ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА КОМПОНЕТА РЕАЛИЗАЦИИ МЕТОДА ОТЖИГА ДЛЯ ПЛАТФОРМЫ .NET

Магистерская диссертация
обучающегося по направлению подготовки 02.04.01 Математика и компьютерные науки
очной формы обучения, группы 07001531
Бондаренко Вадима Юрьевича

Научный руководитель
к.т.н., доцент
Бурданова Е.В.

Рецензент
к.т.н., доцент
Прохоренко Е.И.

БЕЛГОРОД 2017

РАЗРАБОТКА КОМПОНЕНТА РЕАЛИЗАЦИИ МЕТОДА ОТЖИГА ДЛЯ ПЛАТФОРМЫ .NET

Аннотация магистерской диссертации

магистранта очной формы обучения направления подготовки 02.04.01

«Математика и компьютерные науки» 2 года группы 07001531

Бондаренко Вадима Юрьевича

Диссертация 59 с., 12 рис., 2 табл., 22 источника.

Ключевые слова: метод имитации отжига, оптимизация, платформа .Net Framework, библиотека классов, задача о N ферзях.

Объектом исследования является проблематика компьютерной реализации оптимизационных методов на платформе .Net.

Цель работы - разработка компонента реализации метода имитации отжига, для платформы .NET.

Во введении раскрывается актуальность исследования по выбранному направлению, ставится проблема, цель и задачи исследования, определяются объект, предмет научных поисков, ставятся цель и задачи, указывается теоретическая и практическая значимости исследования.

В первой главе описаны основные сведения об оптимизации, такие как: постановка задачи, проблематика, классификация. Так же производится сравнительная характеристика алгоритма отжига и генетического алгоритма. Определены основные схемы отжига.

Во второй главе представлен разбор основных возможностей платформы .Net Framework. Еще в данной главе описано моделирование различных схем отжига.

В третьей главе представлена разработка библиотеки, реализующей метод имитации отжига, а также разработка приложения для тестирования данной библиотеки. Далее описан процесс тестирование выходного продукта и сравнение различных методов отжига реализованных в библиотеке.

Заключение посвящено основным выводам и итогам посвященным проделанной работе.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последние годы интенсивно развиваются алгоритмы поисковой оптимизации, которые называют поведенческими, интеллектуальными, мета-эвристическими, вдохновленными (инспирированными) природой, роевыми, многоагентными, популяционными и т. д. Эффективность таких алгоритмов соизмерима, а часто превосходит эффективность ставших уже классическими эволюционных алгоритмов. Еще в 1993 Л. Инберг рядом тестов смог доказать превосходство метода отжига над генетическими алгоритмами. [2]

Реализация данного оптимизационного метода, как универсальной библиотеки для платформы .Net, а также сравнительный анализ различных вариаций метода, является актуальной задачей на данный момент, так как метод симуляции отжига является одним из самых универсальных, но при этом малоизученных [1], также реализация универсальной библиотеки в будущем существенно сократит время разработки и решения оптимизационных задач для выбранной платформы.

Научная новизна данной работы состоит в предложенном решении ряда оптимизационных задач на платформе .Net.

Научно-практическая ценность результатов диссертации заключается в возможности применения разработанного компонента для решения различного рода задач оптимизации по средствам платформы .Net.

Целью научно-исследовательской работы является разработка компонента реализации метода имитации отжига, для платформы .NET.

Основными задачами работы являются:

- Исследование метода имитации отжига;
- Разработка и внедрения компонента в платформу .NET;
- Создание среды тестирования компонента (задача о N ферзях);
- Сравнительный анализ различных вариаций метода.

В данном исследовании:

- разрабатывается универсальная библиотека классов;

- создается программный продукт для тестирования разработанной модели.

- выполняется сравнительный анализ различных вариаций метода отжига;

Объект исследования – проблематика компьютерной реализации оптимизационных методов на платформе .Net.

Предмет исследования – разработка и применение универсального компонента реализации метода имитации процесса отжига.

Данная исследовательская работа состоит из двух основных частей: теоретической и практической.

В теоретической части научно-исследовательской работы выполняется исследование и сравнение различных методов оптимизации.

В практической части ведется разработка компонента имитации отжига и его тестирование на примере задачи о N ферзей.

ГЛАВА 1. ОСНОВНЫЕ СВЕДЕНИЯ ОБ ОПТИМИЗАЦИИ

Многие сферы деятельности человека в настоящее время, такие как разработка новых материалов и устройств, телекоммуникационные сети требуют решения оптимизационных задач. Как известно, подобные задачи сводятся к поиску экстремумов целевой функции различными методами. Наличие всевозможных ограничений на оптимизируемые параметры и многоэкстремальность целевой функции, как правило, приводят к большим вычислительным затратам и соответственно к невозможности нахождения решения за приемлемое время при использовании одного компьютера [3].

