Наука и Образование
МГТУ им. Н.Э. Баумана
Сетевое научное издание
ISSN 1994-0408

Ссылка на статью:
// Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 11. С. 505–527.
DOI: **10.7463/1115.0820770**
Представлена в редакцию: 27.09.2015
Исправлена: 11.11.2015

УДК 004.2+004.31

Исследование вариантов реализации алгоритмов Крускала и Прима в вычислительной системе с многими потоками команд и одним потоком данных

Попов А. Ю.[1,*]

[*]alexpopov@bmstu.ru

[1]МГТУ им. Н.Э. Баумана, Москва, Россия

В МГТУ им. Н.Э. Баумана ведется проект по разработке принципов функционирования вычислительной системы с принципиально новой архитектурой. Разработан действующий образец системы, позволивший оценить эффективность разработанных аппаратных и программных средств. Результаты экспериментов, приведенные в предыдущих работах, а также анализ принципов функционирования новой вычислительной системы, позволяют сделать выводы о ее эффективности при решении задач дискретной оптимизации, связанной с обработкой множеств. Новая архитектура основана на прямой аппаратной поддержке операций дискретной математики, что выражается в применении специальных средств обработки множеств и структур данных. В рамках проекта разработано специальное устройство — Процессор обработки структур (СП), которое повышает производительность, не ограничивая области применения такой вычислительной системы. В предыдущих работах приведены базовые принципы организации вычислительного процесса в МКОД системе, представлена структура и особенности работы Процессора обработки структур, показаны общие принципы решения задач дискретной оптимизации на графах. В данной работе рассматриваются два алгоритма поиска минимального покрывающего дерева: алгоритмы Крускала и Прима. Исследуются варианты реализации алгоритмов для двух режимов работы СП: режима сопроцессора и режима МКОД. Приводятся результаты экспериментального сравнения производительности МКОД системы в режиме сопроцессора с универсальными ЭВМ.

Ключевые слова: граф; много потоков команд и один поток данных; процессор обработки структур; минимальное остовное дерево; алгоритм Крускала; алгоритм Прима

Введение

Обработка множеств дискретных величин является основой большого количества алгоритмов оптимизации, включая алгоритмы на графах и сетях. Множества принято представлять в оперативной памяти ЭВМ в виде векторных структур данных (векторов, матриц) или в виде списковых структур данных (списков, деревьев). Разработаны и активно применяются несколько десятков структур данных, которые позволяют сократить вычислительную сложность алгоритмов [1, 2]. Однако, при реализации таких структур на универсальных ЭВМ

обнаруживается, что временная сложность программ при использовании списковых структур оказывается выше ожидаемой (для ряда операций списковые структуры обрабатываются в несколько десятков раз медленнее, чем векторные). Это объясняется существенными конструктивными особенностями современных микропроцессоров, таких как: конвейерная организация микропроцессоров, виртуализация памяти, расслоение памяти [3, 4, 5]. В настоящий момент активно ведется поиск новых подходов к построению вычислительных машин и систем, обладающих более высоким быстродействием [6, 7, 8, 9, 10]. при сохранении их универсальности.

Принципы работы вычислительной системы МКОД основаны на разделении потоков команд обработки структур данных на два потока: обработки структурной составляющей с помощью процессора обработки структур; арифметико-логической обработки скалярных величин центральным процессором. Оба устройства имеют независимую память для хранения команд и данных, что обеспечивает возможность одновременного выполнения по крайней мере двух потоков команд [11, 12, 13].

В алгоритмах оптимизации, данные в которых представлены в виде множеств дискретных величин, требуется выполнять ряд базовых операций, таких как: кванторы общности, кванторы существования, операции включения и исключения элементов множества и ряд других. На основе таких операций сформирован набор команд процессора обработки структур (СП). СП реализует в системе МКОД набор функций, позволяющий без участия программиста осуществлять хранение и обработку структур данных, представляющих множества алгоритма [14].

• **Реализация операций над структурами данных в соответствии с ключами.** Ключ является основой структурной части информации и позволяет идентифицировать данные. В данном исследовании использован вариант СП с размером ключа 32 бит и аналогичным полем данных. Результаты выполнения команд передаются в ЦП для последующей обработки вычислительным алгоритмом.

• **Управление памятью при хранении информации структур данных.** В универсальных ЭВМ выделение памяти относится к функциям операционной системы и является одной из важнейших задач. Менеджеры памяти операционных систем, в свою очередь, строятся на основе структур данных: красно-чёрных деревьев, очередей и пр. В МКОД-системе выделение и освобождение памяти, используемой при хранении структур данных, выполняются аппаратно при помощи СП. Помимо этого, СП обеспечивает хранение одновременно нескольких структур данных в своей локальной памяти, что позволяет выполнять такие операции, как объединение, пересечение и дополнение.

• **Исполнение управляющих программ обработки структур данных.** СП выполняет код, хранимый в локальной памяти команд в виде программ, которые составлены таким образом, чтобы результаты их исполнения соответствовали ходу вычислительного процесса в ЦП.

• **Синхронизация с вычислительным процессом в ЦП.** СП функционирует под управлением собственной программы, выбираемой им из локальной памяти команд. При выполнении ветвлений в программе ЦП требуется обеспечение синхронизации кода в обоих вычислительных узлах: СП и ЦП. Эта синхронизация выполняется на основе передачи специальных команд перехода, реализующих механизм сложных событий.

1. Обзор предыдущих работ

В [13] определены ключевые понятия в области обработки структур данных, являющихся представлением множеств в памяти ЭВМ, основанные на трудах [1, 2]. Показано, что обработка двух видов информации (данных и их структурных отношений) может вестись независимо, что и использовано в системе, аппаратно реализующей обработку структур данных и множеств.

В [12] приведены результаты исследований принципов построения и области применения ЭВМ с аппаратной поддержкой операция над структурами данных. Показано, что предложенная архитектура относится к классу ЭВМ с многими потоками команд и одним потоком данных. Приведены результаты экспериментальных исследований вариантов реализации процессора обработки структур. В работе [15] приводятся принципы организации вычислительных систем МКОД, предложена схема взаимодействия устройств системы, обеспечивающая параллельное выполнение потоков команд. На примере алгоритма Дейкстры поиска кратчайших путей на графе поясняются особенности разработки программ оптимизации и ход вычислительного процесса в системе МКОД.

В работе [14] описаны основные механизмы доступа к данным, приведены результаты экспериментов измерения производительности процессора обработки структур при выполнении основных операций. Показаны результаты сравнения аппаратной сложности реализации процессора обработки структур с аппаратной сложностью универсальных микропроцессоров, выполняющих аналогичные действия. В [16] приведены форматы команд процессора обработки структур, предложена методика модификации алгоритмов при реализации в МКОД-системе, предложен вариант алгоритма Форда-Фалкерсона поиска максимального потока на графе для МКОД-модели вычислительной системы, предложены варианты модификации архитектуры МКОД-системы, ведущие к повышению ее производительности. В [17] рассмотрена реализация алгоритмов Беллмана — Форда и Ли поиска кратчайшего пути в графе, получена версия алгоритмов поиска кратчайшего пути в режиме сопроцессора, а также параллельная версия в режиме МКОД.

2. Алгоритмы поиска минимального остовного дерева

Пусть дан связный неориентированный граф $G(V, E)$, где V — множество вершин, E — множество ребер. Для каждого ребра (u, v) определен его вес $w(u, v)$. Задача состоит в нахождении связного ациклического подграфа $T \subset G$, содержащего все вершины исходного

графа G, причем вес его ребер минимален. Подграф T не содержит циклов и обладает свойством связности, в связи с чем он называется минимальным остовным деревом (minimum spanning tree).

Алгоритмы Крускала и Прима относятся к категории локально-оптимальных (или так называемых «жадных»), т.е. на каждом шаге производят выбор наилучшего варианта, что, однако, приводит к получению оптимального конечного решения. Алгоритмы могут применяться в задачах планирования в геоинформационных системах, энергетике, телекоммуникациях, при построении сетей передачи данных, проектирование радио и электронной аппаратуры и прочих областях, требующих определения минимального количества ресурсов, необходимых для покрытия всех распределенных в пространстве объектов [1, 2].

3. Алгоритм Крускала

Алгоритм Крускала выполняет поиск подмножества ребер графа, которые образуют дерево на всех его вершинах. При этом суммарный вес всех ребер минимален среди возможных вариантов. Если граф не является связным, то алгоритм находит лес минимальных покрывающих деревьев.

Алгоритм основан на объединении поддеревьев, если их связывает ребро с минимальным весом. Рассмотрим идею алгоритма Крускала (рис. 1).

