Ордена Ленина ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М. В. Келдыша
Российской Академии наук

А. В. Ворожцов

Мета-методы $\mathcal{NP}$-программирования

Москва
2005

АННОТАЦИЯ

Работа посвящена басизным идеям, которые используют программисти при разработке приближённых алгоритмов решения $\mathcal{NP}$-сложных или плохо формализованных задач. Большая часть этих методов известна и активно используется на практике. Приведена классификация этих методов и рассмотрено несколько важных аспектов осуществления метасистемных переходов на уровне организации алгоритмов. Описан слабо изученный метод введения макро-объектов и макро-языка, а известные генетические алгоритмы, метод отжига, метод масштабирования разложены на более элементарные мета-эвристики.

ABSTRACT

The work concerns basic notions used for constructing approximation method for solution of NP-hard and poorly formalized complex problems. Most of these notions are well-known. Detailed description of these notions and their use-cases are given. Method of inferring macro-entities (macro-objects and macro-actions) is proposed. Attention is paid to the metasystem transactions on the level of algorithm organization. Simulated annealing, scaling method, genetic algorithms and other known methods are considered in this context and decomposed into more elementary metaheuristics.

Работа выполнена при поддержке РФФИ (проект №04-01-00510) и гранта Президента РФ (проект НШ-374.2003.1)

Содержание

1 Введение

В теории алгоритмов выделяют несколько базовых методов решения алгоритмических задач (метод итераций, “разделяй и властвуй”, динамическое программирование, “жадные” алгоритмы и др.) и базовых структур данных (деревья поиска, двоичная и фибоначчиева “кучи”, хэш-таблицы и др.). На этих методах и структурах данных основаны алгоритмы решения большинства классических полиномиально-разрешимых задач, и их можно рассматривать как “базис” полиномиального программирования — P-программирования.

Но на практике часто возникают задачи, которые не решаются указанными методами. Часто это $\mathcal{NP}$-сложные задачи, которые за полиномиальное время можно решать только приближённо, либо плохо формализуемые задачи, с большим количеством деталей и условий, подобные задаче перевода и анализа текстов на естественном языке или задаче классификации (распознавания) и кластеризации каких-либо сложных объектов. При разработке решений таких задач центральными становятся другие методы, близкие к методам, используемым человеческим мышлением.

Методы “разделяй и властвуй” и “жадные алгоритмы” естественны для человеческого мышления и играют важную роль как в P- так и в $\mathcal{NP}$-программировании. Но если в классических алгоритмических задачах они являются методом точного решения, то в “человеческих” задачках они дают приближённые решения или даже являются средством упрощения задачи. “Жадность” приходится применять там, где жадный алгоритм не даёт точного решения. А метод “разделяй и властвуй” приобретает несколько иной оттенок: чтобы хоть как-то подойти к решению задачи, приходится некорректно делить её на несколько подзадач, точное решение которых не гарантирует точного решения исходной задачи.

Ниже сделана попытка выделить базовые методы[1], на которых можно строить решение сложных плохо формализуемых или $\mathcal{NP}$-сложных задач.

2 Базовые составляющие алгоритмов решения сложных задач

Перечислим условные названия этих методов:

divide: Деление на подзадачи (“разделяй и властвуй”).

reduction: Сведение к другой задаче или последовательности задач.

[1]Строго говоря, методы, о которых пойдет разговор, не являются методами в полном смысле этого слова. Под методом понимают алгоритм решения задач из определённого класса. В нашем случаем мы называем методами элементарные кирпичики метода разработки методов, то есть общие идеи, которые могут послужить отправными точками при разработке приближенных алгоритмов.

parallel: Распараллеливание.

competition: Отбор лучших.

macro: Выделение макро-объектов и проецирование задачи на макро-объекты.

local: Последовательные локальные улучшения.

greedy: Жадность.

meta: Метасистемные переходы в программировании (“мета-алгоритмы”).

model: Сведение задачи к моделированию физической системы.

random: Метод случайного выбора.

Этими методами удобно оперировать при разработке сложных алгоритмов. Прежде, чем перейти к их подробному описанию приведём которкое пояснение и несколько примеров.