1.1 Постановка задачи оптимизации

Постановка любой задачи оптимизации начинается с определения набора независимых переменных и обычно включает условия, которые характеризуют их приемлемые значения. Эти условия называют ограничениями задачи. Еще одной обязательной компонентой описания является скалярная мера «качества», именуемая целевой функцией и зависящая каким-то образом от переменных. Решение оптимизационной задачи — это приемлемый набор значений переменных, которому отвечает оптимальное значение целевой функции. Под оптимальностью обычно понимают максимальность или минимальность; например, речь может идти о максимизации прибыли или минимизации массы.

К задачам на поиск оптимума сводятся многие из проблем математики, системного анализа, техники, экономики, медицины и статистики. В частности, они возникают при построении математических моделей. Когда для изучения какого-нибудь сложного явления конструируется математическая модель, к оптимизации прибегают для того, чтобы определить такую структуру и такие параметры последней, которые

обеспечивали бы наилучшее согласование с реальностью. Другой традиционной областью применения оптимизации являются процедуры принятия решений, так как большинство из них нацелено именно на то, чтобы сделать «оптимальный» выбор. Помимо оптимизационных задач, представляющих самостоятельный интерес, на практике часто возникают задачи, которые «встроены» в некоторые вычислительные процессы, где они играют хотя и существенную, но все же вспомогательную роль. Это настолько типичная ситуация, что при описании процесса в целом далеко не всегда указывают на наличие подобных задач; к примеру, сказав, что процесс предполагает определение точки, в которой какая-то функция достигает своего критического значения, могут и не добавить, что эта точка будет найдена решением оптимизационной задачи [4].

В задачах оптимизации рассматриваемые варианты должны быть сравнимы. Наиболее подходящей является оценка варианта числом. Довольно часто чем больше числовая характеристика, тем лучше, например, прибыль предприятия, средний балл поступающего п учебное заведение. Однако потерн электроэнергии в энергетической системе, погрешности измерительного прибора должны быть, по возможности, меньше.

1.2 Основные проблемы оптимизации

Решая оптимизационную задачу можно столкнуться с рядом проблем.

Первая из них связана с существованием оптимальных решений задачи. При анализе этой проблемы в ряде случаев может оказаться полезной следующая теорема, с помощью которой можно иногда определить, достигает ли функция своего глобального минимума и максимума.

Вторая проблема связана с поиском условий, которым должно удовлетворять оптимальное решение задачи.

Третья проблема состоит в том, как найти хотя бы одно такое решение [5, 6].

Одной из основных проблем при решении какой-либо задачи оптимизации является выбор метода решения [7].

Когда оптимизационная задача поставлена, приходит черед выбирать метод ее решения. Прежде всего при этом следует учесть основные характеристики целевой функции и функций ограничений. По ним все задачи разбираются на классы, каждому из которых отвечает своя группа предпочтительных алгоритмов. Однако речь идет именно о группе, так что проблема выбора остается и после определения класса задачи. На каком методе следует остановиться, зависит от того, какую информацию о производных можно предоставить, каков имеющийся объем машинной памяти и как соотносятся трудоемкости вычисления функций и алгебраических блоков сопоставляемых схем.

Самое общее правило выбора звучит следующим образом: чем больше информации о производных, которую можно получить ценой приемлемых затрат, будет использовано при решении задачи, тем лучше. В частности, отказываться от процедуры, требующей аналитические значения градиентов, только потому, что для подсчета этих значений придется писать отдельную программу, неразумно. Такая экономия усилий скорее всего обернется потерями — ведь методы поиска экстремума без применения производных сходятся медленнее и не столь надежны, как градиентные. Аналогично дело обстоит и с матрицами Гессе: если их можно вычислять и если это не в сотни раз сложнее, чем считать градиенты, то пренебрегать ими не следует.

1.3 Примеры реализации оптимизационных методов

Оптимизация в широком смысле слова находит применение в науке, технике, экономике и других областях человеческой деятельности. К оптимизационным задачам относятся, например:

- задачи оптимального планирования деятельности предприятий;

•задачи оптимального прикрепления потребителей к поставщикам (транспортная);

• задачи оптимального распределения трудовых ресурсов;

• задача оптимального составления смесей;

• бинарные задачи распределения;

• задачи о раскрое;

• задачи формирования оптимального портфеля ценных бумаг (инвестиционных проектов) и др [3].

Поиски оптимальных решений привели к созданию специальных математических методов. В качестве инструмента решения оптимизационных задач используется математическое программирование (планирование). До второй половины XX века методы оптимизации во многих областях науки и техники применялись достаточно редко, поскольку практическое использование математических методов оптимизации требовало огромной вычислительной работы, которую без ЭВМ реализовать было крайне трудно, а в ряде случаев и невозможно. С появлением компьютеров для решения таких задач используются специализированные пакеты прикладных программ, языки программирования высокого уровня. Важное место в курсе информатики занимает раздел моделирования. Применение математических моделей позволяет использовать средства вычислительной техники для анализа допустимых решений, поиска наиболее рационального оптимального решения.

При решении конкретной задачи оптимизации исследователь прежде всего должен выбрать математический метод, который приводил бы к конечным результатам с наименьшими затратами на вычисления или же давал возможность получить наибольший объем информации об искомом решении. Выбор того или иного метода в значительной степени определяется постановкой оптимальной задачи, а также используемой математической моделью объекта оптимизации.