В начальный момент времени каждая вершина представляет собой отдельное поддерево в лесу остовных деревьев.

1. Множество E содержит все ребра графа.
2. Определяется ребро графа из множества E, которое обладает минимальным весом.
3. Если ребро связывает два различных поддерева, объединяем два поддерева в одно. Ребро удаляется из множества E.
4. Если множество E пусто, конец алгоритма. Иначе переход к п. 2.

Вычислительная сложность работы алгоритма при представлении графа сбалансированным деревом определяется следующим образом. Действие 1 предполагает создание структуры из E ребер, что требует $O(E \log E)$. Выборка ребра с минимальным весом требует $O(\log E)$, а количество итераций (действия 2–4) равно $|E|$. Таким образом, вычислительная сложность алгоритма Крускала равна $O(E \log E)$.

Рассмотрим более подробно структуры данных, обеспечивающие реализацию действий алгоритма. Прежде всего определим способ представления графа. Граф должен быть задан таким образом, чтобы для каждой вершины можно было определить индекс поддерева, в которое она входит. Кроме того, необходимо проводить процедуру объединения поддеревьев, что требует обхода всех входящих в поддерево вершин. Также необходимо реализовать множество ребер, упорядоченное по их весу. Выборка должна производиться для очередного ребра минимального веса. Объединить столь противоречивые требования в одной структуре не представляется возможным. Поэтому реализуем три структуры.

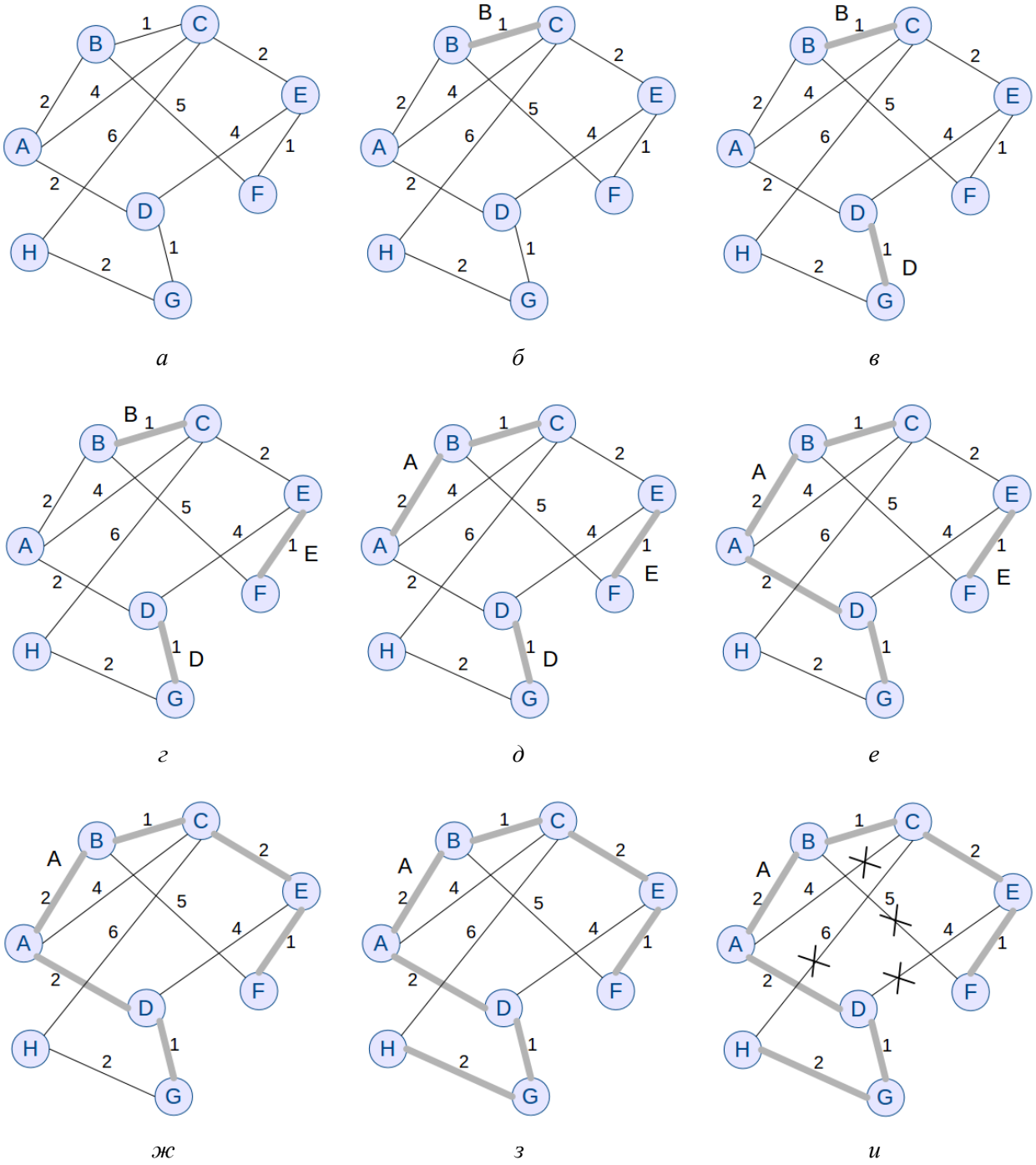


Рис. 1. Пример работы алгоритма Крускала: *а* — исходный граф; *б* — объединение поддеревьев B и C в поддерево B; *в* — объединение поддеревьев D и G в поддерево D; *г* — объединение поддеревьев E и F в поддерево E; *д* — объединение поддеревьев A и B в поддерево A; *е* — объединение поддеревьев A и D в поддерево A; *ж* — объединение поддеревьев A и E в поддерево A; *з* — объединение поддеревьев A и H в поддерево A; *и* — просмотр оставшихся ребер, образующих циклы

Структура T для хранения леса поддеревьев содержит списки ребер для каждого поддерева. В начальный момент времени структура T содержит поля описания поддеревьев (так называемые индексные записи). Для записи поддерева используется номера вершин, образующих ребро в поддереве. Так как в начальный момент в поддеревьях содержится только

по одной вершине, все индексы равны единице. Записи с нулевым индексом используются для хранения количества вершин в поддереве: T.KEY $= (subtree, index)$; для индексной записи T.KEY $= (subtree, 0)$ поле T.DATA $= (index_{\max})$; для всех остальных ненулевых индексов поле данных содержит номера вершин: T.DATA $= (u, v)$. После выполнения алгоритма структура T представляет результат работы алгоритма, так как по данной структуре определяется количество поддеревьев и принадлежность им вершин.

Структура G хранит множество вершин и номера их поддеревьев: G.KEY $= (v)$, где v — номер вершины; G.DATA $= (subtree)$, где $subtree$ — номер поддерева. В начальный момент работы алгоритма для всех вершин $subtree = v$. Для хранения множества ребер необходимо использовать структуру E с составным ключом, в которой вершины с меньшим весом ребер имеют заведомо меньший ключ, т.е. поле веса ребра w должен находится в старшей части ключа. При этом ключ должны быть уникальными, поэтому необходимо включить в ключ также и индексы вершин u, v. Таким образом, ключ в структуре E представляется следующим образом: E.$KEY = (w, u, v)$. При этом поле данных в структуре E не используется. Возможны два варианта выборки ребер из структуры E в порядке увеличения их весом: извлечение минимального элемента и его удаление; обход структур от элемента с минимальным ключом до элемента с максимальным ключом.