Метод, заключающийся в отборе лучшего варианта (choose the best), полученного в результате работы нескольких различных приближёных алгоритмов, является синтезом методов parallel и competition:

$$\text{choose the best} = \text{parallel} + \text{competition}.$$

Ярким примером использования методов reduction и model является алгоритм `neato` [1] получения наглядного плоского представление графа, основанный на пружинной модели (reduction). В этом алгоритме каждому ребру графа ставится в соответствие притягивающая пружина некоторой фиксированой длины. Кроме того, устанавливаются отталкивающие пружины между парами вершин, а также между точками ребер и вершинами, чтобы избежать перекрытия вершин (поощрить отсутствие перекрытий) и перечечения рёбер. Далее моделируется (model) эволюция диссипативной динамической системы с различными начальными случайными (random) состояними и наиболее оптимальное (competition) из получившися конечных состояний возвращается как результат. Разработчики алгоритма `neato` использовали метод reduction и свели задачу рисования графа к задаче поиска абсолютного минимума некоторой функции. Далее, интерпретируя эту функцию как потенциальную энергию физической системы, они перешли к задаче моделирования динамической системы с целью поиска локальных минимумумов потенциальной энергии. В данной задаче метод model является лишь способом реализации метода локальных улучшений local, модификацией известного метода градиентного спуска.

Генетические алгоритмы (evolution) являются другим ярким примером воплощения методов model и local. Их суть заключается в моделировании процесса эволюции с целью поиска оптимальной конфигурации — конфигурации, на которой достигается минимум некоторого функционала. Генетические алгоритмы являются синтезом нескольких компонент:

$$\text{evolution} = \text{model} + \text{random} + \text{parallel} + \text{competition} + \text{local}.$$

Последовательные локальные улучшения (local) в генетических алгоритмах проявляются в том, что мутация и рекомбинация генов – это малые случайные (random) небольшие изменения параметров особи-решения, которые, если являются улучшающими, будут переданы потомкам, поскольку эти особи победят в соревновании с другими особями за ресурсы (competition). Идея параллельности parallel заложена в том, что особи-решения "живут" одновременно.

Если к указанной сумме добавить составляющую meta, то получится терариум алгоритмов, в котором живут особи-алгоритмы. Естественному отбору уже будут подвергаться не сами приближённые решения, а алгоритмы, которые их ищут. Если эти алгоритмы сами построены как генетические алгоритмы, то мы получим естественный отбор различных эволюционных алгоритмов.

Метод масштабирования (scale) — следующий важный метод решения сложных задач. Когда к задаче сложно подступится, можно осуществить операцию *огрубления задачи*. Цель операции огрубления заключается в том, чтобы уменьшить сложность (размер) исходных входных данных, путём выделения из них только самых ключевых объектов. Часто операция огрубления представляет собой разбиение большого числа объектов на меньшее число макро-объектов. Сильно огрубив задачу, её можно решить одним из эвристических или/и неполиномиальных алгоритмов, хорошо работающих на входных данных небольшого размера. Затем можно уменьшить степень огрубления задачи, и, использовав результат, полученный при решении сильно огрублённой задачи, методом локальных улучшений получить решение менее огрублённой задачи. Такими последовательными шагами можно дойти до решения исходной задачи. Таким образом, метод масштабирования есть синтез трёх идей:

$$\text{scale} = \text{macro} + \text{local} + \text{meta}.$$

Рассмотрим базовые методы $\mathcal{NP}$-программирования подробнее.

2.1 Метод divide: "разделяй и властвуй"

Метод разбиения на подзадачи divide — один из ключевых методов человеческого мышления. Люди, как и вычислительные машины, в отдельный момент времени могут работать с ограниченным количеством информации и совершать небольшое количество базовых действий. Поэтому единственный способ решать сложные задачи — это разбивать их на более простые.

Важно отметить два принципиально различных типа разбиения на подзадачи: разбиение на подобные задачи ("devide and conquer") и разбиение на последовательность разнородных задач ("decompose and conquer").

В первом случае программисты используют рекурсию — пишут процедуры и функции испольующие сами себя при вычислении. Классическим приме-

ром является метод быстрой сортировки, который сводит сортировку одного массива к сортировке двух массивов меньшего размера.

А во втором случае задача разбивается на последовательность шагов, которые выполняют разнородные задачи. При более подробном рассмотрении каждый шаг разбивается на более элементарные шаги. Собственно, этот подход является основой процедурного и структурного программирования [4, 5]. Программисту рекомендуется требуемую задачу разбивать на последовательность макро-действий и оформлять их в виде процедур, особенно если они достаточно универсальны и могут пригодиться в других задачах.