1.4 Классификация методов оптимизации

В настоящее время для решения оптимальных задач применяют в основном следующие методы:

- методы исследования функций классического анализа;
- методы, основанные на использовании неопределенных множителей Лагранжа;
- вариационное исчисление;
- динамическое программирование;
- принцип максимума;
- линейное программирование;
- нелинейное программирование.

В последнее время разработан и успешно применяется для решения определенного класса задач метод *геометрического программирования*.

На рисунке 1.1 изображено дерево оптимизации.

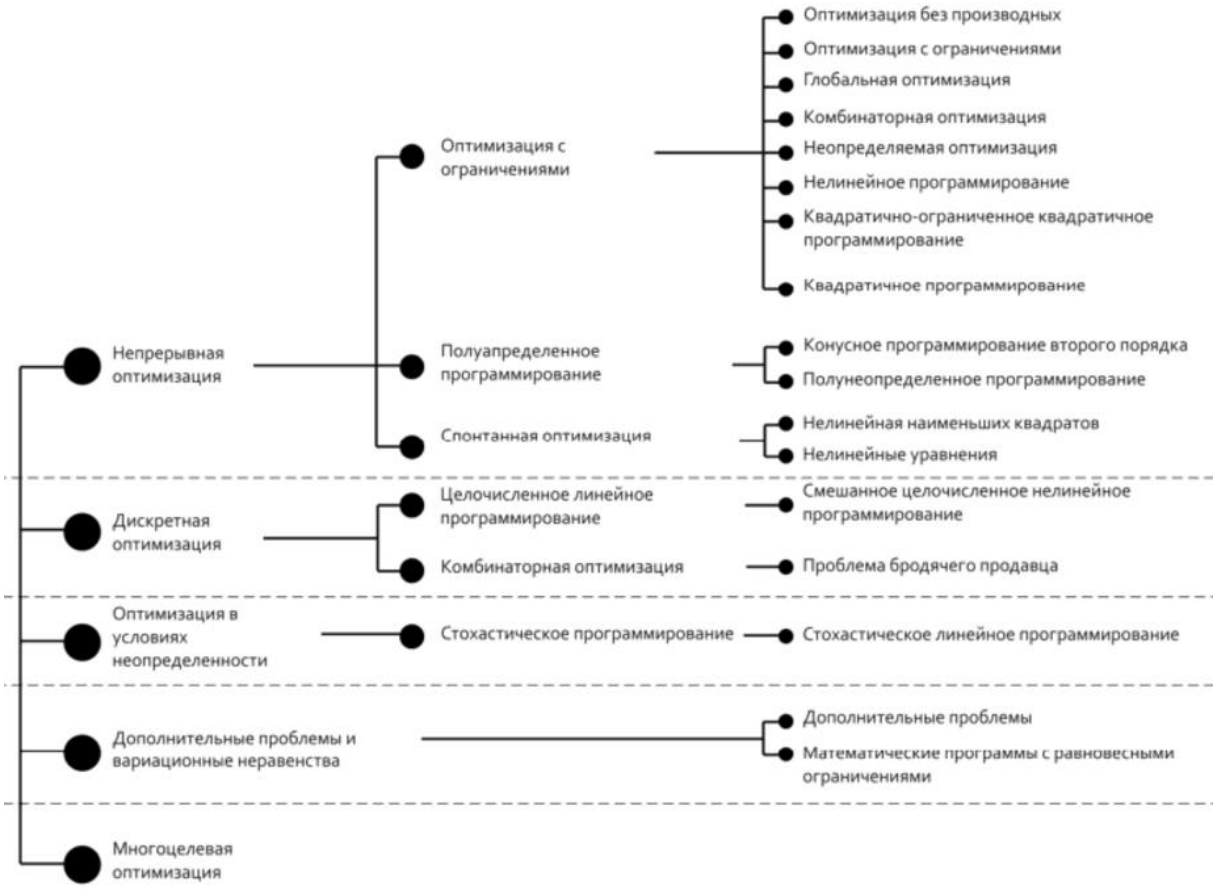

Рисунок 1.1. Дерево оптимизации

Как правило, нельзя рекомендовать какой-либо один метод, который можно использовать для решения всех без исключения задач, возникающих на практике. Одни методы в этом отношении являются более общими, другие - менее общими. Наконец, целую группу методов (методы исследования функций классического анализа, метод множителей Лагранжа, методы нелинейного программирования) на определенных этапах решения оптимальной задачи можно применять в сочетании с другими методами, например, динамическим программированием или принципом максимума.

Отметим также, что некоторые методы специально разработаны или наилучшим образом подходят для решения оптимальных задач с математическими моделями определенного вида. Так, математический аппарат линейного программирования, специально создан для решения задач с линейными критериями оптимальности и линейными ограничениями на переменные и позволяет решать большинство задач, сформулированных в такой постановке. Так же и геометрическое программирование предназначено для решения оптимальных задач, в которых критерий оптимальности и ограничения представляются специального вида функциями позиномами [3].