Алгоритм 1. АЛГОРИТМ КРУСКАЛА (G,E) /псевдокод алгоритма в ОКОД-системе/

1: $v \leftarrow$ MIN(G) ▷ Перебор вершин графа и создание структуры T
2: **ЦИКЛ ПОКА** $v \neq \varnothing$
3: INSERT(T, (v.KEY, 0), (0)) ▷ Вставка индексной записи
4: $v \leftarrow$ NEXT(G)
5: **ВСЕ ЦИКЛ ПОКА**
6: $e \leftarrow$ MIN(E) ▷ Перебор ребер графа в порядке увеличения весов
7: **ЦИКЛ ПОКА** $e \neq \varnothing$
8: $u \leftarrow$ SEARCH(G, e.KEY.u) ▷ Чтение информации о вершине u
9: $v \leftarrow$ SEARCH(G, e.KEY.v) ▷ Чтение информации о вершине v
10: **ЕСЛИ** u.DATA.$subtree \neq v$.DATA.$subtree$ **ТО**
11: $t_v \leftarrow$ SEARCH(T, (v.DATA.$subtree$, 0)) ▷ Чтение индексной записи поддерева вершины v
12: $index_v \leftarrow t_v$.DATA.$index_{\max}$ ▷ Определение количества ребер в поддереве вершины v
13: $t_u \leftarrow$ SEARCH(T, (u.DATA.$subtree$, 0)) ▷ Чтение индексной записи поддерева вершины u
14: $index_u \leftarrow t_u$.DATA.$index_{\max}$ ▷ Определение количества ребер в поддереве вершины u
15: **ЦИКЛ** $i = 1$ to $index_v$ ▷ Перенос всех ребер поддерева вершины v
16: ▷ в поддерево вершины u
17: $index_u \leftarrow index_u + 1$ ▷ Определяем индекс ребра
18: $temp \leftarrow$ SEARCH(T, (v.DATA.$subtree$, i)) ▷ Определяем номер вершины
19: INSERT(G, ($temp$.DATA.u), (u.DATA.$subtree$)) ▷ Меняем номер поддерева в G
20: INSERT(G, ($temp$.DATA.v), (u.DATA.$subtree$))
21: DELETE(T, (v.DATA.$subtree$, i)) ▷ Удаляем ребро из поддерева вершины v

22: INSERT(T, $(u$.DATA.$subtree, index_u)$, $(temp$.DATA)) ▷ Перенос вершины u
23: **ВСЕ ЦИКЛ**
24: INSERT(G, $(e$.KEY.$v)$, $(u$.DATA.$subtree))$ ▷ Меняем номер поддерева в G
25: DELETE(T, $(v$.DATA.$subtree, 0))$ ▷ Удаляем входную вершину поддерева вершины v
26: INSERT(T, $(u$.DATA.$subtree, index_u + 1)$, $(e$.KEY.$u, e.KEY.v))$ ▷ Добавляем ребро (u, v)
27: INSERT(T, $(u$.DATA.$subtree, 0)$, $(index_u + 1))$ ▷ Меняем информацию в индексной
28: ▷ записи
29: **ВСЕ ЕСЛИ**
30: $e \leftarrow$ NEXT(E) ▷ Переходим к следующему ребру
31: **ВСЕ ЦИКЛ ПОКА**
32: **КОНЕЦ** ▷ Все ребра рассмотрены

Алгоритм получает на вход структуры, описывающие граф: структура G для хранения номеров вершин и их поддеревьев; структура E, хранящая ребра. В строках 1–5 происходит инициализация структуры T: для каждой вершины из структуры G добавляется входная вершина поддерева: INSERT(T, $(v$.KEY$, 0)$, (0), где v.KEY$, 0$ — ключ с номером вершины и индексом. В начальный момент в поле данных записывается единица.

Основной цикл алгоритма в строках 7–31 построен на выборке очередного ребра e с минимальным весом из оставшихся вершин. По полученной информации о ребре читается информация о вершинах (u, v) в строках 8 и 9. Если в строке 10 оказывается, что вершины находятся в разных поддеревьях, то происходит объединение двух поддеревьев, номера которых соответственно равны: u.DATA.$subtree$ и v.DATA.$subtree$. Для этого все ребра дерева вершины v включаются в дерево вершины u. Для этого определяется количество ребер в поддеревьях и происходит выборка очередного ребра из поддерева вершины v (строка 17) в структуре T, после чего ребро удаляется и добавляется уже в поддерево вершины u (строка 21). В строках 18 и 19 и 23 происходит соответствующее изменение в структуре G: для каждой переносимой вершины меняется номер поддерева; номер поддерева меняется для вершины, указанной в ребре e. В строке 24 происходит удаление входной вершины поддерева вершины v, а соответствующая входная вершина объединенного поддерева модифицируется (строка 26). В строке 25 происходит добавление ребра, связывающего объединенные поддеревья. После объединения двух поддеревьев рассматривается следующее ребро.

Рассмотрим модификацию алгоритма Крускала для МКОД-системы.

Алгоритм 2. АЛГОРИТМ КРУСКАЛА (G,E) /псевдокод алгоритма в МКОД-системе/

1: –Поток ЦП– ▷ –Поток СП–
2: GET(v) ▷ MIN(G)*
3: **ЦИКЛ ПОКА** $v \neq \varnothing$
4: PUT$(v$.KEY$, 0)$ ▷ INSERT(T,?,0)
5: GET(v) ▷ NEXT(G)*
6: **ВСЕ ЦИКЛ ПОКА**
7: GET(e) ▷ MIN(E)*

```
8:  ЦИКЛ ПОКА e ≠ ∅
9:      PUT(0)                                              ▷ @1:JT(?,@2)
10:     PUT(e.KEY.u)
11:     GET(u)                                              ▷ SEARCH(G,?)*
12:     PUT(e.KEY.v)
13:     GET(v)                                              ▷ SEARCH(G,?)*
14:     ЕСЛИ u.DATA.subtree ≠ v.DATA.subtree ТО
15:         PUT(0)                                          ▷ JT(?,@3)
16:         PUT(v.DATA.subtree, 0)
17:         GET(t_v)                                        ▷ SEARCH(T,?)*
18:         index_v ← t_v.DATA.index_max
19:         PUT(u.DATA.subtree, 0)
20:         GET(t_u)                                        ▷ SEARCH(T,?)*
21:         index_u ← t_u.DATA.index_max
22:         ЦИКЛ i = 1 to index_v
23:             PUT(0)                                      ▷ @4:JT(?,@5)
24:             index_u ← index_u + 1
25:             PUT(v.DATA.subtree, i)
26:             GET(temp)                                   ▷ SEARCH(T,?)
27:             PUT(temp.DATA.u)
28:             PUT(u.DATA.subtree)                         ▷ INSERT(G,?,?)*
29:             PUT(temp.DATA.v)
30:             PUT(u.DATA.subtree)                         ▷ INSERT(G,?,?)*
31:             PUT(v.DATA.subtree, i)                      ▷ DELETE(T,?)*
32:             PUT(u.DATA.subtree, index_u)
33:             PUT(temp.DATA)                              ▷ INSERT(T,?,?)
34:         ВСЕ ЦИКЛ                                        ▷ JT(1,@4)*
35:         PUT(e.KEY.v)
36:         PUT(u.DATA.subtree)                             ▷ @5:INSERT(G,?,?)*
37:         PUT(v.DATA.subtree, 0)                          ▷ DELETE(T,?)*
38:         PUT(u.DATA.subtree, index_u + 1)
39:         PUT(e.KEY.u, e.KEY.v)                           ▷ INSERT(T,?,?)
40:         PUT(u.DATA.subtree, 0)
41:         PUT(index_u + 1)                                ▷ INSERT(T,?,?)*
42:     ИНАЧЕ
43:         PUT(1)
44:     ВСЕ ЕСЛИ
45:     GET(e)                                              ▷ @3:NEXT(E)*
46: ВСЕ ЦИКЛ ПОКА                                           ▷ JT(1,@1)
47: PUT(1)                                                  ▷ @2:
48: КОНЕЦ
```

Представленный вариант алгоритма повторяет ход алгоритма для ОКОД-системы: в строках 3-6 представлен цикл инициализации структуры T; с 8 по 44 строку реализуется основной цикл. Знаком (*) в потоке команд СП отмечены команды, все операнды которых либо являются константными, либо могут быть получены из результатов выполнения предыдущих команд СП. Таким образом, эти команды могут быть запущены на исполнение без ожидания операндов от потока ЦП. Всего в алгоритме использованы 18 команд обработки структур данных, при этом 14 команд (78%) являются независимыми. Данный результат показывает высокий потенциал для ускорения алгоритмов в МКОД-системе.

4. Алгоритм Прима

Идея алгоритма Прима состоит в добавлении к частично построенному остовному дереву исходящего из него ребра, причем его вес минимален среди всех подобных ребер. В начальный момент времени остовное дерево состоит из одной вершины, которая выбирается в том ребре, вес которого наименьший среди всех ребер графа. Исходный граф задается следующими множествами: множество G содержит инциденции вершин ребрам графа; множество Q содержит вершины графа. Алгоритм можно представить в виде следующих действий.

1. По G определяется ребро (u, v), которое обладает минимальным весом.
2. Исключаем u из множества Q.
3. Для каждой вершины в Q определяем вес ребра $w[T]$, связывающего его с вершиной u. Если таких ребер нет, то связность принимаем равной ∞. Полученную связность записываем в множество E.
4. По множеству E определяем ребро (u, v) и его вершину v, принадлежащую множеству Q и обладающую минимальной связностью с вершинами остовного дерева T.
5. Удаляем из E все ребра, связывающие вершину v с остовным деревом T.
6. Исключаем v из множества Q. Добавляем ребро (u, v) в остовное дерево T.
7. Для каждой вершины из Q, смежной v, определяем новое значение $w[T]$.
8. Если Q пусто, то конец алгоритма, иначе переход к п. 4.

