MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 30, NO. 2, 2023
journal homepage: www.mais-journal.ru

ALGORITHMS

Recursive-Parallel Algorithm for Solving the Maximum Common Subgraph Problem

V. V. Vasilchikov[1]

DOI: 10.18255/1818-1015-2023-2-128-139

[1]P. G. Demidov Yaroslavl State University, 14 Sovetskaya, Yaroslavl 150003, Russia.

MSC2020: 68W10
Research article
Full text in Russian

Received March 22, 2023
After revision May 16, 2023
Accepted May 17, 2023

The paper proposes an algorithm for solving the problem of finding the maximum common subgraph. Both the sequential and the parallel version of the algorithm, their software implementation are described, and an experimental study of their effectiveness is carried out.

This problem is one of the most famous NP-complete problems. Its solution may be required when solving many practical problems related to the study of complex structures. We solve it in a statement when we need to find all possible isomorphisms of the found common subgraph. Due to the extremely high complexity of the problem, it is natural to want to speed up its solution by parallelizing the algorithm. To organize parallel computing, the author used the RPM_ParLib library. It allows to develop effective applications for parallel computing on a local network under the control of the runtime environment .NET Framework.The library supports a recursive-parallel programming style and provides effective work distribution and dynamic load balancing of computational modules during program execution. It can be used for applications written in any programming language supported by the .NET Framework. The purpose of the numerical experiment was to study the acceleration achieved due to the recursive-parallel organization of calculations. For the experiment, the author developed a special application in the C# language designed to generate various sets of initial data with specified parameters. The paper describes the features of the generated initial pairs of graphs, as well as the results obtained during the experiment.

Keywords: maximum common subgraph; isomorphism; parallel algorithm; recursion; .NET

INFORMATION ABOUT THE AUTHORS

Vladimir V. Vasilchikov
corresponding author

orcid.org/0000-0001-7882-8906. E-mail: vvv193@mail.ru
PhD.

Funding: Yaroslavl State University (project VIP-016).

For citation: V. V. Vasilchikov, “Recursive-Parallel Algorithm for Solving the Maximum Common Subgraph Problem”, *Modeling and analysis of information systems*, vol. 30, no. 2, pp. 128-139, 2023.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 30, № 2, 2023
сайт журнала: www.mais-journal.ru

Рекурсивно-параллельный алгоритм поиска максимального общего подграфа

В. В. Васильчиков[1]

DOI: 10.18255/1818-1015-2023-2-128-139

[1]Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, Ярославль, 150000, Россия.

УДК 519.688: 519.85
Научная статья
Полный текст на русском языке

Получена 22 марта 2023 г.
После доработки 16 мая 2023 г.
Принята к публикации 17 мая 2023 г.

В работе предложен алгоритм решения задачи нахождении максимального общего подграфа. Описаны последовательный и параллельный вариант алгоритма, их программная реализация и произведено экспериментальное исследование их эффективности.

Данная задача является одной из самых известных NP-полных задач. Ее решение может потребоваться при решении многих практических задач, связанных с исследованием сложных структур. Мы решаем ее в постановке, в которой требуется найти все возможные изоморфизмы найденного общего подграфа. Ввиду чрезвычайно высокой трудоемкости задачи желание ускорить ее решение за счет распараллеливания алгоритма является вполне естественным. Для организации параллельных вычислений автором использовалась библиотека RPM_ParLib, которая позволяет создавать параллельные приложения, работающие в локальной вычислительной сети под управлением среды исполнения .NET Framework. Библиотека поддерживает рекурсивно-параллельный стиль программирования и обеспечивает эффективное распределение работы и динамическую балансировку загрузки вычислительных модулей в процессе исполнения программы. Она может быть использована для приложений, написанных на любом языке программирования, поддерживаемом .NET Framework. Целью численного эксперимента было исследование ускорения, достигаемого за счет рекурсивно-параллельной организации вычислений. Для эксперимента автором было разработано специальное приложение на языке C#, предназначенное для генерации различных наборов исходных данных с заданными параметрами. В работе описаны характеристики сгенерированных исходных пар графов, а также результаты, полученные в ходе эксперимента.

Ключевые слова: максимальный общий подграф; изоморфизм; параллельный алгоритм; рекурсия; .NET

ИНФОРМАЦИЯ ОБ АВТОРАХ

Владимир Васильевич Васильчиков
автор для корреспонденции

orcid.org/0000-0001-7882-8906. E-mail: vvv193@mail.ru
канд. техн. наук, зав. кафедрой вычислительных и программных систем.

Финансирование: ЯрГУ (проект VIP-016).