В обоих своих вариантах метод devide очень часто применяется в жизни и при построении компьютерных систем и кажется нам обычным и очевидным подходом к решению задач. Но простота этого метода обманчива — иногда за делением на подзадачи скрывается сложная операция. Задачу "получить машину серебристого цвета" можно разбить на три: "получить машину", "получить серебристую краску", "покрасить машину". Но дальше эти задачи так просто не делятся. Далее появляются задачи проектирования и создания заводов по металлообработке и отливке деталей, задачи создания двигателя и получения бензина, тестирования и др. Впрочем, другой человек задачу получения машины может разбить совсем на другие подзадачи: "заработать денег", "выбрать марку и цвет", "получить права" и "купить". Для того чтобы эффективно разбить задачу на подзадачи, нужно знать имеющийся базис функциональности.

Следующий сложный момент, который возникает на этапе деления задачи на подзадачи — это некорректное деление, то есть деление на подзадачи, точное решение которых не даст точного решения исходной задачи. Например, пусть нужно найти такое представления графа на плоскости, в котором рёбра являются ломаными с сегментами, параллельными осям координат, и число пересечений рёбер минимально. Эту задачу можно некорректно разбить на две подзадачи: "найти оптимальное положение вершин графа на плоскости" и "провести оптимальным образом рёбра для данного положения вершин на плоскости". При этом для первой подзадачи нужно будет придумать *искусственную меру оптимальности* положения вершин. Эта искусственная мера должна как можно сильнее "коррелировать" с настоящей мерой оптимальности для исходной задачи.

К такой вариациации метода divide разделения задачи на подзадачи можно отнести метод reduction — метод сведения одной задачи к другой. Он заключается в том, что данная сложная задача сводится к другой, более простой, или задаче с известным решением. Например, задачу "получить защищенный канал передачи информации" можно свести к задаче "найти очень большое простое число", которую, в свою очередь, можно свести к задаче "найти большое простое число вида $2^{2^n} + 1$".

2.2 Метод parallel: распараллеливание

Если людей (процессоров) несколько, то распараллеливание естественным образом работает в связке с разбиением на подзадачи. Кроме того, даже одному человеку (процессору) присуще решать сложные задачи параллельно. Эта параллельность применяется в различных ситуациях.

Во-первых, задачу можно решать “с разных концов” или разными способами. Это позволяет со временем увидеть наиболее продуктивные пути, ведущие к быстрому и оптимальному решению задачи. В случае решения задач на компьютере, можно контролировать прогресс вычислений различных алгоритмов и через некоторое время отдать процессорное время только перспективному алгоритму. Исследователи довели эту мысль до конца и пришли к генетическим алгоритмам.

Синтез — это понятие двойственое к распараллеливанию. Ветвление подразумевает слияние. Результаты, полученные в параллельных вычислительных потоках, в конечном счете, должны быть объеденены.

Синтез и распараллеливание являются лишь частными случаями общего представления задачи как графа подзадач, связанных ребрами зависимостей. Обощение метода parallel звучит следующим образом.

Задачу можно представить как набор вершин (подзадач), соединённых направленными рёбрами (зависимостями). Среди вершин одна отмечена как начальная, а несколько вершин отмечены как конечные. Для каждой из вершин известна предварительная оценка вычислительных ресурсов, которые необходимы для того, чтобы решить соответствующую ей подзадачу. К решению подзадачи можно приступать лишь тогда, когда все подзадачи, от которых она зависит, уже решены. Необходимо, следуя указанному правилу, как можно скорее добраться до одной из конечных вершин, имея фиксированное количество вычислительных ресурсов. По ходу решения подзадач необходимо делать коррекцию к оптимальному маршруту – сделанные на начальном этапе оценки времени решения подзадач могут существенно отличаться от реальных. Полезно также иметь несколько запасных путей на случай, когда одна из вершин в оптимальном пути окажется неподъёмной или откжет какое-либо звено (несработавший приближенный алгоритм, ненадёжность передачи данных и вычислительных машин, и др.). Запасаные пути хороши также тем, что если решений будет получено несколько, из них можно будет выбрать оптимальное.

Еще раз хочется отметить, что здесь речь идёт не о проектировании системы, а о методе рассуждений при разработке приближенных алгоритмов решения сложных задач. В конечном итоге алгоритм, в который будет заложена указанная идея параллельности, может с успехом отработать и на одном процессоре. Параллельные вычисления (также как и двумерные массивы) лишь абстракция, полезная при разработке алгоритмов (на обычном PC процессор в каждый момент времени выполняет лишь одну операцию, а

память является линейной). Но при желании, заложенную в алгоритме параллельность можно воплотить и в реальные параллельные вычисления на многопроцессорных системах.