В отличие от алгоритма Крускала, алгоритм Прима позволяет найти остовное дерево только в связном графе. В случае, если граф не является связным, в п. 4 будет выбрана вершина с бесконечной связностью с первоначальным остовным деревом и алгоритм продолжит свою работу. Но, так как это ребро не обладает минимальным весом из оставшихся, корректность алгоритма будет нарушена. Поэтому, для связного графа следует модифицировать алгоритм таким образом, чтобы при выборке вершины с бесконечной связностью происходил перезапуск всего алгоритма для оставшихся связных компонент графа. В нашем случае будем рассматривать вариант приложения алгоритма Прима к связному графу.

На рис. 2 показан пример построения остовного дерева алгоритмом Прима. На рис. 2, *а* показан исходный граф. Веса ребер указаны в соответствии с вершинами. На рис. 2, *б*

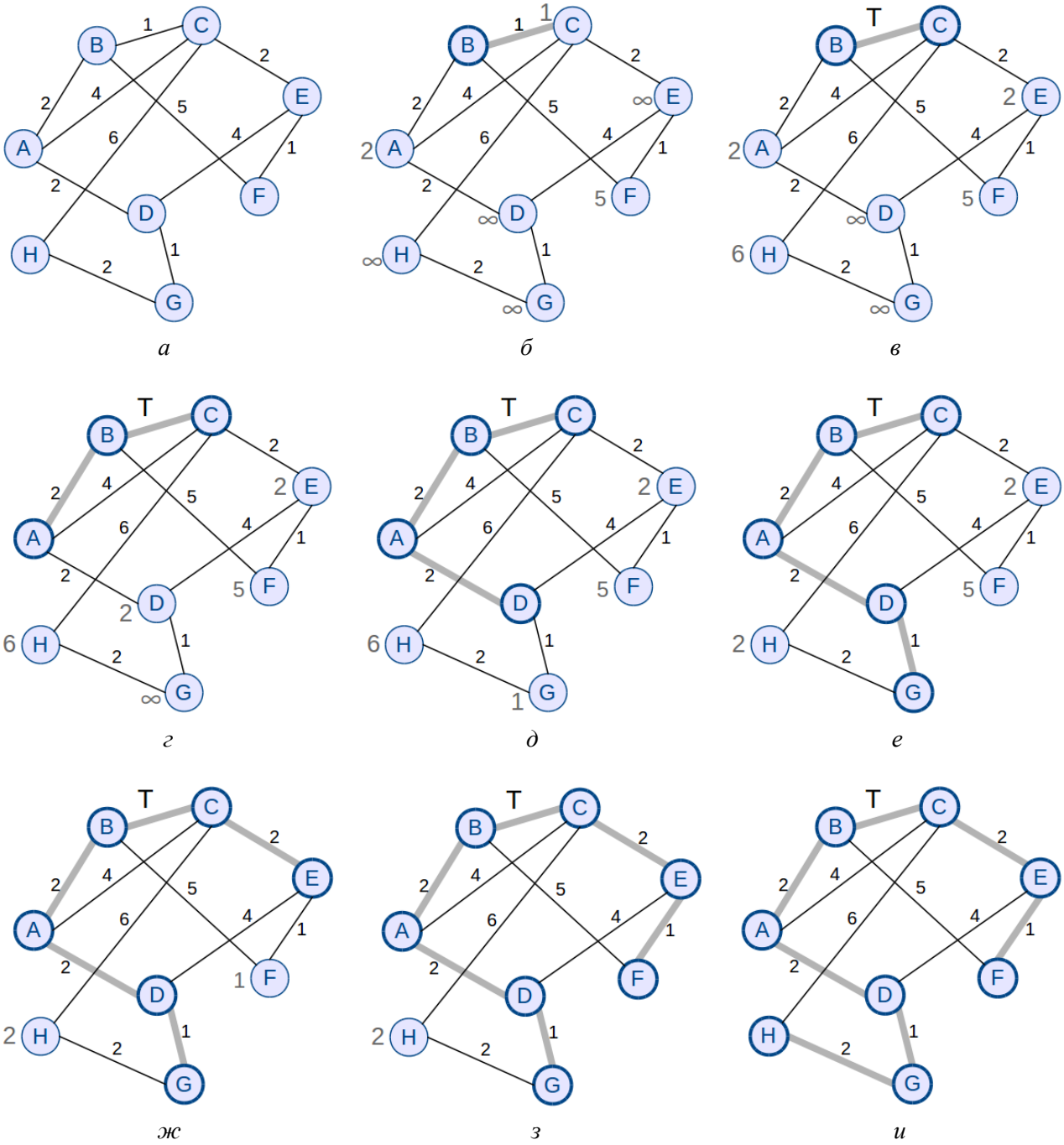


Рис. 2. Пример работы алгоритма Прима: *а* — исходный граф; *б* — выбор ребра с минимальным весом и определение $w[T]$; *в* — включение ребра BC в остовное дерево T и пересчет связности для вершин E, H; *г* — включение ребра AB в остовное дерево T и пересчет связности для вершины D; *д* — включение ребра AD в остовное дерево T и пересчет связности для вершины G; *е* — включение ребра DG в остовное дерево T; *ж* — включение ребра CE в остовное дерево T и пересчет связности для вершин F; *з* — включение ребра EF в остовное дерево T; *и* — включение ребра GH в остовное дерево T и конец работы алгоритма

показан результат выполнения пунктов 1–3: найдено ребро наименьшего веса BC и выбрана первая вершина графа. Также для каждой вершины определены значения $w[T]$, которые показаны слева от вершин. Если таких ребер нет, то для вершины указана бесконечная связность (∞).

На рис. 2, *в* показан результат выполнения первой итерации. Выбрано ребро BC с весом 1, определена вершина C для включения в остовное дерево, выполнен пересчет $w[T]$ для вершин E, H. Ребро BC включено в T (остовное дерево показано на рисунке серым). На следующей итерации (рисунок 2, *г*) определяется ребро AB и, соответственно, вершина A включается в T и пересчитывается показатель $w[T]$ для вершины D (если имеется несколько ребер минимального веса, выбор происходит в соответствии с алфавитным порядком вершин). Далее выбирается ребро AD с весом 2 и включаем вершину D в T (рис. 2, *д*). При этом изменяется связность вершины G остовным деревом T. Следующее ребро с минимальным $w[T]$ — ребро DG (рис. 2, *е*). Далее происходит выбор ребра CE и пересчет связности для вершины F на значение 1 (результат показан на рис. 2, *ж*). Соответственно, ребро включается в остовное дерево на следующем шаге алгоритма (рис. 2, *з*). На последней итерации добавляется ребро GH и, так как все вершины находятся в остовном дереве, алгоритм заканчивает свою работу.

Асимптотика вычислительной сложности алгоритма Прима зависит от используемых структур данных для хранения графа и остовного дерева. При использовании сбалансированных деревьев вычислительная сложность поиска ребра с минимальным весом (действие 4) требует $O(\log n)$ операций. Удаление записи из множества Q также требует $O(\log n)$ операций. Удаление ребер в действии 5 производится не более чем $|E|$ раз, а итерационный цикл повторяется не более $|V|$ раз. В связи с этим оценка вычислительной сложности алгоритма при представлении структур данных в виде деревьев составляет $O((V + E) \log n)$.

Выполним анализ действия алгоритма, определим состав и назначение структур данных. В первую очередь следует выбрать способ представления графа в структуре G. В пункте 5 необходимо определить все инцидентные вершине ребра, что невозможно сделать с помощью структуры E. Для этих целей необходимо хранить списки смежных вершин в соответствии с первым вариантом представления графа [16]: поле G.KEY хранит номера вершин u и порядковый номер ребра (G.KEY $= (u, \text{count})$); поле данных G.DATA хранит номер инцидентной вершины v и вес ребра w (G.DATA $= (v, w)$). Такой вариант ключа структуры G позволяет также определить списки смежных вершин в пункте 7. Следует также отметить, что граф является неориентированным. Поэтому одно и то же ребро указывается дважды как для вершины u, так и для вершины v. Это обстоятельство нужно учесть при обработке ребер.