Для цитирования: V. V. Vasilchikov, “Recursive-Parallel Algorithm for Solving the Maximum Common Subgraph Problem”, *Modeling and analysis of information systems*, vol. 30, no. 2, pp. 128-139, 2023.

Введение

Задача о нахождении максимального общего подграфа (MCS — maximum common subgraph) является одной из классических NP-полных задач дискретной оптимизации [1]. Потребность в решении этой задачи возникает в самых разных предметных областях, где требуется установление идентичности структур тех или иных сложных систем. В качестве примеров можно назвать транспортные, энергетические системы, системы связи, электронные схемы, задачи сравнения молекулярных структур, системы распознавания образов, а также задачи математической химии, исследование социальных сетей и многие другие.

Многочисленные авторы, занимавшиеся исследованием данной проблемы, решали ее в самых разных постановках. Например, задача решалась для неориентированных [2] и ориентированных [3] графов. Искомый подграф может быть индуцированным подграфом (MCIS — maximum common induced subgraph), полученным путем удаления узлов, или частичным подграфом (MCPS — maximum common partial subgraph), полученным путем удаления дуг и узлов [3]. В первом случае максимум определяется по числу вершин, во втором — по числу ребер. Для второго варианта также принято использовать аббревиатуру MCES — maximum common edge subgraph. Например в [4] задача рассматривается в обеих этих постановках применительно к неориентированным графам. Также в постановке задачи могут фигурировать дополнительные атрибуты (метки, веса) в зависимости от специфики области применения.

Методы решения задачи также предлагаются самые разные. Например, в [5] предлагаемый алгоритм похож на VF2 [6], а в [7] решение основано на построении так называемого графа ассоциаций и сведения к задаче о клике [8]. Там же в том числе предложены подходы к созданию параллельных алгоритмов, предназначенных для реализации на системах самой разной архитектуры [9].

Следует отметить чрезвычайно высокую вычислительную сложность данной задачи. Она намного выше, чем у других трудоемких задач об изоморфизмах, которыми автор занимался раньше [10, 11].

В первой части автор предлагает алгоритм, пригодный для решения обеих задач (MCIS и MCES) в случае как ориентированных, так и неориентированных графов. Алгоритм построен в стилистике метода ветвей и границ. При решении мы ограничились построением только связного подграфа, максимального по числу вершин либо ребер, а более точно: всех возможных изоморфизмов таких подграфов. Отсутствие упомянутого ограничения могло бы многократно повысить трудоемкость решения.

Далее предлагается параллельный вариант этого алгоритма, основанный на парадигме рекурсивного распараллеливания [12], а также обсуждаются результаты его исследования в виде серии экспериментов с его программной реализацией.

1. Постановка задачи

В упомянутой монографии М. Гэри и Д. Джонсона [1] приводится постановка задачи в следующей формулировке. Пусть есть два графа, заданных своими множествами вершин и дуг: $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$, и положительное целое число K. Существуют ли такие подмножества $E_1^* \subseteq E_1$ и $E_2^* \subseteq E_2$, что $|E_1^*| = |E_2^*| \geq K$, а графы $G_1^* = (V_1, E_1^*)$ и $G_2^* = (V_2, E_2^*)$ изоморфны? В этой формулировке максимум определяется по числу дуг, и, кроме того, отсутствует требование связности искомого подграфа. Впрочем, такая постановка задачи отнюдь не является единственно возможной.

Мы будем ее решать в постановке MCIS, которую сформулируем следующим образом. Пусть есть два графа, заданных своими множествами вершин и дуг: $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$, и положительное целое число K. Существуют ли такие подмножества $V_1^* \subseteq V_1$ и $V_2^* \subseteq V_2$, что $|V_1^*| = |V_2^*| \geq K$, а графы $G_1^* = (V_1^*, E_1^*)$ и $G_2^* = (V_2^*, E_2^*)$ изоморфны? Здесь через E_1^* и E_2^* мы обозначили множества дуг,

связывавших в исходных графах вершины из множеств V_1^* и V_2^* соответственно. Дополнительно мы потребуем связности графов G_1^* и G_2^*.

2. Последовательный алгоритм решения задачи

Основная схема предлагаемого алгоритма одинакова для вариантов MCIS и MCES, для ориентированных и неориентированных графов. Ее основным методом является рекурсивный метод *RecursiveMatch()*, осуществляющий перебор возможных пар (n, m), где вершина n берется из первого графа, а вершина m — из второго. Пока трудно предложить иной подход для получения точного решения этой задачи, кроме полного перебора. В то же время можно придумать целый ряд проверок для оптимизации этого перебора. Собственно, в этом и заключается основная идея метода ветвей и границ.