2.3 Метод competition: отбор лучших

Как уже было отмечено, генетические алгоритмы (evolution) основаны на методах распараллеливания (parallel), отборе лучшего (competition) и методе локальных случайных улучшений (local+random).

Метод competition отбора лучшего подразумевает также и "забывание" худшего. Причём забывание худшего играет не меньшую роль, чем отбор и поощрение лучшего.

Одним из простых примеров использования в программировании метода competition является кэширование значений функций, которые часто используются в алгоритме, а также кэширование блоков жесткого диска в оперативной памяти, кусочков оперативной памяти в кэше процессоре и другие виды кэширования.

Для человека (мозга) естественно рассматривать методы решения задач, приёмы, базовые идеи и другие элементарные составляющие мышления как особей, между которыми идет борьба. Наиболее успешные из них человек держит в активной памяти и забывает те, которые не находят применения или регулярно не приводят к решению. Если бы человек (мозг) помнил всё, то, скорее всего, потерял бы работоспособность. Необходимо, чтобы лишние знания не мешали быстрому поиску нужных знаний. Опыт есть не что иное, как множество успешно конкурирующих и уживающихся друг с другом приёмов мышления и решения задач.

Решение задач можно организовать как террариум, в котором живут различные методы и наиболее успешным из них позволяется "почковаться", то есть создавать свои слегка модифицированные копии (мутация). Иногда есть естественный способ соединять два алгоритма в один (рекомбинация). Можно сказать, что здесь используется ещё один общий метод — синтез. Но мы не будем выделять его в отдельный метод $\mathcal{NP}$-программирвания. Знак "плюс", которые мы пишем в формулах, соответствует этому мета-методу.

Таким образом, соревнование происходит и между решениями, и между алгоритмами, и между методами конструирования алгоритмов (методами мышления).

Отбор лучших — это простое, всем известное основополагающе явление. Ученые и програмисты всегда занимались сравнением качества решений и методов и основывали на отборе лучшего свои алгоритмы. Сложность заключается в том, чтобы правильно выбрать уровень (уровень системы, метасистемы и т.д.) и этап, на котором производить отбор, а также установить наиболее оптимальные параметры отбора.

2.4 Метод local: локальные улучшения, “жадные” алгоритмы

Метод локальных улучшений, “жадные алгоритмы” и метод градиентного спуска можно сформулировать общим образом: делать последовательные шаги, которые улучшают текущую ситуацию (приближают к решению задачи). Жадные алгоритмы отличаются лишь тем, что из всех возможных локальных (малых) изменений выбирается самое лучшее. Конечно, иногда жадный алгоритм формулируется в терминах пошагового построения решения, а не пошагового изменения одного из неоптимальных решений. Но, тем не менее, даже такие жадные алгоритмы можно интерпретировать как последовательное жадное дополнение имеющегося частичного решения.

Общий метод local можно трактовать как метод сведения глобальной оптимизации к локальной.

Есть ограниченное количество задач, для которых метод local приводит к точному решению. Это, в частности, задача о заявках, кратчайший путь в графе, минимальное покрывающее дерево, коды Хаффмана, поиск минимума функции с одним минимумом и др.

Но в случае $\mathcal{NP}$-сложных задач метод локальных улучшений сам по себе не приводит к хорошим решениям. $\mathcal{NP}$-сложные задачи тем и отличаются от полиномиально разрешимых, что их целевая функция содержит множество локальных минимумов, которые могут существенно отличаться по величине от абсолютного минимума. Можно сказать, что график целевых функций $\mathcal{NP}$-сложных оптимизационных задач имеет фрактальный характер.

Метод локальных улучшений обычно дополняют различными эвристиками, элементами случайного блуждания (см. метод отжига). Эвристики необходимы, в первую очередь, для того, чтобы подготовить начальную ситуацию, из которой методом малых локальных изменений будет получаться решение.

2.5 Метод scale: масштабирование

Укажем одну важную модификацию метода локальных улучшений. Есть ряд эвристик, которые позволяют сгладить (огрубить, усреднить) целевую функцию, удалив множество малых неровностей графика целевой функции, и применять метод локальных улучшений к сглаженной целевой функции. После обнаружения локального минимума сглаженную функцию заменяют на исходную и уточняют решение снова с помощью локальных улучшений. Можно ввести несколько уровней сглаживания (огрубления) и прийти к методу масштабирования: на каждом этапе в качестве начальной позиции для осуществления последовательности локальных улучшений используется решение, полученное на предыдущем уровне огрубления.