Динамическое программирование хорошо приспособлено для решения задач оптимизации многостадийных процессов, особенно тех, в которых состояние каждой стадии характеризуется относительно небольшим числом переменных состояния. Однако при наличии значительного числа этих переменных, т. е. при высокой размерности каждой стадии, применение метода динамического программирования затруднительно вследствие ограниченных быстродействия и объема памяти вычислительных машин.

1.5 Генетический алгоритм

Генетический алгоритм — это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём

случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как наследование, мутации, отбор и кроссинговер. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе [7].

Генетические алгоритмы служат, главным образом, для поиска решений в многомерных пространствах поиска.

Можно выделить следующие этапы генетического алгоритма:

1) Задать целевую функцию (приспособленности) для особей популяции
2) Создать начальную популяцию (Начало цикла)
    a. Размножение (скрещивание)
    b. Мутирование
    c. Вычислить значение целевой функции для всех особей
    d. Формирование нового поколения (селекция)
3) Если выполняются условия остановки, то (конец цикла), иначе (начало цикла).

Основными проблемами данного алгоритма являются:

1) Плохая масштабируемость
2) Сходимость к локальному оптимуму

1.6 Метод отжига

Алгоритм имитации отжига (Simulated annealing) — алгоритм решения различных оптимизационных задач. Он основан на моделировании реального

физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твёрдое, в том числе при отжиге металлов.

Целью алгоритма является минимизация некоторого функционала. В процессе работы алгоритма хранится текущее решение, которое является промежуточным результатом. А после работы алгоритма оно и будет ответом [1].

Стоит отметить, что метод отжига может быть эффективным при решении задач различных классов, требующих оптимизации. Ниже приводится их краткий список:

- создание пути;
- реконструкция изображения;
- назначение задач и планирование;
- размещение сети;
- глобальная маршрутизация;
- обнаружение и распознавание визуальных объектов;
- разработка специальных цифровых фильтров.

В отличие от вышеописанного генетического алгоритма, метод имитации отжига «не страдает» от попадания в локальный минимум. Это одно из главных преимуществ алгоритма отжига перед многими оптимизационными методами. Следует отметить, что и данный метод хорошо масштабируется в отличии от вышеупомянутого генетического алгоритма.

Метод имитации отжига имеет несколько вариаций:

- Больцманов отжиг;
- Отжиг Коши (быстрый отжиг);
- Сверхбыстрый отжиг;
- Алгоритм Ксин Яо.

Эти алгоритмы промоделированы в главе 2, а сравнительная характеристика на примере задачи о N ферзях продемонстрирована в третей главе данной работы.

ГЛАВА 2. ТЕХНОЛОГИИ СОЗДАНИЯ БИБЛИОТЕК КЛАССОВ И МОДЕЛИРОВАНИЕ СХЕМ ОТЖИГА

2.1 Microsoft .NET Framework

.NET Framework — это платформа для построения и исполнения приложений. Ее основные компоненты — общеязыковая исполняющая среда (common language runtime, CLR) и библиотека классов .NET Framework (FCL). Для разработки компонента и дальнейшего его тестирования была выбрана среда MS Visual Studio 2015. Окно среды MS VS2015 изображено на рисунке 2.1. CLR абстрагирует сервисы ОС и служит механизмом для исполнения управляемых приложений (managed applications), любое действие которых должно получить одобрение со стороны CLR. FCL предоставляет объектно-ориентированный API, к которому обращаются управляемые приложения. При написании приложений для .NET Framework вы отказываетесь от Windows API, MFC, ATL, COM и других знакомых инструментов и технологий и взамен используете FCL. Конечно, вы сможете задействовать API Windows или COM-объект, но вы этого не захотите, так как это потребует перехода отяжеляемого кода (кода, исполняемого CLR) к неуправляемому коду («родному» машинному коду, исполняющемуся без помощи CLR). Такие переходы негативно сказываются на производительности и могут быть даже запрещены системным администратором. [10]

В основном Microsoft .NET — это Web-сервисы XML, но .NET Framework поддерживает и другие программные модели. В дополнение к Web-сервисам вы можете писать консольные приложения, приложения с графическим интерфейсом пользователя (Windows Forms), Web-приложения (Web Forms) и даже службы Windows, более известные как службы NT. Инфраструктура также помогает потреблятъ Web-сервисы, т. е. писать

клиенты Web-сервисов. Однако приложения, написанные на основе .NET Framework, не обязаны использовать Web-сервисы.