Для возможно более быстрого решения задачи наш алгоритм должен обладать следующими характеристиками:

- упорядоченный порядок рассмотрения пар, гарантирующий отсутствие повторов;
- максимально быстрое отсечение негодных вариантов;
- минимально возможную трудоемкость вычислений при выборе очередной пары в качестве кандидата на включение в строящуюся подстановку.

Для достижения этой цели очень важен выбор структур данных для хранения информации о текущем состоянии объектов и операций с этими структурами. Мы завели для каждого из графов объект класса *GraphState*, в котором имеются следующие ключевые массивы:

- *CoreByOrder* — содержит номера вершин в порядке их рассмотрения, они не обязательно идут подряд (для первого графа), поскольку при построении ядра (*Core*) нам требуется связность строящегося подграфа;
- *RecNodeStates* — содержит состояния вершин, которое задает либо номер парной вершины из другого графа, либо специальный признак, указывающий на возможность использования вершины на данной ветви вычислений.

Блок-схема метода *RecursiveMatch()* изображена на рис. 1. Метод имеет два параметра: *CoreSize* — размер построенной части подстановки и *Depth* — глубина вложенности рекурсии. Общая схема: метод порождает две ветви: с включением вершины n в строящуюся подстановку и исключением ее из таковой. Далее мы приводим необходимые комментарии к отдельным участкам вычислений.

1. *Проверка текущей ветви вычислений на бесперспективность.* Это основной элемент метода ветвей и границ, позволяющий сразу исключить из рассмотрения ветви, которые не могут улучшить уже достигнутый результат. В нашей задаче критерий для проверки почти очевиден. Для задачи MCIS мы считаем ветвь бесперспективной при выполнении условия $CoreSize + N - Depth < BestResult$, где N — количество вершин в первом графе, *BestResult* — уже достигнутый результат. Для задачи MCES мы могли бы считать ветвь бесперспективной, если оставшихся ребер не хватает для улучшения результата. Как показали эксперименты, это очень слабый критерий, который обычно не позволяет ускорить решение задачи, однако автор пока ничего другого предложить не может. Поэтому далее мы в основном будем обсуждать задачу только в постановке MCIS.
2. *Поиск n_ind — индекса первой подходящей вершины в массиве CoreByOrder для первого графа.* Подходящей вершина считается, если она не включена еще в *Core*, не исключена из рассмотрения на данной ветви и имеет связь с текущим ядром. Далее происходит перестановка номеров вершин в массиве *CoreByOrder*, которая позволяет все рассмотренные вершины держать в массиве подряд. Пусть это вершина n. Здесь же для каждой вершины первого графа вне *Core* происходит пересчет важной числовой характеристики, используемой в пункте 3а для быстрой отбраковки неподходящих для нее пар. Трудоемкость этого пересчета составляет $O(N - Depth)$.

Fig. 1. Block diagram of the serial method *RecursiveMatch(CoreSize, Depth)*

Рис. 1. Рис. 1. Блок-схема последовательного метода *RecursiveMatch(CoreSize, Depth)*

3. *Начало цикла по m — вершинам в массиве CoreByOrder для второго графа, начиная с индекса CoreSize.* Вершины с меньшими индексами к данному моменту уже включены в *Core*. В начале итерации происходит вычисление числовой характеристики, используемой в пункте 3a. Для вершин второго графа трудоемкость этого вычисления составляет $O(N - |Core|)$.
   (a) *Предварительная проверка пары* (n, m). Упомянутая числовая характеристика предназначена для возможно быстрой отбраковки пар, в которых вершины n и m не годятся для включения в текущее ядро. Мы попробовали около десятка разных вариантов такой характеристики и в итоге остановились на следующем. В случае неориентированных графов мы вычисляем сумму индексов вершин из *Core*, с которыми связана данная вершина (n или m в зависимости от графа). Здесь индекс в *Core* соответствует уровню вложенности рекурсии, на котором вершина была добавлена. В случае ориентированных графов такая характеристика вычисляется отдельно для входящих и выходящих ребер (мы для общности изложения не используем здесь термин «дуга»). Такая проверка здесь требует $O(1)$ операций, а вычисление ее (делается на ветке один раз) имеет трудоемкость $O(|Core|)$.
   (b) *Полная проверка пары* (n, m). Сводится к циклу проверки соответствия ребер двух графов при добавлении пары в ядро, имеет трудоемкость $O(|Core|)$). При положительном результате сохраняем состояние второго графа и запоминаем пару (n, m) в массивах *RecNodeStates* для обоих графов.
   (c) *Если результат лучший (или это повторение лучшего результата), запоминаем его.*
   (d) *Делаем рекурсивный вызов RecursiveMatch(CoreSize+1, Depth+1).* Ветвь «+».
   (e) *Восстанавливаем состояние второго графа.* Трудоемкость этого восстановления составляет $O(N - |Core|)$.
4. *Конец цикла по m.*
5. *Восстанавливаем состояние первого графа.* Трудоемкость восстановления составляет $O(N - Depth)$. Подготавливаем вторую ветвь вычислений, на которой вершина n не включена в подстановку. Пересчитываем состояние первого графа перед началом вычислений на этой ветви, учитывая запрет на включение вершины n в подстановку. Трудоемкость пересчета составляет $O(N - Depth)$.
6. *Выполняем рекурсивный вызов RecursiveMatch(CoreSize, Depth+1).* Ветвь «−».
7. *Восстанавливаем состояние первого графа.* Трудоемкость восстановления можно оценить как $O(N - Depth)$.
8. *Выходим из метода.*