В пунктах 4 и 7 выполняется проверка на вхождение вершины в множество Q. Поэтому ключом для Q может являться номер вершины: Q.KEY $= (u)$. В пункте 7 требуется определить новое значение $w[T]$, связывающее вершину с остовным деревом. Если в остовное дерево попадает новая вершина, то необходимо пересчитать связность $w[T]$ всех смежных с ней вершин. Если вес ребра w, связывающий смежную вершину и добавляемую в остовное дерево вершину меньше $w[T]$, необходимо его изменить на величину w. Для реализации такого действия нужно хранить старое значение $w[T]$ в соответствии с номером вершины. Поэтому будем использовать поле Q.DATA для для хранения этого значения. Таким образом Q.DATA $= (w[T])$.

Структура T используется в качестве результата и должна содержать те ребра, которые были найдены в структуре E в ходе итераций. Структуру остовного дерева можно сохранить, если для каждой вершины сохранить ребро с минимальным весом, которое использовалось в алгоритме для включения этой вершины. Поэтому выберем в качестве ключа номера связывающих вершин T.KEY $= (u)$, а поле данных будет содержать вторую вершину ребра T.DATA $= (v)$.

В пункте 4 требуется определить ребро, обладающее минимальной связностью с остовным деревом. Для этого требуется хранить в структуре E информацию о ребрах и выбирать их упорядоченно в порядке увеличения веса. Поэтому выберем третий вариант представления графа [16], при котором ключ задается полями $w[T]$, u, v (E.KEY $= (w[T], u, v)$), а поле данных не имеет значения (E.DATA $= (\varnothing)$). Однако выбор начальной вершины в пункте 1 не может осуществляться на основе структуры E, так как в начальный момент не известна начальная вершина и, соответственно, веса $w[T]$. Поэтому поиск начальной вершины осуществляется вместе с инициализацией структур Q и E.

Рассмотрим подробнее реализацию алгоритма Прима с учетом предложенных структур.

Алгоритм 3. АЛГОРИТМ ПРИМА (G) /псевдокод алгоритма в ОКОД-системе/

1: $v \leftarrow$ MIN(G) ▷ Перебор вершин графа и создание структуры Q
2: $min\_edge\_wight \leftarrow \infty$ ▷ Ищем начальную вершину в ребре с минимальным весом
3: **ЦИКЛ ПОКА** $v \neq \varnothing$ ▷ Перебор всех вершин графа
4: **ЕСЛИ** v.KEY.count $== 0$ **ТО** ▷ Если запись содержит информацию о вершине
5: INSERT(Q, $(v.KEY.u)$, (∞)) ▷ Вставка вершины u в структуру Q
6: **ИНАЧЕ** ▷ Если запись содержит информацию о вершине
7: **ЕСЛИ** v.DATA.$w < min\_edge\_weight$ **ТО**
8: $min\_edge \leftarrow v$.KEY.u ▷ Сохраняем номер начальной вершины
9: $min\_edge\_weight \leftarrow v$.DATA.$w$
10: **ВСЕ ЕСЛИ**
11: **ВСЕ ЕСЛИ**
12: $v \leftarrow$ NEXT(G)
13: **ВСЕ ЦИКЛ ПОКА**
14: $v \leftarrow$ MIN(G) ▷ Создание структуры E
15: **ЦИКЛ ПОКА** $v \neq \varnothing$ ▷ Перебор всех вершин графа
16: **ЕСЛИ** $(v$.KEY.count! $= 0)\&\&(v$.DATA.$u! = min\_edge)$ **ТО** ▷ Для каждого ребра
17: **ЕСЛИ** v.DATA.$v == min\_edge$ **ТО** ▷ Если вершина связана с изначальной
18: INSERT(E, $(v$.DATA.w, v.KEY.u, v.DATA.$v)$, $(\varnothing)$) ▷ Выполняем вставку ребра с полем w
19: INSERT(Q, $(v$.KEY.$u)$, $(v.DATA.w)$) ▷ Сохраняем значение $w[T]$
20: **ИНАЧЕ**
21: INSERT(E, $(\infty, v$.KEY.u, v.DATA.$v)$, $(\varnothing)$) ▷ Выполняем вставку ребра с полем $w[T] = \infty$

22: **ВСЕ ЕСЛИ**
23: **ВСЕ ЕСЛИ**
24: $v \leftarrow$ NEXT(G)
25: **ВСЕ ЦИКЛ ПОКА**
26: DELETE(Q, (min_edge)) ▷ Удаляем из Q начальную вершину
27: **ПОВТОРЯТЬ**
28: $e \leftarrow$ MIN(E) ▷ Перебор ребер графа в порядке увеличения весов
29: $new \leftarrow$ SEARCH(G, $(e$.KEY.$u, 0))$ ▷ Выбираем вершину u
30: $count \leftarrow new$.DATA.count ▷ Определяем количество инцидентных вершине new ребер
31: **ЦИКЛ** i=1 to count
32: $tmp \leftarrow$ SEARCH(G, $(e$.KEY.$u, i))$ ▷ Определяем очередную смежную с new вершину
33: DELETE(E, $(e$.KEY.$w[T], e$.KEY.u, tmp.DATA.$v))$ ▷ Удаляем ребра вершины new
34: $q \leftarrow$ SEARCH(Q, $(tmp$.DATA.$v))$ ▷ Читаем информацию о смежной вершине
35: **ЕСЛИ** $q! = \varnothing$ **ТО** ▷ Если смежная вершина не в остовном дереве
36: **ЕСЛИ** tmp.DATA.$w < q$.DATA.$w[T]$ **ТО** ▷ Определяем связность с остовным деревом
37: DELETE(E, $(q$.DATA.$w[T], tmp$.DATA.v, e.KEY.$u))$ ▷ Меняем связность
38: INSERT(E, $(tmp$.DATA.w, tmp.DATA.v, e.KEY.$u), (\varnothing))$
39: INSERT(Q, $(tmp$.DATA.$v), (tmp$.DATA.$w))$ ▷ Меняем связность в Q
40: **ВСЕ ЕСЛИ**
41: **ВСЕ ЕСЛИ**
42: **ВСЕ ЦИКЛ**
43: INSERT(T, $(e$.KEY.$u), (e$.KEY.$v))$ ▷ Включаем ребро в T
44: DELETE(Q, $(e$.KEY.$u))$ ▷ Удаляем вершину u из Q
45: **ЦИКЛ ПОКА** POWER(Q) $== \varnothing$
46: **КОНЕЦ** ▷ Все ребра рассмотрены

На вход алгоритма подается исходная структура G, описывающая множество инциденций вершин ребрам графа. По этой структуре необходимо сформировать множество вершин Q, упорядоченное множество ребер E, а в ходе алгоритма определяется структура остовного дерева T. Все вершины, включенные в T должны автоматически удаляться из Q. Для этого происходит обход структуры G: в строке 1 выбирается первый элемент, в строках 3–13 происходит обработка очередной записи из G, в строке 12 читается следующая запись. В строке 4 происходит проверка, является ли текущая запись индексной (v.KEY.count $== 0$). В этом случае в строке 5 инициализируется новая вершина в структуре Q: поле Q.KEY $= (v.KEY.u)$, т.е. номер вершины; поле $w[T]$ связности с остовным деревом равно максимальному значению Q.DATA $= (\infty)$, так как остовное дерево не инициализировано.

Одновременно с инициализацией структуры Q происходит поиск ребра минимального веса: в строке 2 инициализируется переменная минимального веса, а при чтении записи о ребре (v.KEY.count! $= 0$) в строке 7 проверяется, обладает ли очередное ребро меньшим

весом среди рассмотренных. Если это так, то в строке 8 запоминается номер вершины u (фактически, любой из двух вершин), а минимальный вес изменяется в строке 9.

После того, как определено ребро с минимальным весом (ребро, исходящее из вершины min_edge, вес ребра равен min_edge_wight), происходит инициализация структуры E. Для этого выполняется повторный обход структуры G в строках 14–25. Если прочтенная из G запись не является индексной, т.е. содержит информацию о ребре (v.KEY.count! $= 0$), то в строке 17 производится проверка, связано ли это ребро с вершиной min_edge. Если да, то в строке 18 выполняется вставка новой записи о ребре в структуру E, где в поле $w[T]$ записывается вес найденного ребра v.DATA.w. Это значение также записывается в структуру Q в строке 19. Если же ребро не связано с начальной вершиной, то в строке 21 в структуру E записывается значение ∞. Завершается инициализация в строке 26, где происходит удаление начальной вершины из множества Q. Следует отметить, что так как граф не является ориентированным, запись о ребре присутствует в структуре G и структуре E для обеих вершин u и v. Это следует учитывать в дальнейших действиях алгоритма. Например, в строке 17 выполнялась проверка на то, является ли вторая вершина ребра начальной. Если это так, то добавление ребра в E происходит. Однако для второй записи того же ребра при обратном порядке вершин условие в строке 17 не выполняется и запись в структуру E не добавляется. Таким образом в E отсутствуют записи о ребрах, исходящих из остовного дерева.