Метод локальных улучшений хорошо сочетается с методом “разделяй и властвуй”, поскольку локальные улучшения работают хорошо только на вход-

ных данных малого размера (когда число возможных конфигураций в пространстве поиска мало, то и число потенциальных локальных минимумов тоже мало). В случае поиска оптимального представления графа естественно разбить граф на кусочки и к каждому кусочку применить метод локальный улучшений. Далее можно воспользоваться другим важным методом — методом введения макро-объектов. А именно, рассмотреть кусочки графа как вершины нового графа (разбить граф на подграфы и свернуть каждый подграф в вершину) и найти для этого графа оптимальное представление. По сути, это снова та же идея масштабирования. Метод масштабирования scale является комбинацией метода локальных улучшений local и введения макро-объектов macro нескольких уровней огрубления:

$$\text{scale} = \text{local} + \text{macro}.$$

При огрублении целевой функции роль макро-объектов выполняют окрестности близких друг к другу конфигураций, по которым происходит усреднение. Типичная схема алгоритмов, основанных на методе масштабирования выглядит следующим образом:

Решить задачу P:
  $Solution = Random(P)$ или $Solution = Greedy(P)$
  $N = EstimateN(P)$
  for $i = N$ downto 0
    $P' = Scale(P, i)$
    $Solution = LocalBest(P', Solution)$
  return $Solution$

Этот псевдокод раскрывает смысл знака “+” в формуле $\text{scale} = \text{local} + \text{macro}$. Возможны и другие интерпретации этого знака, которые соответствуют алгоритмам с другими схемами.

2.6 Метод macro: введение макро-объектов

Мы упомянули выше операцию введения макро-объектов. Она в определённом смысле обратна операции разбиения на кусочки и заключается в том, что множество данных объектов разделяется на группы и с каждой группой связывается макро-объект. Затем формулируется новая задача в терминах этих макро-объектов, решение которой будет важным шагом на пути решения исходной задачи. Решить задачу с макро-объектами обычно бывает проще, так как число макро-объектов меньше, чем число исходных объектов. Кроме того, по своей природе она может быть совсем другой и легче решаемой, нежели исходная.

Например, при программировании стратегии игры в шахматы естественно с каждой клеткой поля связать два макро-объекта — множество белых фигур,

и множество чёрных фигур, которые бьют эту клетку. Каждой такой группе можно присвоить два атрибута: “размер”, равный числу фигур, и “ценность”, равную сумме ценностей фигур.

Другой пример относится к различным задачам на графах. В роли макрообъектов в них могут выступать подграфы. В частности, в задаче рисования графов ярко выраженные подграфы (отдельные связные компоненты или компоненты, связанные малым количеством ребер с остальными вершинами) можно на время “сворачивать в точку” и оперировать ими как отдельными вершинами.

Способность эффективно выделять макрообъекты и переформулировать задачу в терминах макрообъектов является, пожалуй, самой удивительной и таинственной способностью человеческого разума.

Ярким результатом является математика с множеством абстрактных объектов (алгебры, изоморфизмы, матрицы, векторные расслоения, аттракторы и др.), в терминах которых мыслят современные математики.

При решении сложных программистских задач выделение макро-объектов часто является ключевым шагом на пути эффективного приближенного решения задачи.

Есть ряд простых примеров: задача сжатия текстов — выделение часто повторяющихся лексем и контекстов, задача кластеризации по набору бинарных признаков — выделение групп сильно коррелирующих признаков, задача разпознавания образов – выделение контуров, направляющих линий и точек пересечения контуров и направляющих линий.

Есть более сложные примеры. При программировании игровых стратегий возникают следующие макро-объекты: “вилка”, “нападение”, “укрепление левого фланга”, “активные фигуры”. Шахматисты при изучении определенной позиции отбрасывают много ненужных ходов засчет того, что не рассматривают ходы неактивных фигур, и фигур, уже занятых каким-либо важным “делом”. Кроме того, отбрасываются многие нелепые ходы. Все это позволяет значительно повысит эффективную глубину перебора ходов “в уме”. Конечно, программа, которая просто перебирает дерево ходов, следуя стандартной процедуре $\alpha\beta$-отсечения, может при достаточной глубине перебора “увидеть” и “вилку”, и как-бы подсчитать число нападений и защит на ферзя и сделать правильный ход. Но при этом указанные макрообъекты будут присутствовать в программе неявно. Это будет призрак, никак не присутствующий в коде программы. Кроме того, при простом переборе дерева ходов может проявится так называемый эффект горизонта. Если у соперника есть достаточное количество отвлекающих маневров, он может сделать так, что “вилка” наверняка сработает лишь через несколько ходов. Например, соперник может сделать ряд нападений, которые вы с легкостью парируйте и лишь потом честно заработаете фигуру благодаря успешно расставленной “вилке”. Но при обдумывании дерева ходов на малую глубину эта “вилка” не будет

обнаружена программой.