Как можно видеть из сделанных замечаний, на каждом этапе вычислений мы свели трудоемкость вычислений к минимально возможной. Кроме того следует отметить, что в последовательном режиме вычисления на ветви «+» выгоднее выполнять раньше, чем на ветви «−». Так мы быстрее достигнем лучшего результата и следовательно раньше отбросим неперспективные ветви вычислений.

3. Программная реализация последовательного алгоритма

Для отладки и оценки качества предложенного алгоритма на языке C# были написаны две программы.

Первая предназначалась для генерации исходных данных задачи MCS, а именно: двух случайных графов с заданными характеристиками. Она позволяла выстроить пару ориентированных либо неориентированных графов заданного размера с заданной вероятностью дуги или ребра соответственно, а также пару регулярных графов заданного размера с заданной степенью вершины. В случае ориентированных графов под «регулярностью» мы понимали фиксированную сумму количества входящих и исходящих дуг. Кроме того, для отладки на начальном этапе программа

позволяла заложить в эти графы подграф заданного размера, конечно, с точностью до изоморфизма. Впрочем, обычно программа, реализующая предложенный выше алгоритм, находила решение лучшее, чем специально заложенное. На этапе проведения экспериментов возможность задания такого «гарантированного» решения уже не использовалась.

Вторая программа представляла собой собственно реализацию предложенного алгоритма. Она позволяет решать задачу как в постановке MCIS, так и в постановке MCES. Программа находит все лучшие решения в заданной постановке. В ряде случаях таких решений оказывается довольно много, поэтому при выводе в протокол их количество можно ограничить. Также программа собирает и выводит в протокол разнообразную статистику о ходе решения задачи. Мы дополнительно заложили в нее возможность свопинга графов еще до начала вычислений. Причина в том, что в случае, если рассматриваемые графы имеют разное количество вершин, алгоритм, как показали эксперименты, заметно быстрее работает, если в качестве первого выступает граф меньшего размера. Впрочем, в ходе экспериментов в качестве исходных данных мы использовали только графы одинакового размера и с одинаковыми характеристиками (вероятность ребра или степень вершины).

Как и следовало ожидать, трудоемкость данной задачи оказалась намного выше, даже по сравнению с задачей об изоморфизме граф-подграф [11]. Причина в том, что решая задачу MCS, мы не можем заранее знать размер искомого изоморфизма, а значит, вынуждены перебирать все подмножества искомых пар вершин. Количество необходимых вычислений в нашем случае растет с увеличением размера задачи намного быстрее, причем по-разному для упомянутых выше категорий исходных данных.

В процессе тестирования использовались компьютеры на базе двухъядерного процессора Intel Core i3-7100 с тактовой частотой 3.90 GHz и 8 GB оперативной памяти, работающие под управлением 64-разрядной ОС Windows 10. В ходе эксперимента мы ограничились теми исходными данными, которые позволяли на компьютере указанной конфигурации получить решение задачи в постановке MCIS за приемлемое время (не более 40 минут). Поэтому предельными размерами задач для наших экспериментов и как в последовательном, так и в параллельном варианте были следующие:

- для случайных ориентированных графов: 40 вершин, вероятность дуги 0.5;
- для случайных неориентированных графов: 25 вершин, вероятность ребра 0.5;
- для регулярных ориентированных графов: 30 вершин степени 15;
- для регулярных неориентированных графов: 25 вершин степени 10.

Из приведенных значений можно сделать вывод, что для неориентированных графов время работы алгоритма выше, чем для ориентированных, что почти очевидно, поскольку на этапе 3а (предварительная проверка пары (n, m)) отсекается больше вариантов.