Основной цикл алгоритма выполняется в строках 27-45. Цикл состоит в выборке из структуры E очередного ребра с минимальным весом (строка 28). Так как в ключе структуры E в старшей части использовано значение $w[T]$, запись с минимальным ключом обладает минимальным значением связности с остовным деревом. После этого в переменную new происходит чтение информации о вершине ребра, не входящей в остовное дерево. Так как на остовное дерево указываю только ребра, в которых вершина остовного дерева является второй, искомая вершина определяется как e.KEY.u. Для этой вершины читается информация из структуры G (строка 29).

В строках 31–42 происходил перебор всех вершин, смежных с вершиной new. Для этого используется структура G, по которой определяется количество смежных вершин count (строка 30). Далее в счетном цикле перебирается каждая связанная с new вершина. Все ребра, исходящие из new удаляются в строке 33, что сохраняет условие, при котором в структуре E вершины остовного дерева не имеют ребер. Если смежная вершина не принадлежит остовному дереву (строка 35), выполняется расчет нового значения связности: если вес ребра с вершиной new меньше $w[T]$, значение $w[T]$ заменяется весом ребра (строка 37 и 38). Изменения вносятся и в поле данных структуры Q (строка 39). В строках 43 и 44 вершина удаляется из Q, а найденное ребро включается в остовное дерево.

Далее модифицируем представленный алгоритм для работы в МКОД-системе.

Рассмотрим модификацию алгоритма Прима для МКОД-системы.

Алгоритм 4. АЛГОРИТМ ПРИМА (G,E) /псевдокод алгоритма в МКОД-системе/

```
1: –Поток ЦП–                                                    ▷ –Поток СП–
2: GET(v)                                                        ▷ MIN(G)*
3: min_edge_wight ← ∞
4: ЦИКЛ ПОКА v ≠ ∅
5:     PUT(0)                                                    ▷ @1:JT(?,@2)
6:     ЕСЛИ v.KEY.count == 0 ТО
7:         PUT(0)                                                ▷ JT(?,@3)
8:         PUT(v.KEY.u)                                          ▷ INSERT(Q,?,∞)*
9:     ИНАЧЕ
10:        PUT(1)
11:        ЕСЛИ v.DATA.w < min_edge_weight ТО
12:            min_edge ← v.KEY.u
13:            min_edge_weight ← v.DATA.w
14:        ВСЕ ЕСЛИ
15:    ВСЕ ЕСЛИ
16:    GET(v)                                                    ▷ @3:NEXT(G)*
17: ВСЕ ЦИКЛ ПОКА                                                ▷ JT(1,@1)*
18: PUT(1)
19: GET(v)                                                       ▷ @2:MIN(G)*
20: ЦИКЛ ПОКА v ≠ ∅
21:    PUT(0)                                                    ▷ @4:JT(?,@5)
22:    ЕСЛИ (v.KEY.count! = 0)&&(v.DATA.u! = min_edge) ТО
23:        PUT(0)                                                ▷ JT(?,@6)
24:        ЕСЛИ v.DATA.v == min_edge ТО
25:            PUT(0)                                            ▷ JT(?,@7)
26:            PUT((v.DATA.w, v.KEY.u, v.DATA.v))                ▷ INSERT(E,?,∅)*
27:            PUT(v.KEY.u)
28:            PUT(v.DATA.w)                                     ▷ INSERT(Q,?,?)*
29:                                                              ▷ JT(1,@6)*
30:        ИНАЧЕ
31:            PUT(1)
32:            PUT(∞, v.KEY.u, v.DATA.v)                         ▷ @7:INSERT(E,?,∅)*
33:        ВСЕ ЕСЛИ
34:    ИНАЧЕ
35:        PUT(1)
36:    ВСЕ ЕСЛИ
37:    GET(v)                                                    ▷ @6:NEXT(G)*
38: ВСЕ ЦИКЛ ПОКА                                                ▷ JT(1,@4)*
39: PUT(1)
40:                                                              ▷ @5: DELETE(Q,(min_edge)
```

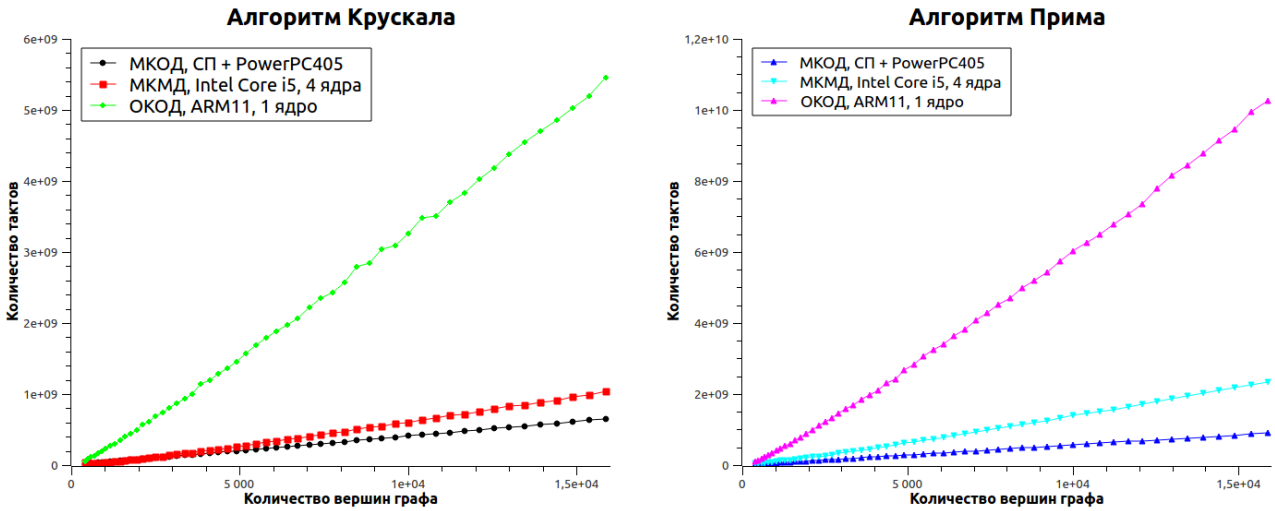
```
41: ПОВТОРЯТЬ
42:     PUT(1)                                                    ▷ JT(?,@8)
43:     GET(e)                                                    ▷ @8:MIN(E)*
44:     PUT(e.KEY.u)
45:     GET(new)                                                  ▷ SEARCH(G,?)*
46:     count ← new.DATA.count
47:     ЦИКЛ i = 1tocount
48:         PUT(0)                                                ▷ JT(?,@10)
49:         PUT(e.KEY.u, i)
50:         GET(tmp)                                              ▷ @9:SEARCH(G,?)*
51:         PUT(e.KEY.w[T], e.KEY.u, tmp.DATA.v)                  ▷ DELETE(E,?)*
52:         PUT(tmp.DATA.v)
53:         GET(q)                                                ▷ SEARCH(Q,?)*
54:         ЕСЛИ q! = ∅ ТО
55:             PUT(0)                                            ▷ JT(?,@11)
56:             ЕСЛИ tmp.DATA.w < q.DATA.w[T] ТО
57:                 PUT(0)                                        ▷ JT(?,@11)
58:                 PUT(q.DATA.w[T], tmp.DATA.v, e.KEY.u)         ▷ DELETE(E,?)*
59:                 PUT(tmp.DATA.w, tmp.DATA.v, e.KEY.u)          ▷ INSERT(E,?,∅)*
60:                 PUT(tmp.DATA.v)
61:                 PUT(tmp.DATA.w)                               ▷ INSERT(Q,?,?)*
62:             ИНАЧЕ
63:                 PUT(1)
64:             ВСЕ ЕСЛИ
65:         ИНАЧЕ
66:             PUT(1)
67:         ВСЕ ЕСЛИ
68:     ВСЕ ЦИКЛ                                                  ▷ @11:JT(1,@9)
69:     PUT(1)
70:     PUT(e.KEY.u)
71:     PUT(e.KEY.v)                                              ▷ @10:INSERT(T,?,?)*
72:     PUT(e.KEY.u)                                              ▷ DELETE(Q,?)*
73: ЦИКЛ ПОКА GET() == ∅                                          ▷ POWER(Q)*
74: PUT(0)                                                        ▷ JT(?,@8)
```

Псевдокод алгоритма получен из представленного алгоритма Прима для ОКОД-системы с использованием замены структурных конструкций без использования непосредственных адресов. Поток команд процессора обработки структур представлен в правой части в соответствии с командами пересылки операндов PUT() и командами выборки результатов GET(). Команды СП, отмеченные знаком (*), являются независимыми, т.е. используют повторно загруженные операнды или результаты предыдущих команд. Таким образом, при выделении

регистров и генерации составных ключей количество независимых команд обработки в алгоритме составляет 19 (95%) из общего количества команд (всего в алгоритме используется 20 команд обработки структур).