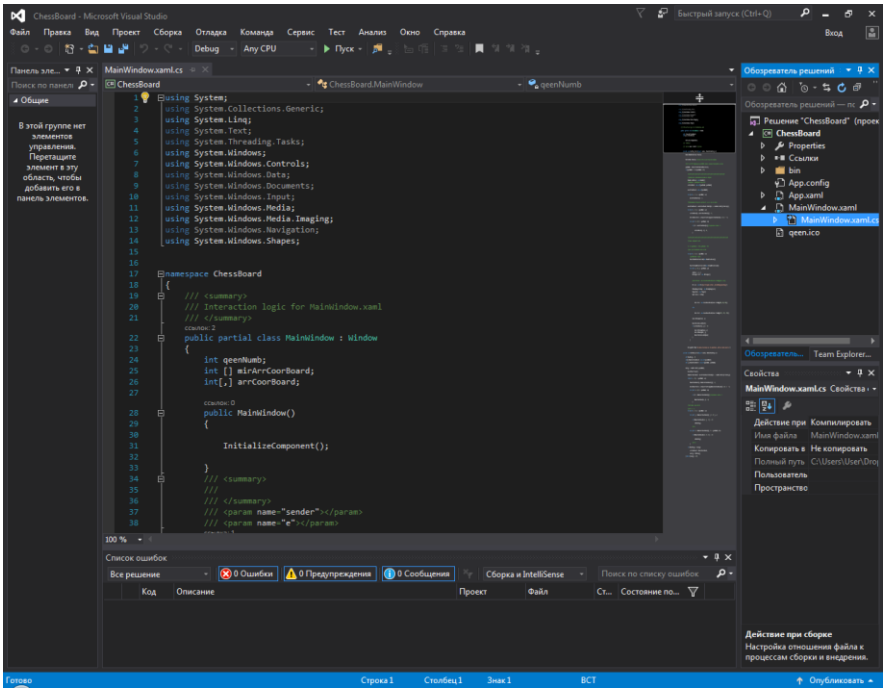

Рис. 2.1. Окно среды программирования VS 2015

Наибольший после Web-сервисов потенциал заложен в ASP.NET. Это название произошло от Active Server Pages (ASP) — технологии которая стала революцией в Web-программировании 90-х, предоставив простую модель динамической генерации HTML-страниц Web-серверами с помощью серверных сценариев. ASP.NET - это следующая версия ASP, предоставляющая новый удобный способ написания Web-приложений, не имеющий аналогов в прошлом. [11]

2.1.1 Общеязыковая исполняющая среда (CLR)

Сердце и душа .NET Framework — CLR. Каждый байт кода, написанный для этой инфраструктуры, либо исполняется CLR. либо получает ее разрешение на исполнение за ее пределами. Ничто не происходит без участия CLR.

CLR расположена поверх ОС и предоставляет виртуальную среду для управляемых приложений. При запуске управляемой программы CLR загружает содержащий ее модуль и исполняет его код. Код, предназначенный для CLR, называется управляемым кодом и состоит из команд псевдомашинного языка — общего промежуточного языка (common intermediate language, CIL). Команды CIL компилируются в машинный код (обычно код процессора x£6) по запросу (just-in-time) в период выполнения. Обычно компиляция любого метода происходит лишь раз — при первом его вызове, и затем результат компиляции кэшируется в памяти, чтобы при повторном вызове он мог быть исполнен без задержки. Код, который никогда не вызывается, никогда и не компилируется. Хотя компиляция по запросу, несомненно, снижает производительность, эти накладные расходы компенсируются тем, что на протяжении исполнения приложения каждый метод компилируется не более раза, а также огромными усилиями разработчиков CLR сделать ЦТ-комиилятор как можно быстрее и эффективнее. Теоретически производительность кода, скомпилированного по запросу, может быть выше производительности обычного кода, так как JIT-компилятор способен выполнить оптимизацию специально для того процессора, на котором он в данный момент исполняется. Скомпилированный по запросу код — это не то же самое, что интерпретируемый код, и не верьте тому, кто скажет обратное.

Преимуществ исполнения кода в управляемой среде CLR масса. Преобразуя команды CIL в команды процессора, JIT-компилятор выполняет верификацию кода на предмет безопасности типов. Практически невозможно выполнить команду, обращающуюся к области памяти, к которой у этой команды нет разрешения на доступ. В управляемом приложении не бывает проблем ошибочно инициализированных указателей, так как CLR генерирует исключение, прежде чем такой указатель может быть использован. Нельзя преобразовать тип в нечто, чем он не является, так как это нарушение безопасности типов. И вы не сможете вызвать метод с разрушенным

стековым фреймом, так как CLR просто не даст этому случиться. Помимо устранения наиболее распространенных ошибок, влияющих на приложения, процедура верификации кода значительно затрудняет написание вредоносных программ, пытающихся разрушить систему. В том маловероятном случае, когда верификация кода не нужна, ее может отключить системный администратор,

Верификация кода также является основой способности CLR исполнять несколько приложений внутри одного процесса — трюк состоит в разделении процесса на виртуальные отсеки — домены приложений (application domains). Windows изолирует одно приложение от другого, размещая их в отдельных процессах. Отрицательный побочный эффект, присущий модели «одно приложение — один процесс», — больший расход памяти. Эффективность использования памяти не очень важна на изолированном компьютере, обслуживающем одного пользователя, но на серверах, обслуживающих тысячи пользователей одновременно, она ставится во главу угла. Иногда (главные примеры — приложения ASP.NET и Web-сервисы) CLR не создает нового процесса для каждого приложения — вместо этого создается один или небольшое число процессов, и отдельные приложения размещаются в доменах приложений. Домены приложений не менее безопасны, чем процессы, так как их границы не могут быть нарушены управляемыми приложениями. Вместе с тем домены приложений эффективнее процессов, так как в одном процессе может размещаться несколько доменов приложений, а библиотеки могут загружаться в домены приложений и использоваться совместно.