Также было отмечено, что количество операций, необходимых для решения задачи, и соответственно время решения задачи для регулярных графов примерно с одинаковыми характеристиками имеет существенно больший разброс, чем для случайных графов. В ходе эксперимента для приемлемых размеров задачи оно могло различаться в 100 и более раз. Про случайные графы такого сказать нельзя, для них расхождение было не более 2–3 раз. Можно также отметить, что решение задачи для регулярных графов в среднем требует больше времени, чем для случайных, что, впрочем, по-видимому, объясняется характером связей внутри графов.

Для метода ветвей и границ ключевым является этап отсечения заведомо бесперспективных ветвей перебора (в нашем алгоритме это первый пункт). Эксперимент позволил оценить эффективность работы данного этапа для предлагаемого алгоритма в обеих постановках (MCIS и MCES). Как было отмечено выше, в случае MCES предложенная оценка оказалась слишком слабой и не позволила сколько-нибудь ускорить вычисления. Вместе с тем прогоны программы в варианте для тех же данных, что и для MCIS, позволили оценить эффективность этого отсечения для последнего,

поскольку в обоих случаях мы проходим по одним и тем же ветвям и можем вычислить, сколько в итоге их было забраковано алгоритмом MCIS. Конечно, размеры решаемых задач в этом случае еще меньше, поскольку трудоемкость для задачи MCES намного выше.

Приведем некоторые результаты таких прогонов. Они достаточно условны, поскольку имеется изрядный разброс статистики для разных исходных данных, однако позволяют сделать некоторую качественную оценку. Отметим также, что эти эксперименты проводились для меньших размеров задачи, чем было указано выше, поскольку при отключенной отбраковке неперспективных ветвей (а в задаче MCES так фактически и было) решение сильно замедляется и это ограничивает размер задачи, которую можно решить за приемлемое время.

Для случайных ориентированных графов (35 вершин, вероятность дуги 0.5) на этапе 1 отбраковывалось 45–50 % ветвей, из оставшихся вариантов на этапе 3а оставался 1 % или даже менее. Ускорение за счет обеих проверок составило примерно 200 раз.

Для случайных неориентированных графов (20 вершин, вероятность ребра 0.5) на этапе 1 отбраковывалось почти 90 % ветвей, из оставшихся вариантов на этапе 3а оставалось 5–6 %. Ускорение за счет обеих проверок составило более 150 раз.

Для регулярных ориентированных графов (28 вершин, степень вершины 14) на этапе 1 отбраковывалось примерно 45 % ветвей, из оставшихся вариантов на этапе 3а оставалось 1.5–2 %. Ускорение за счет обеих проверок составило примерно 135 раз.

Для регулярных неориентированных графов (20 вершин, степень вершины 10) на этапе 1 отбраковывалось примерно более 90 % ветвей, из оставшихся вариантов на этапе 3а оставалось 5–6 %. Ускорение за счет обеих проверок составило примерно 200 раз.

Следует отметить, что такие эксперименты мы проводили и для других (меньших) размеров задач. Они позволили сделать вывод, что с увеличением размера задачи результативность примененных проверок ощутимо возрастает, и этот факт наглядно демонстрирует эффективность применения метода ветвей и границ с предложенной выше оценкой перспективности для решения задачи в постановке MCIS.

4. Параллельный алгоритм решения задачи

Метод рекурсивного распараллеливания наиболее эффективно работает в тех случаях, когда решаемую задачу (или подзадачу) можно разделить на две подзадачи с равным или хотя бы сопоставимым объемом вычислений. Вариант с равным объемом является идеальным и, как любой идеальный вариант, в жизни не встречается. Поэтому в программные средства поддержки РП программирования [13] были заложены возможности динамической балансировки загрузки. Но если объемы двух порождаемых подзадач различаются очень сильно, от программиста требуется выстроить процесс порождения параллельных ветвей несколько иначе, чем в базовом алгоритме [12].

Предложенный выше последовательный алгоритм решения задачи распадается на две ветви («+» и «−»), объем которых отличается очень существенно. Более того, ветвь «+» представляет собой цикл с рекурсивными вызовами на отдельных итерациях. К такой схеме нас привело желание исключить повторное вычисление значений, образующих текущее состояние характеристик первого графа. Выстраивая параллельный алгоритм, представляется разумным рекурсивные вызовы ветви «−» на отдельных итерациях оставить для выполнения процессорному модулю (ПМ), породившему эту ветвь (через *H_Call()*).Тогда выполнение вычислений на ветви «+» было бы естественно оформить их через вызов *P_Call()*.