Программирование на уровне макрообъектов подразумевает целенаправленый и более-менее дешевый поиск макро-объектов и введение соответствующих им программных сущностей.

С найденными макро-объектами, как с элементами головоломки, надлежит еще поработать, чтобы сложить из них нужную картинку и получить решение.

Приведём ещё одну итерпретацию метода macro. Можно сказать, что этот метод заключается в выделении макро-сущностей — макро-объектов и макро-процедур, что, по сути, означает создание языка, удобного для описания алгоритма решения конкретной задачи. Собственно, в компьютерных технологиях можно найти подтвержение тому, что этот метод активно применяется. Сейчас существует множество языков и каждый из них удобен для решения определённого класса задач. И прежде, чем программист приступает к решению задачи, он анализирует, какие из существующих технологий и языков программирования будут наиболее подходящими (полезными) для этой задачи. Например, существует язык для нейро-программирования, язык для описания 3D-сцен, язык описания физических систем, язык для программирования стратегий исскуственных организмов в различных играх и много других языков.

Полезно уметь бысто и эффективно создавать специальные языки для решения каждой конкретной сложной задачи. Сначала создается язык, приближенный по семантике к предметной области, а затем пишется на нём программа. Причём важнейшим шагом на пути создания интеллектуальных систем было бы перенос этих задачи на “плечи” компьютеров и мета-алгоритмов.

2.7 Метод meta: метасистемные переходы в программировании

Многие из указанных выше методов являются мета-методами, то есть указывают как на основе имеющихся, не достаточно точных или слишком медленных алгоритмов построить новый, более точный и более быстрый алгоритм. Для эффективного применения многих рассмотренных методов полезно иметь некоторое начальное решение, которое будет проанализировано и улучшено.

Метод meta это “оперирование имеющимися методами, алгоритмами и решениями с целью получения более эффективного алгоритма и решения”, то есть осуществление метасистемного перехода[2] в организации алгоритма. Метасистемный переход — это изменение (повышение уровня) организации

[2]Термин “метасистемный переход” ввёл В.Ф. Турчин в работе “Метасистемные переходы” [8, 9]

системы, при котором элементарными объектами новой системы становятся системы предыдущего уровня.

В технологиях программирования есть множество ярких примеров метасистемных переходов. Например, *процедурное программирование* заключается в разбиении программы на процедуры (функции, действия), при котором для описания новых, высокоуровневых процедур используются имеющиеся (более) низкоуровневые процедуры.

Другой простой пример — метод выбора лучшего competition. Пусть есть несколько различных алгоритмов решения одной и той же задачи, которые успешны в различных случаях. Тогда удобно создать мета-алгоритм, который оценивает ситуацию на входе и использует ниболее подходящий для этой ситуации алгоритм.

Современные программисты (возможно неосознанно) активно работают над методами осуществления метасистемных переходов на уровне алгоритмов. Это, в первую очередь, проявляется в том, что они активно используют уже разработанные алгоритмы и системы, стараются выделить классы задач, найти для них общее приемлемое решение и опубликовать (продать) его в стандартном удобном виде. Среди программистов популярно слово “reuse” — повторное использование, что подразумевает модульное программирование, где каждый модуль предоставляет достаточно общий инструментарий, который может быть использован при создании других модулей. Суть метода, который скрывается за термином “reuse”, можно выразить следующим образом:

“При решении задач используйте существующий стандартный функционал и экспортируйте результаты как расширение стандартного функционала.”

Именно благодаря методу meta людям удается решать сложные задачи и создавать сложные компьютерные системы.

Метод meta отличается от методов, рассмотренных выше. Он несёт в себе более общую идею и является скорее основополагающим принципом построения (эволюции) сложных систем, нежели методом конструирования приближённых алгоритмов сложных задач. Но именно про это применение метода meta здесь идёт речь. Не так часто программисты в рамках одного алгоритма (одной задачи) осмеливаются применять метасистемные переходы. Обычно метасистемые переходы применяются в процессе создания алгоритма. А именно, сначала к задаче пробуют подступиться с разных сторон и написать несколько черновых алгоритмов, основанных на разных принципах. Затем из них выбирают наиболее перспективный и переписывают его аккуратно, выделяют структуру, находят ярко выраженные подзадачи. Затем анализируют работу получившегося алгоритма, дополнительно оптимизируют узкие места и переписывают кусочки, которые оказались несостоятельными. Тщательно анализируются параметры, отвечающие за баланс “качество–время работы”

и выбираются наиболее подходящие. То есть большую часть работы на meta-уровне стремится на себя взять программист. Внедрение meta-компоненты в сами программы активно началось с развитием генетических алгоритмов. Сегодня генетические алгоритмы можно встретить и в серьезных промышленных системах, например базе СУБД Postgres, поисковых системах, системах распознавания образов и в ядре операционных систем.