Еще одно преимущество исполнения в управляемой среде в том, что ресурсы, выделяемые управляемым кодом, освобождаются сборщиком мусора. Иначе говоря, вы выделяете память, но не освобождаете ее — за вас это делает ОС. В состав CLR входит совершенный сборщик мусора, отслеживающий ссылки на объекты, создаваемые вашей программой, и

уничтожающий эти объекты, когда занимаемая ими память требуется где-нибудь еще.

Благодаря сборщику мусора, в приложениях, состоящих только из управляемого кода, не бывает утечек памяти. Сборка мусора даже повышает производительность, так как алгоритм выделения памяти, реализованный в CLR, очень быстр — гораздо быстрее, чем аналогичные функции выделения памяти библиотеки периода выполнения языка С. Обратной стороной является то. что при сборке мусора все остальное в этом процессе на мгновение приостанавливается. К счастью, мусор собирается относительно редко, что существенно снижает влияние этого процесса на производительность.

2.2 Языки программирования

Еще одно преимущество исполнения приложений в среде CLR в том. что весь код компилируется в CIL, поэтому выбор языка программирования становится практически вопросом личных предпочтений. «Общеязыковая» в словосочетании «общеязыковая исполняющая среда» указывает на то, что CLR безразлична к языку' программирования. В других средах язык, на котором написано приложение, неизбежно влияет на структуру и работу последнего. Так, в программе на Visual Basic сложно запускать новые потоки. Хуже того, современные языки, такие как Visual Basic и Visual C++, используют разные API, и поэтому знания, приобретенные вами при написании Windows-программ на Visual Basic, будут стоить весьма немного, когда шеф попросит вас написать DLL на C++. [12]

NET Framework все меняет. Язык — это просто синтаксическое устройство для генерации CIL и за немногочисленными исключениями все, что можно сделать на одном языке, возможно и на всех остальных. Более того, независимо от языка, на котором они написаны, все управляемые приложения используют один и тот же интерфейс прикладного

программирования: API библиотеки классов .NET Framework. Перенос приложения Visual Basic 6 на Visual C++ лишь немногим проще написания приложения с нуля. В то же время перенос приложения Visual Basic .NET на C# (или наоборот) гораздо проще. В прошлом несовершенство инструментов для преобразования программ из одного языка в другой делало их практически бесполезными. С появлением .NET Framework кто-то. может быть, напишет действительно работоспособный конвертор. Так как в конечном счете код на языке высокого уровня все равно компилируется в CIL, инфраструктура позволяет даже написать класс на одном языке и затем использовать его (или создать производный класс) на другом.

Microsoft поставляет CIL-компиляторы для пяти языков: C#, J#, C++, Visual Basic и JScript. В .NET Framework Software Development Kit (SDK) входит даже ассемблер CIL — IUSM, так что при желании можно писать прямо на CIL. Другие компании поставляют компиляторы для других языков, включая Perl, Python, Eiffel и даже COBOL. Независимо от того, какой язык вы предпочитаете, почти наверняка для него есть компилятор CIL. И если вы поклонник COBOL, теперь можете спать спокойно зная, что в состоянии сделать практически все то же, что могут эти снобы, программирующие на C#.

2.3 Сборки

Теперь вы знаете, что компиляторы NET Framework генерируют управляемые модули и что эти модули содержат CIL и метаданные. Однако вас может удивить, что CLR неспособен использовать управляемые модули напрямую. Дело в том, что базовой единицей защиты, управления версиями и развертывания в .NET Framework является не управляемый модуль, а сборка (assembly).

Сборка — это файл или набор файлов, в совокупности составляющих логическую единицу. В данном контексте термином файлы обозначаются

главным образом управляемые модули, но в сборку могут входить и иные файлы. Большинство сборок содержит один файл, но может содержать и иногда содержит несколько файлов. Все файлы в составе одной сборки должны находиться в одном каталоге. Когда вы с помощью компилятора C# создаете простой EXE, то он является не только управляемым модулем, но и сборкой. Большинство компиляторов в состоянии создавать управляемые модули, не являющиеся сборками, а также добавлять другие файлы к сборкам, которые они создают. В состав .NET Framework SDK входит утилита AL (Assembly Linker) для объединения файлов в сборки. [13, 14]

Многофайловые сборки обычно служат для объединения модулей, написанных на разных языках, и для объединения управляемых модулей с обычными файлами, содержащими изображения в формате JPEG и другие ресурсы, Многофайловые сборки также применяются для разделения приложений на дискретные загружаемые части, что может пригодиться в случае развертывания приложения через Интернет. Представьте себе, например, что кто-то пытается загрузить многомегабайтное приложение, состоящее из одного файла-сборки, по коммутируемой телефонной линии, Загрузка такого кода может длиться вечность. Снизить остроту проблемы могло бы разделение кода на несколько файлов, являющихся частями одной сборки. Так как неиспользуемые модули не загружаются, пользователю не придется ждать окончания загрузки тех частей приложения, которые ему не нужны. Если код приложения удачно разбит на части, загрузка большей его части может вообще никогда не понадобиться.