Однако правильной последовательностью оформления вызовов параллельных активаций является такая: *P_Call(), H_Call(), Wait()*. Поэтому в отличие от базовой версии алгоритма, где по указанным выше причинам ветвь «+» шла первой, мы изменили порядок следования ветвей. В результате алгоритм приобрел вид, представленный на рис. 2.

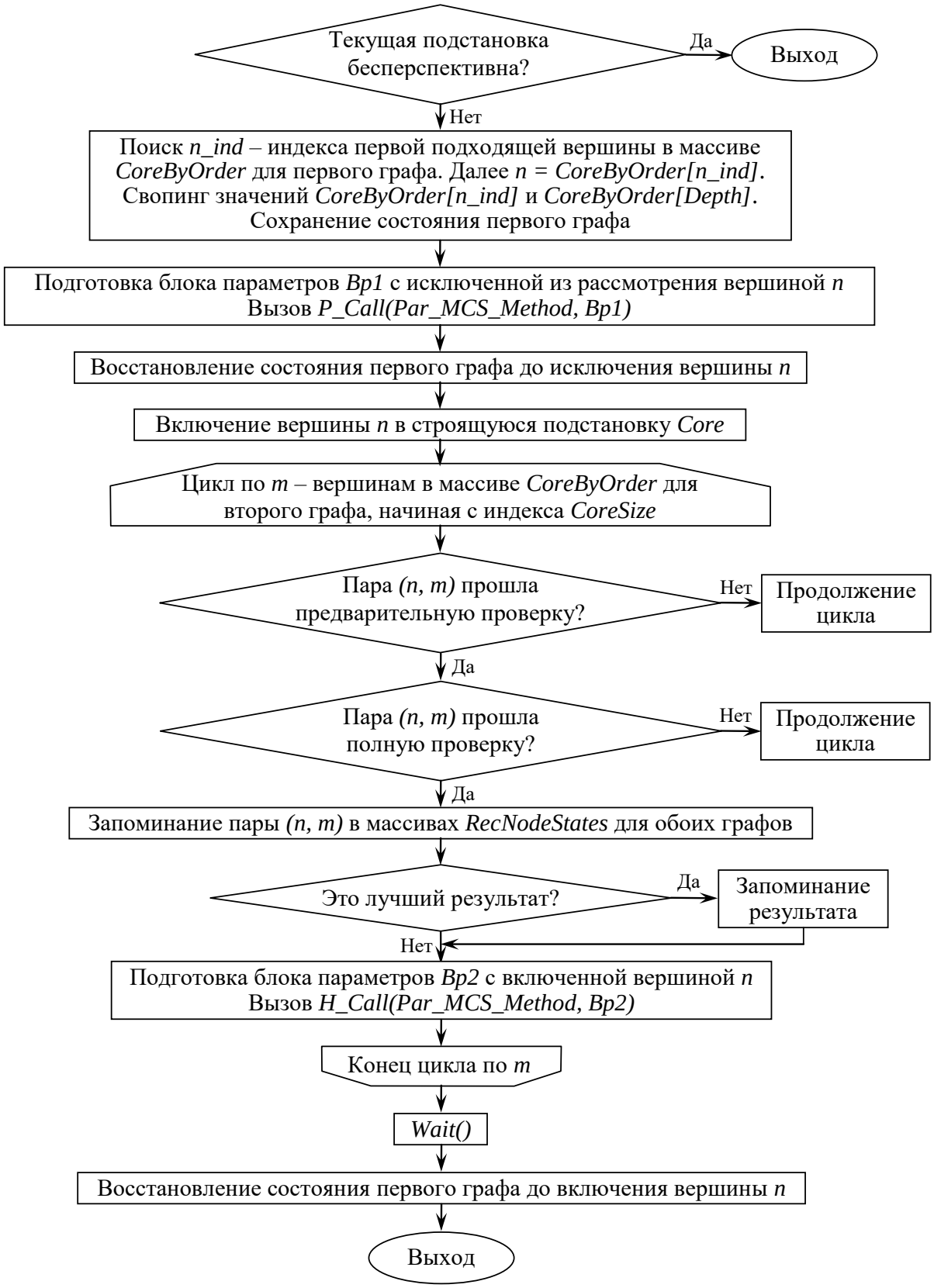

Fig. 2. Block diagram of the parallel method *RecursiveMatch(actState)*

Рис. 2. Блок-схема параллельного варианта метода *RecursiveMatch(actState)*

Table 1. The Acceleration of the RP-algorithm **Таблица 1.** Ускорение РП-алгоритма