Можно указать и множество других идей полезных при решешении сложных программистких задач, программировании вообще, или даже при решении произвольных жизненных задач. Но несмотря на всю общность, они более "мелкокалиберные" нежели рассмотренные выше методы.

active: Запоминай активно-используемые значения

extreme: Уделяй внимание феноменальным явлениям и отклонениям от среднего ("принцип крайнего")

invariants: Метод инвариантов

В этот список можно внести и другие ключевые общие математические методы решения задач.

3 Базовые $\mathcal{NP}$-сложные задачи

Естественным шагом является создание библиотек приближённых алгоритмов решения наиболее популярных $\mathcal{NP}$-сложных задач.

Но здесь мы встречаемся с рядом проблем. Во-первых, многообразие $\mathcal{NP}$-сложных задач велико, и по множеству различных форм и деталей, превосходит множество типичных полиномиально-разрешимых задач. Их сложнее классифицировать, разбить на классы и выделить наиболее типичные. Более того, очень часто одна и та же задача но с разными особенностями входных данных может эффективно решаться принципиально разными приближенными алгоритмами.

Принцип "reuse" часто не оправдывает себя (не срабатывает), когда речь идет о более-менее сложных алгоритмах[3]. Многие классические алгоритмы программисты предпочитают реализовывать сами. Во-первых, потому что им приятно реализовать известный классический алгоритм и оптимизировать его чуть-чуть, используя специфику входных данных и имеющихся вычислительных мощностей. Во-вторых, очень часто возникает необходимость добавить в алгоритм немного "своей специфики", а стандартная библиотека алгоритмов может этого не позволить. Наконец, многим просто не хочется подстраиваться под чужие инструменты. Зависимость от внешних инструментов приносит дополнительные хлопоты. Например, формат входных и

[3] Это верно лишь для алгоритмов средней сложности. Когда встречается действительно сложная задача и имеется каноническая открытая система (программа), её решающая, программисты с удовольствием её используют.

выходных данных у внешних инструментов обычно отличается от внутреннего формата, и поэтому требуется постоянная конвертация данных.

По этим причинам экспорт на уровне функций и алгоритмов (“reuse в малом”) не так популярен. Гораздо удобнее импортировать/экспортировать функциональность в виде полнофункциональных систем, решающих конкретную задачу. Но проблема в том, что конкретных задач слишком много, чтобы под каждую из них делать отдельную систему. Удивительно высокий уровень экспорта стандартной функциональности был достигнут в Unix-системах, в которых зародилась и развилась культура небольших, но достаточно общих и типичных для большого класса задач инструментов.

В области классических алгоритмов такой общий инструментарий представлен структурами данных и алгоритмами, связанными с коллекциями. Также активно используется функция сортировки, хэш-таблицы, очередь с приоритетами и деревья поиска. И на этом, практически, список базисных элементов полиномиального программирования заканчивается. Конечно, часто возникает ситуация, когда берётся чей-то инструментарий, который затем дорабатывается, меняется и только после этого используется. Но в конце работы не редко высказываются мысли, что сделать всё самим с нуля было бы проще (программисты — большие любители “варить кашу из топора”).

Практика показывает, что алгоритмы на графах не попадают в список активного экспорта и импорта — их предпочитают писать с нуля, либо брать чужие исходники и подстраивать под свою задачу.

Эта обратная сторона медали универсальных средств многим известна. “Универсальный” часто означает “много для чего разработанный”, но на практике не удобный ни для того, ни для другого.

И особенно тяжело обстоит дело с универсальными инструментами в области $\mathcal{NP}$-программирования. Несмотря на то, что есть множество общих методов, подобных перечисленным выше, сложно реализовать действительно универсальный набор инструментов в помощь программисту, решающему $\mathcal{NP}$-сложные задачи.