5. Исследование производительности Процессора обработки структур

Разработанные алгоритмы были реализованы в МКОД-системе и были проведены эксперименты по измерению количества тактов их работы для различных размеров графов. В экспериментах использовалась реализация СП со следующими параметрами: разрядность поля ключа и данных 32 бита; максимальное количество ключей в структуре $2 * 10^6$; количество уровней аппаратного (B+)-дерева 8; количество вершин на одном уровне 8; локальная память DDR 256 МБ; ширина шины памяти 64 бит; частота шины памяти 100 МГц; частота СП 100 МГц, ПЛИС FPGA Virtex II Pro; режим работы СП режим сопроцессора.



а *б*

Рис. 3. Исследование производительности процессора обработки структур на алгоритмах построения минимального остовного дерева

Для сравнения эффективности алгоритмов МКОД с аналогичными алгоритмами в универсальных системах была использована программная реализация (B+)-дерева для процессоров с микроархитектурой x86 и ARM11. Для сравнения результатов измерений были использованы ЭВМ со следующими параметрами:

- ЭВМ с многими потоками команд и одним потоком данных (МКОД): разрядность поля ключа и данных 32 бита; максимальное количество ключей в структуре $2 * 10^6$; количество уровней аппаратного (B+)-дерева 8; количество вершин на одном уровне 8; локальная память 256 МБ, DDR; ширина шины памяти 64 бит; частота шины памяти 100 МГц; частота СП 100 МГц, ПЛИС FPGA Virtex II Pro; режим работы СП режим сопроцессора.
- ЭВМ с многими потоками команд и многими потоками данных (МКМД): объем оперативной памяти 4 ГБ, DDR3; количество процессорных ядер 4, Core i5; частота процессоров

2530 МГц; операционная система Ubuntu Linux 12.04 i386; компилятор c99; ширина шины памяти 64 бит; частота шины памяти 1333 МГц.

• ЭВМ с одним потоком команд и одним потоком данных (ОКОД): объем оперативной памяти 512 МБ, DDR3; количество процессорных ядер 1, ARM11 Broadcom 2835; частота процессоров 700 МГц; операционная система Raspbian; компилятор gcc; ширина шины памяти 64 бит; частота шины памяти 400 МГц.

Количество вершин графов, использованных в эксперименте, было ограничено размерностью ключа (32 бит). Так как в структуре E как для алгоритма Крускала, так и для алгоритма Прима хранятся номера вершин ребра и его вес (w, u, v), количество вершин в графе не может быть большим. Поэтому были использованы графы, содержащие от 400 до 16К вершин. На рис. 3 показаны результаты сравнения количества тактов для обоих алгоритмов, полученных в МКОД-системе и в системе на основе микропроцессора Intel. МКОД-система работала в режиме сопроцессора. Эксперименты показали, что аппаратная эффективность МКОД-системы в режиме сопроцессора для алгоритма Крускала в 1,5 раза выше, а для алгоритма Прима в 2,4 раза выше. При этом алгоритм Крускала решает задачу несколько быстрее (в 1,3 раза для МКОД-системы, в 1,8 раз для ARM11 и в 2,3 раза эффективнее на Intel Core i5).

Таблица 1

Результаты сравнения ЭВМ МКОД и универсальных ЭВМ

Эксперимент	Ускорение в МКОД
Алгоритм Прима (МКОД и ARM11)	10.3
Алгоритм Крускала (МКОД и ARM11)	7.8
Алгоритм Прима (МКОД и Intel Core i5)	2.4
Алгоритм Крускала (МКОД и Intel Core i5)	1.5

Оценка количества тактов работы алгоритмов Прима и Крускала в МКОД-системе в сравнении с универсальной системой на основе четырехъядерного процессора Intel Core i5 показала достаточно малое ускорение для первой. Объясняется это несколькими причинами. Во первых, разделение 32-разрядного ключа на три группы разрядов ($w[T]$, u, v) привело к тому, что в экспериментах были использованы небольшие графы (количество вершин от 400 до 16К). При таких размерностях задачи не удалось в полной мере показать возможности системы МКОД. Во вторых, при программной реализации алгоритмов в системе на основе многоядерных процессоров Intel выделение и освобождение памяти выполняется не самим алгоритмом, а менеджером памяти операционной системы. В многоядерной системе функционирование механизмов управления памяти осуществляется независимо от тестовой программы до или после ее работы, а также во время ее работы на других процессорных ядрах. В связи с этим, в полученных результатах для ЭВМ МКМД не учитываются влияние параллельного выполнения кода менеджера памяти. Это подтверждается результатами

экспериментов, выполненных на одноядерном микропроцессоре ARM11, для которого преимущество МКОД заметно выше (от 7.8 до 10.3 раз). Вместе с тем ускорение достигнуто также в сравнении с x86 системами.

Заключение

В работе были представлены модификации алгоритмов Крускала и Прима для МКОД-системы. Все операции алгоритмов были представлены в виде действий над структурами данных, выбраны варианты представления графа и определен состав ключей и значений в структурах. Это позволило получить модификацию алгоритма для реализации в процессоре обработки структур как в режиме сопроцессора, так и в МКОД-режиме.

В проведенных экспериментах именно размер ключа для текущей версии процессора обработки структур (32 разряда), а не объем локальной памяти, ограничивал размерность задачи. Из результатов экспериментов очевидно, что для успешного практического применения системы МКОД требуется реализовать процессор обработки структур с существенно большими размерностями ключей и данных (не менее 64 бит).

Было показано, что зависимость по данным между двумя потоками команд в МКОД-системе может быть сокращена при реализации в процессоре обработки структур специализированного блока генерации составных ключей. Результаты экспериментов показали, что процессор обработки структур позволяет ускорить выполнение алгоритмов Крускала в 1,5 раза, и алгоритма Прима в 2,4 раза в сравнении с универсальными системами.

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ: пер. с англ. М.: МЦНМО, 2000. 960 с.
2. Кнут Д. Искусство программирования. В 3 т. Т. 3. Сортировка и поиск: пер. с англ. 2-е изд. М.: Вильямс, 2000. 832 с.
3. Орлов С.А., Цилькер Б.Я. Организация ЭВМ и систем: учебник для вузов. 2-е изд. СПб.: Питер, 2011. 688 с.
4. Harris D.M., Harris S.L. Digital Design and Computer Architecture. 2nd ed. Boston: Morgan Kaufmann, 2012. 721 p.
5. Таненбаум Э.С., Остин Т. Архитектура компьютера: пер. с англ. 6-е изд. СПб.: Питер, 2015. 816 с.
6. Coole J., Stitt G. Traversal Caches: A Framework for FPGA Acceleration of Pointer Data Structures // International Journal of Reconfigurable Computing. 2010. Vol. 2010, Art. ID 652620. DOI: 10.1155/2010/652620
7. Williams J., Massie Ch., George A.D., Richardson J., Gosrani K., Lam H. Characterization of Fixed and Reconfigurable Multi-Core Devices for Application Acceleration // ACM

Transactions on Reconfigurable Technology and Systems. 2010. Vol. 3, no. 4. Art. no. 19. DOI: 10.1145/1862648.1862649