Source Data		Sequently	1 PM	2 PM	4 PM	6 PM	8 PM	12 PM	16 PM
Example 1	Exec. Time (s)	1827.16	1507.80	772.53	608.41	412.29	371.10	316.54	293.55
Branches (mln):	Ratio to Seq		***1.21***	***2.37***	***3.00***	***4.43***	***4.92***	***5.77***	***6.22***
518.65	Ratio to 1 PM		**1.00**	**1.95**	**2.48**	**3.66**	**4.06**	**4.76**	**5.14**
Example 2	Exec. Time (s)	1647.64	1369.98	833.04	558.49	355.37	306.41	289.03	275.55
Branches (mln):	Ratio to Seq		***1.20***	***1.98***	***2.95***	***4.64***	***5.38***	***5.70***	***5.98***
463.26	Ratio to 1 PM		**1.00**	**1.64**	**2.45**	**3.86**	**4.47**	**4.74**	**4.97**
Example 3	Exec. Time (s)	1653.02	1379.50	815.17	573.53	352.70	351.02	327.41	261.34
Branches (mln):	Ratio to Seq		***1.20***	***2.03***	***2.88***	***4.69***	***4.71***	***5.05***	***6.33***
462.06	Ratio to 1 PM		**1.00**	**1.69**	**2.41**	**3.91**	**3.93**	**4.21**	**5.28**
Example 4	Exec. Time (s)	2399.06	1769.01	789.57	566.08	423.37	357.85	284.71	280.72
Branches (mln):	Ratio to Seq		***1.36***	***3.04***	***4.24***	***5.67***	***6.70***	***8.43***	***8.55***
888.37	Ratio to 1 PM		**1.00**	**2.24**	**3.13**	**4.18**	**4.94**	**6.21**	**6.30**
Example 5	Exec. Time (s)	2391.80	1763.76	787.03	632.88	425.31	357.83	327.41	322.73
Branches (mln):	Ratio to Seq		***1.36***	***3.04***	***3.78***	***5.62***	***6.68***	***7.31***	***7.41***
884.58	Ratio to 1 PM		**1.00**	**2.24**	**2.79**	**4.15**	**4.93**	**5.39**	**5.47**
Example 6	Exec. Time (s)	1937.06	1221.53	743.62	450.25	335.93	304.79	267.13	247.84
Branches (mln):	Ratio to Seq		***1.59***	***2.60***	***4.30***	***5.77***	***6.36***	***7.25***	***7.82***
712.73	Ratio to 1 PM		**1.00**	**1.64**	**2.71**	**3.64**	**4.01**	**4.57**	**4.93**
Example 7	Exec. Time (s)	1931.40	1463.80	661.91	416.18	373.29	293.43	221.17	212.74
Branches (mln):	Ratio to Seq		***1.32***	***2.92***	***4.64***	***5.17***	***6.58***	***8.73***	***9.08***
598.41	Ratio to 1 PM		**1.00**	**2.21**	**3.52**	**3.92**	**4.99**	**6.62**	**6.88**
Example 8	Exec. Time (s)	2698.58	1995.70	860.23	503.60	378.03	375.82	322.94	285.84
Branches(mln):	Ratio to Seq		***1.35***	***3.14***	***5.36***	***7.14***	***7.18***	***8.36***	***9.44***
818.04	Ratio to 1 PM		**1.00**	**2.32**	**3.96**	**5.28**	**5.31**	**6.18**	**6.98**
Example 9	Exec. Time (s)	3412.75	2668.96	1090.94	598.74	475.16	423.73	398.26	350.27
Branches (mln):	Ratio to Seq		***1.28***	***3.13***	***5.70***	***7.18***	***8.05***	***8.57***	***9.74***
1036.09	Ratio to 1 PM		**1.00**	**2.45**	**4.46**	**5.62**	**6.30**	**6.70**	**7.62**
Example 10	Exec. Time (s)	2298.15	1662.66	768.89	658.51	446.11	370.32	353.07	285.87
Branches (mln):	Ratio to Seq		***1.38***	***2.99***	***3.49***	***5.15***	***6.21***	***6.51***	***8.04***
836.57	Ratio to 1 PM		**1.00**	**2.16**	**2.52**	**3.73**	**4.49**	**4.71**	**5.82**
Example 11	Exec. Time (s)	1976.47	1480.91	667.13	470.35	355.83	289.67	226.97	222.30
Branches (mln):	Ratio to Seq		***1.33***	***2.96***	***4.20***	***5.55***	***6.82***	***8.71***	***8.89***
702.42	Ratio to 1 PM		**1.00**	**2.22**	**3.15**	**4.16**	**5.11**	**6.52**	**6.66**
Example 12	Exec. Time (s)	2026.67	1467.65	680.11	474.15	332.51	317.74	260.46	224.02
Branches (mln):	Ratio to Seq		***1.38***	***2.98***	***4.27***	***6.09***	***6.38***	***7.78***	***9.05***
724.37	Ratio to 1 PM		**1.00**	**2.16**	**3.10**	**4.41**	**4.62**	**5.63**	**6.55**