Как же CLR узнает, какие файлы относятся к сборке? Один из файлов, входящих в сборку содержит декларацию (manifest). Физически декларация — это просто дополнительные метаданные. Когда компилятор создает управляемый модуль, одновременно являющийся и сборкой, декларация просто помещается в метаданные модуля. Логически декларация — это путеводитель по содержимому сборки. Ее наиболее важные элементы:

- имя сборки;

• список всех остальных файлов сборки вместе с криптографическими хэш-значениями, вычисленными по содержимому файлов;

• список типов данных, экспортируемых другими файлами сборки, и информация, связывающая эти типы данных с файлами, где они определены-.

• номер версии в формате major.minor.build.revision (например, 1.0.3705.0). Декларация может содержать и другие сведения, в том числе название компании, описание, требуемые права доступа и строку региональных стандартов. Последняя определяет языковые и другие параметры, для которой предназначена эта сборка (например, «en-US» обозначает «United States English*), и обычно используется сателштными сборками (satellite assemblies), содержащими только ресурсы. На рисунке 2.2 изображена многофайловая сборка, состоящая из трех управляемых модулей и файла JPEG. Main.exe содержит декларацию со ссылками на другие файлы. С точки зрения файловой системы, это по-прежнему отдельные файлы, а с точки зрения CLR — одна логическая единица. [11]

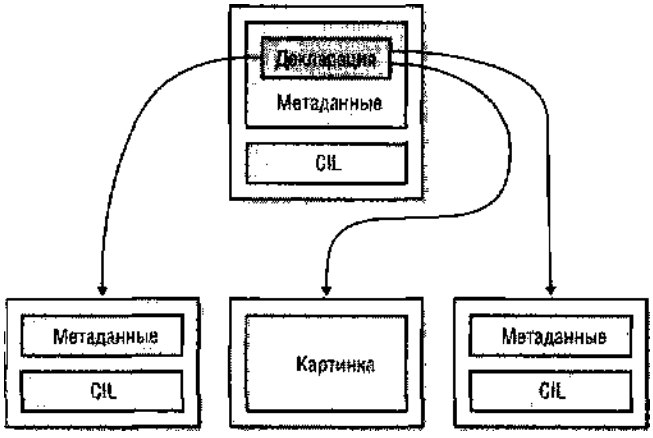

Рис. 2.2. Многофайловая сборка

В отсутствие специальных указаний компиляторы генерируют нестрого именованные (weakly named) сборки. Это значит, что сборка не имеет криптографической подписи и для ее идентификации CLR использует только имя, указанное в декларации (представляющее собой лишь имя файла сборки

без расширения). Однако сборки могут быть и строго именованы (strongly named). Такие сборки содержат открытый ключ своего создателя, а также цифровую подпись, являющуюся хэш-значением, сгенерированным для декларации сборки, в которой хранится открытый ключ.

Цифровая подпись, генерируемая с помощью закрытого ключа создателя сборки, может быть проверена открытым ключом и делает декларацию сборки (а значит, саму сборку) устойчивой к подделкам. Для идентификации строго именованной сборки служат имя сборки, открытый ключ, номер версии и строка региональных стандартов, если она есть. Любое, даже самое малое отличие является достаточным, чтобы отличить две в остальном идентичные сборки.

Для создания строго именованных сборок можно использовать утилиту AL из SDK. Большинство компиляторов, включая C# и Visual Basic .NET. также может генерировать строго именованные сборки. Вы сами принимаете решение, какую сборку — строго или нестрого именованную — использовать. Выбор зависит от назначения сборки. Размещаемая в глобальном кэше сборок (global assembly cache, GAC) — глобальном репозитарии сборок, предназначенных для использования разными приложениями, — сборка должна быть строго именована. [12]

Сборка также должна быть строго именована, если вы хотите использовать контроль версий. При загрузке нестрого именованной сборки CLR не проверяет версию. Это может быть и хорошо, и плохо. Хорошо, если старая версия сборки замещается новой (вероятно, с исправленными ошибками) и приложения, использовавшие эту сборку, должны автоматически начать работать с новой версией. Плохо, если приложение было протестировано для работы с определенной версией сборки, которую затем кто-то заменил на новую, полную ошибок версию. Это один из признаков «ада DLL», столь хорошо знакомого Windows-разработчикам. Строгое именование позволяет его избежать. При загрузке строго именованной сборки CLR сравнивает номер версии загружаемой сборки с

тем номером, для которого компилировалось загружающее эту сборку приложение. (Эти сведения хранятся, как можно догадаться, в метаданных модуля.) Если номера не совпадают, CLR генерирует исключение.

Конечно, в строгом контроле версий есть свои ловушки. Допустим, вы решили использовать строгое именование, но затем обнаружили в своей сборке ошибку. Вы исправляете ошибку и распространяете исправленную сборку. Догадались, что произойдет? Приложения, использующие эту сборку, не смогут загрузить новую версию, если их не собрать заново. Они будут по-прежнему загружать старую версию, и если вы ее удалите, то вообще перестанут работать. Решение состоит в изменении политики связывания GLR. Администратор может относительно легко — редактируя конфигурационный файл — перенаправить CLR на новую версию строго именованной сборки. Конечно, если в новой версии есть ошибки, старая проблема возникает снова. Поэтому-то не следует предоставлять административные права всем и каждому.