Но тем не менее, попытки в этом направлении делаются. Приведём несколько наиболее популярных $\mathcal{NP}$-сложных задач, приближенные решения которых могли быть включены в базис инструментов $\mathcal{NP}$-программирования:

- Кластеризация — поиск групп точек в данном метрическом пространстве, которые отделены от остальных точек.
- Поиск выделенных компонент графа — поиск групп вершин графа которые сильно связаны друг с другом. Выделенность группы вершин означает, что число рёбер, соединяющих пары вершин в группе, велико относительно числа рёбер идущих из вершин группы в другие вершины. Эта задача близка к задаче кластеризации.
- Нечёткий поиск — быстрый поиск в хранилище таких объектов (документов), которые удовлетворяют как можно большему числу указанных

в запросе признаков (условий).

- Задача коммивояжера — поиск пути (или цикла) наименьшего веса, проходящего по всем вершинам.
- Поиск минимума — поиск минимума данной функции с большим числом аргументов. Эту задача, решаемая обычно методом градиентного спуска, для сложных случаев может быть приближенно решена методом отжига в совокупности с методом масштабирования.
- "Задача о рюкзаке" — найти для данного набора объектов с указанными массами и ценностями подмножество максимальной ценности с суммарной массой меньшей некоторой заданной величины.

Разработано уже несколько достаточно общих инструментов: инструменты для кластеризации, классификации и распознавания, решение задачи нечёткого поиска и жадные алгоритмы решающие задачу коммивояжера [16].

Список литературы

[1] Graphviz – Graph Visualization Software, http://graphviz.org

[2] Metaheuristics, Progress as Real Problem Solvers Series: Operations Research /Computer Science Interfaces Series, Vol. 32 Ibaraki, Toshihide; Nonobe, Koji; Yagiura, Mutsunori (Eds.) 2005, XII, 414 p. 106 illus., Hardcover.

[3] Essays and Surveys in Metaheuristics Series: Operations Research/Computer Science Interfaces Series, Vol. 15 Ribeiro, Celso C.; Hansen, Pierre (Eds.) 2001, 664 p., Hardcover.

[4] *Дейкстра Э.* Заметки по структурному программированию (в составе сборника "Структурное программирование", – М.: "Мир", 1975.

[5] *Йордан Э.* Структурное проектирование и конструирование программ, – М.: Мир, 1979.

[6] *Johnson, D. S.* A catalog of complexity classes, in Algorithms and Complexity, volume A of Handbook of Theoretical Computer Science, Handbook of Theoretical Computer Science, Elsevier science publishing company, Amsterdam, pp. 67-161, 1990.

[7] A compendium of $\mathcal{NP}$ optimization problems, Editors: Pierluigi Crescenzi, and Viggo Kann, `http://www.nada.kth.se/~viggo/wwwcompendium/`

[8] *Турчин В. Ф.* Феномен науки: Кибернетический подход к эволюции, М.: Наука, 295 с., 1993, `http://pespmc1.vub.ac.be/POSBOOK.html`.

[9] *Turchin V. F.* A Dialogue on Metasystem Transition // Heylighen F., Joslyn C., Turchin V. F. (eds.): The Quantum of Evolution. Toward a theory

of metasystem transitions, (Gordon and Breach Science Publishers, New York) (special issue, Vol. 45:1–4, of "World Futures: the journal of general evolution"), pp. 5–58, 1995.

[10] *Petit J.* "Experiments on the Minimum Linear Arrangement Problem", Technical report LSI-01-7-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, 2001.

[11] *Koren Y., Harel D.* "Multi-Scale Algorithm for the Linear Arrangement Problem", Technical Report MCS02-04, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, 2004.

[12] *Atkins J. E., Boman E. G., Hendrickson B.* "A Spectral Algorithm for Seriation and the Consecutive Ones Problem", SIAM Journal on Computing 28 (1998), 297–310.

[13] *Karp R. M.* "Reducibility among combinatorial problems", Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Eds. Plenum, New York, pp. 85103, 1972.

[14] *Garey M. R., Johnson D. S.* Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness, W. H. Freeman and Company, NY, 1979.

[15] *Even G., Naor J. S., Rao S., Scieber B.* "Divide-and-Conquer approximation algorithms via spreading matrices", in Proc. 36th Annual IEEE Symposium on Foundation of Computer Science, IEEE Computer Society Press, pp. 62–71, 1995.

[16] *Ворожцов А.В.* Два приближенных алгоритма решения задачи коммивояжера, Моделирование процессов управления. М.:МФТИ, 2004.

[17] *Ворожцов А.В.* Критерии интеллектуальности искусственных систем, препринт **№60** ИПМ (2004).