Передача накопленной информации о текущем состоянии строящейся подстановки осуществляется через специальную структуру, называющуюся блоком параметров. Она содержит список уже включенных в построенную часть подстановки (*Core*) вершин в порядке их включения, а также информацию о связях каждой такой вершины с вершинами из *Core*. Эта информация используется для предварительной проверки очередной пары кандидатов на включение в *Core*. Решение о том, какую версию основного метода (последовательную или параллельную) запускать на очередном этапе вычислений, принимается на основании двух параметров: предельной глубины вложенности и предельного размера построенной части подстановки.

5. Программная реализация параллельного алгоритма и результаты экспериментов

Целью эксперимента была оценка ускорения решения задачи, достигаемого за счет использования предложенного параллельного алгоритма. В качестве исходных данных мы брали сгенерированные случайным образом пары графов с различными характеристиками. Они перечислены в разделе, посвященном описанию экспериментального исследования последовательного алгоритма. Там же указаны вычислительные характеристики компьютеров, использованных для проведения эксперимента. Пропускная способность сети равнялась 100 Mb/s.

Собранные в таблице 1 результаты демонстрируют довольно хорошую эффективность предложенного рекурсивно-параллельного алгоритма. Результаты получены на 12 парах случайных графов с различными характеристиками, заложенными при их построении, а именно:

- примеры 1–3: случайные ориентированные графы из 40 вершин, вероятность дуги 0.5;
- примеры 4–6: случайные неориентированные графы из 25 вершин, вероятность ребра 0.5;
- примеры 7–9: регулярные ориентированные графы из 30 вершин степени 15;
- примеры 10–12: регулярные неориентированные графы из 25 вершин степени 10.

Отметим, что наличие ускорения, достигаемого параллельной программой на одном компьютере, по сравнению с последовательной ее версией, объясняется использованием многопоточности.

References

[1] M. R. Garey and D. S. Johnson, “Computers and intractability: A guide to the theory of NP-completeness”, *Siam Review*, vol. 24, no. 1, pp. 90–91, 1982.

[2] R. Hoffmann, C. McCreesh, and C. Reilly, “Between subgraph isomorphism and maximum common subgraph”, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017, pp. 3907–3914.

[3] S. N. Ndiaye and C. Solnon, “CP models for maximum common subgraph problems”, *Lecture Notes in Computer Science*, vol. 6876, pp. 637–644, 2011.

[4] D. Conte, P. Foggia, and M. Vento, “Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs”, *Journal of Graph Algorithms and Applications*, vol. 11, no. 1, pp. 99–143, 2007.

[5] J. J. McGregor, “Backtrack search algorithms and the maximal common sub-graph problem”, *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “An improved algorithm for matching large graphs”, in *Proc. of the 3rd IAPR-TC-15 InternationalWorkshop on Graph-based Representations*, Citeseer, 2001, pp. 149–159.

[7] P. J. Durand, R. Pasari, J. W. Baker, and C.-c. Tsai, “An efficient algorithm for similarity analysis of molecules”, *Internet Journal of Chemistry*, vol. 2, no. 17, pp. 1–16, 1999.

[8] C. Bron and J. Kerbosch, “Algorithm 457: Finding all cliques of an undirected graph”, *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[9] A. Marcelli, S. Quer, and G. Squillero, *The maximum common subgraph problem: A portfolio approach*, 2019. arXiv: 1908.06418 [cs.DS]. [Online]. Available: http://arxiv.org/abs/1908.06418.

[10] V. V. Vasilchikov, “Parallel algorithm for solving the graph isomorphism problem”, *Modeling and analysis of information systems*, vol. 27, no. 1, pp. 86–94, 2020.

[11] V. V. Vasilchikov, “Recursive-parallel algorithm for solving the graph-subgraph isomorphism problem”, *Modeling and analysis of information systems*, vol. 29, no. 1, pp. 30–43, 2022.

[12] V. V. Vasilchikov, *Sredstva parallelnogo programmirovaniya dlya vychislitelnykh sistem s dinamicheskoy balansirovkoy zagruzki.* YarGU, Yaroslavl, 2001, in Russian.

[13] V. V. Vasilchikov, "On the recursive-parallel programming for the .NET framework", *Automatic Control and Computer Sciences*, vol. 48, pp. 575–580, 2014.