8. Integrated Cryptographic and Compression Accelerators on Intel Architecture Platforms. SOLUTION BRIEF. Intel QuickAssist Technology. Order No. 329879-001US. Intel Corporation. 2013. 5 p.
9. Kumar Sn., Shriraman A., Srinivasan V., Lin D., Phillips J. SQRL: hardware accelerator for collecting software data structures // Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT'14). ACM, New York, NY, USA, 2014. P. 475–476. DOI: 10.1145/2628071.2628118
10. Singh M., Leonhardi B. Introduction to the IBM Netezza warehouse appliance // Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'11) / ed. by M. Litoiu, E. Stroulia, S. MacKay. IBM Corp., Riverton, NJ, USA, 2011. P. 385–386.
11. Попов А.Ю. Электронная вычислительная машина с многими потоками команд и одним потоком данных: пат. 71016 Российская Федерация. 2008. Бюл. № 5. 1 с.
12. Попов А.Ю. Реализация электронной вычислительной машины с аппаратной поддержкой операций над структурами данных // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2011. Спец. вып. Информационные технологии и компьютерные системы. С. 83–87.
13. Попов А.Ю. Электронная вычислительная машина с аппаратной поддержкой операций над структурами данных // Аэрокосмические технологии: тр. Второй Междунар. научно-техн. конф., посвященной 95-летию со дня рождения академика В.Н. Челомея (РФ, Реутов–Москва, 19–20 мая 2009 г.). Т. 1 / ОАО «ВПК «НПО машиностроения»; МГТУ им. Н.Э. Баумана. М.: МГТУ им. Н.Э. Баумана, 2012. С. 296–301.
14. Попов А.Ю. Исследование производительности процессора обработки структур в системе с многими потоками команд и одним потоком данных // Инженерный журнал: наука и инновации. 2013. № 11. DOI: 10.18698/2308-6033-2013-11-1048
15. Попов А.Ю. Применение вычислительных систем с многими потоками команд и одним потоком данных для решения задач оптимизации // Инженерный журнал: наука и инновации. 2012. № 1. DOI: 10.18698/2308-6033-2012-1-80
16. Попов А.Ю. О реализации алгоритма Форда — Фалкерсона в вычислительной системе с многими потоками команд и одним потоком данных // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2014. № 9. С. 162–180. DOI: 10.7463/0914.0726416
17. Подольский В.Э. Об организации параллельной работы некоторых алгоритмов поиска кратчайшего пути на графе в вычислительной системе с многими потоками команд и одним потоком данных // Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 4. С. 189–214. DOI: 10.7463/0415.0764268

Science & Education of the Bauman MSTU
Electronic journal
ISSN 1994-0408

Science and Education of the Bauman MSTU, 2015, no. 11, pp. 505–527.
DOI: **10.7463/1115.0820770**
Received: 27.09.2015
Revised: 11.11.2015

The study of Kruskal's and Prim's algorithms on the Multiple Instruction and Single Data stream computer system

Popov A. Yu.[1,*]

[*]alexpopov@bmstu.ru

[1]Bauman Moscow State Technical University, Russia

Keywords: graph, MISD computer system, structure processor, minimum spanning tree, Kruskal's algorithm, Prim's algorithm

Bauman Moscow State Technical University is implementing a project to develop operating principles of computer system having radically new architecture. A developed working model of the system allowed us to evaluate an efficiency of developed hardware and software. The experimental results presented in previous studies, as well as the analysis of operating principles of new computer system permit to draw conclusions regarding its efficiency in solving discrete optimization problems related to processing of sets.

The new architecture is based on a direct hardware support of operations of discrete mathematics, which is reflected in using the special facilities for processing of sets and data structures. Within the framework of the project a special device was designed, i.e. a structure processor (SP), which improved the performance, without limiting the scope of applications of such a computer system.

The previous works presented the basic principles of the computational process organization in MISD (Multiple Instructions, Single Data) system, showed the structure and features of the structure processor and the general principles to solve discrete optimization problems on graphs.

This paper examines two search algorithms of the minimum spanning tree, namely Kruskal's and Prim's algorithms. It studies the implementations of algorithms for two SP operation modes: coprocessor mode and MISD one. The paper presents results of experimental comparison of MISD system performance in coprocessor mode with mainframes.

References

1. Cormen T.H., Leiserson C.E., Rivest R.L. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990. (Russ. ed.: Cormen T.H., Leiserson C.E., Rivest R.L. *Algoritmy: postroenie i analiz*. Moscow, MTsNMO Publ., 2000. 960 p.).

2. Knuth D.E. *The Art of Computer Programming. Vol. 3. Sorting and Searching. 2nd ed.* Reading, Massachusetts, Addison-Wesley, 1998. (Russ. ed.: Knuth D.E. *Iskusstvo programmirovaniia. V 3 t. Vol. 3. ortirovka i poisk*. Moscow, Vil'iams Publ., 2000. 832 p.).
3. Orlov S.A., Tsil'ker B.Ya. *Organizatsiya EVM i system* [Computer and systems organization]. St. Petersburg, Piter Publ., 2011. 688 p. (in Russian).
4. Harris D.M., Harris S.L. Digital Design and Computer Architecture. 2nd ed. Boston, Morgan Kaufmann, 2012. 721 p.
5. Tanenbaum A., Austin T. *Structured Computer Organization*. Prentice Hall, 2012. 800 p. (Russ. ed.: Tanenbaum A., Austin T. *Arkhitektura komp'yutera*. St. Petersburg, Piter Publ., 2015. 816 p.).
6. Coole J., Stitt G. Traversal Caches: A Framework for FPGA Acceleration of Pointer Data Structures. *International Journal of Reconfigurable Computing*, 2010, vol. 2010, art. ID 652620. DOI: 10.1155/2010/652620
7. Williams J., Massie Ch., George A.D., Richardson J., Gosrani K., Lam H. Characterization of Fixed and Reconfigurable Multi-Core Devices for Application Acceleration. *ACM Transactions on Reconfigurable Technology and Systems*, 2010, vol. 3, no. 4, art. no. 19. DOI: 10.1145/1862648.1862649
8. Integrated Cryptographic and Compression Accelerators on Intel Architecture Platforms. SOLUTION BRIEF. Intel QuickAssist Technology. Order No. 329879-001US. Intel Corporation. 2013. 5 p.
9. Kumar Sn., Shriraman A., Srinivasan V., Lin D., Phillips J. SQRL: hardware accelerator for collecting software data structures. *Proceedings of the 23rd International conference on Parallel architectures and compilation* (PACT'14). ACM, New York, NY, USA, 2014, pp. 475–476. DOI: 10.1145/2628071.2628118
10. Singh M., Leonhardi B. Introduction to the IBM Netezza warehouse appliance. In: Litoiu M., Stroulia E., MacKay S., eds. *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research* (CASCON'11). IBM Corp., Riverton, NJ, USA, 2011, pp. 385–386.
11. Popov A.Iu. *Elektronnaia vychislitel'naia mashina s mnogimi potokami komand i odnim potokom dannykh* [Electronic computer with multiple instruction streams and single data stream]. Patent RF, no. 71016, 2008. (in Russian).
12. Popov A.Iu. Implementation of an electronic computer with hardware support for operations on data structures. *Vestnik MGTU. Ser. Priborostroenie = Herald of the Bauman MSTU. Ser. Instrument Engineering*, 2011, Spec. iss. *Informatsionnye tekhnologii i komp'iuternye sistemy* [Information technology and computer systems], pp. 83–87. (in Russian).
13. Popov A.Iu. Electronic computer with hardware support for operations on data structures. *Aerokosmicheskie tekhnologii: tr.Vtoroi mezhdunarodnoi nauchno-tekhnicheskoi konferentsii,*

posviashchennoi 95-letiiu so dnia rozhdeniia akademika V.N. Chelomeia [Aerospace Technology: proceedings of the Second International Scientific and Technical Conference dedicated to the 95th anniversary of the birth of Academician V.N. Chelomei], RF, Reutov — Moscow, 19–20 May 2009. Vol. 1. Moscow, Bauman MSTU Publ., 2009, pp. 296–301. (in Russian).

14. Popov A.Yu. The study of the structure processor performance in the computer system with multiple-instruction streams and single-data stream. *Inzhenernyy zhurnal: nauka i innovatsii = Engineering Journal: Science and Innovation*, 2013, no. 11. DOI: 10.18698/2308-6033-2013-11-1048 (in Russian).
15. Popov A.Yu. Application of Computing Systems with Multiple Instruction Streams and Single Data Stream for Solving Optimization Problems. *Inzhenernyy zhurnal: nauka i innovatsii = Engineering Journal: Science and Innovation*, 2012, no. 1. DOI: 10.18698/2308-6033-2012-1-80 (in Russian).
16. Popov A.Yu. On the implementation of the Ford–Fulkerson algorithm on the Multiple Instruction and Single Data computer system. *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2014, no. 9, pp. 162–180. DOI: 10.7463/0914.0726416 (in Russian) .
17. Podol'skii V.E. On the Organization of Parallel Operation of Some Algorithms for Finding the Shortest Path on a Graph on a Computer System with Multiple Instruction Stream and Single Data Stream. *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2015, no. 4, pp. 189–214. DOI: 10.7463/0415.0764268 (in Russian).