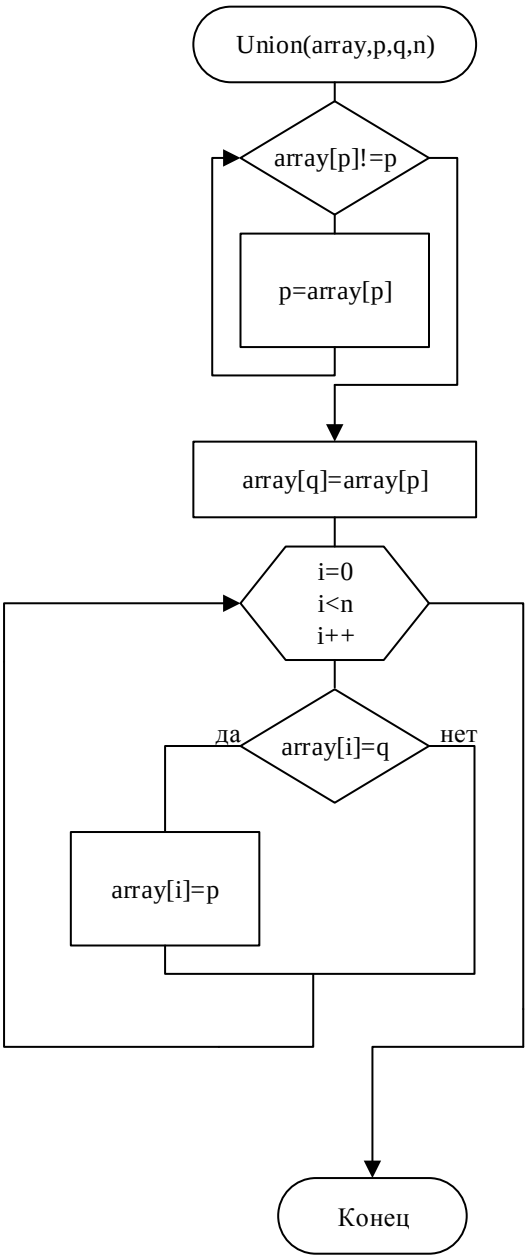


Рисунок В.2 – Графическая схема подпрограммы *Union*