Процесс работы со сборками выглядит довольно сложным, и иногда так оно и есть. К счастью, если вы не создаете совместно используемые сборки или сборки, связываемые с другим сборками (отличными от FCL, которая, кстати, представляет собой набор совместно используемых сборок), то большинство проблем с именами и связыванием вас не коснется. Вы просто вызываете свой компилятор, копируете получившуюся сборку в нужный каталог и запускаете ее — проще некуда.

2.4 Библиотека классов .NET Framework

C-программисты, пишущие для Windows, обычно используют в своих программах вызовы API Windows и DLL других производителей: C++-программисты часто используют библиотеки классов, написанные ими самими или стандартные библиотеки типа MFC. Visual Basic-программисты

используют API Visual Basic, представляющие абстракцию низкоуровневых API ОС.

Работая с .NET Framework, можно забыть обо всех этих устаревших API. Вам нужно выучить совершенно новый API — библиотеку классов .NET Framework, которая содержит более 7 000 типов: классов, структур, интерфейсов, перечислений и делегатов (так называются оболочки функций обратного вызова, обеспечивающие безопасность типов). Некоторые классы FCL содержат до 100 методов, свойств и других членов, так что изучить FCL нелегко. Плохо, что это подобно- изучению новой ОС, но прелесть в том, что все языки используют один и тот же API, поэтому ваши усилия по изучению FCL не пропадут, если ваша компания решит перейти с Visual Basic на C++ или наоборот.

Чтобы облегчить изучение и использование FCL, Microsoft разделила эту библиотеку на иерархические пространства имен - Таблица 2.1. Всего в FCL около 100 таких пространств. В каждом содержатся классы и другие типы, имеющие некоторое общее назначение. Так, большая часть API Windows для управления окнами инкапсулирована в пространстве имен System.Windows.Forms. Здесь можно найти классы, представляющие окна, диалоги, меню и другие элементы, обычно применяемые в приложениях с графическим интерфейсом пользователя. Отдельное пространство — Syslem.Collections — содержит классы хэш-таблиц, массивов переменной размерности и других контейнеров, а System — классы файлового ввода-вывода. Полный список пространств имен FCL см. в электронной документации Framework SDK. Вам, подающим надежды .NET-программистам, надо познакомиться с ними. К счастью, FCL настолько громадна и всеобъемлюща, что большинству разработчиков нет нужды подробно изучать ее полностью.

В таблице перечислены несколько пространств имен FCL и кратко описано их содержимое. Выражение обозначает пространства-потомки.

Например, SystemData и др.» — это SystemData, SystemData.Common, SystemData.OleDb, SystemData.SqlClent и SystemDataSqlTypes.

Таблица 2.1

Некоторые пространства имен FCL

System	Фундаментальные типы данных и вспомогательные классы
System.Collections	Хэш-таблицы, массивы переменной размерности и другие контейнеры
SystemData	Классы ADO.NET для доступа к данным
System Drawing	Классы для вывода графики (GDI+)
SystemJO	Классы файлового и потокового ввода-вывода
SystemNet	Классы для работы с сетевыми протоколами, например, с HTTP
SysiemReflection	Классы для чтения и записи метаданных
SystemRuntimeRemoting	Классы для распределенных приложений
SvslemServiceProcess	Классы для создания служб Windows
System.Threading	Классы для создания и управления потоками
System.Web	Классы для поддержки HTTP
System.Web.Services	Классы для разработки Web-сервисов
SystemWebServiceProtocols	Классы для разработки клиентов Web-сервисов
System.Web.UI	Основные классы, используемые ASP.NET
SystemWeb.UI.WebControIs	Серверные элементы управления ASP.NET
Svstem.WindowsForms	Классы для приложений с графическим интерфейсом пользователя
SystemXml	Классы для чтения и вывода данных в формате XML

Первое, и самое важное пространство имен FCL, используемое каждым приложением, — это System. Помимо прочего, в нем определены фундаментальные типы данных, необходимые управляемым приложениям: байты, целые, строки и т. д. Когда вы объявляете переменную типа int на C#, фактически создается экземпляр Systemlnt32. Компилятор C# допускает сокращение int, потому что проще написать:

int a = 7;

чем:

System.Int32 a = 7;

В пространстве имен System находятся также многие типы исключений, определенные FCL (например, InvalidCastException), и такие полезные классы, КАК Math, содержащий методы для выполнения сложных математических операций, и Random, в котором реализован генератор псевдослучайных чисел и GC, предоставляющий программный интерфейс сборщика мусора. [13]

Физически FCL представляет собой набор DLL в каталоге SystemRoot/o\Microsoft.NET\Framework. Каждая DLL — это сборка, загружаемая CLR по запросу. Фундаментальные типы данных, такие как Int32, реализованы в Mscorlib.dll, другие типы разбросаны по разным DLL FCL. В документации на каждый тип указана сборка, в которой он определен. Это важно, поскольку вы обязаны указать конкретную сборку, в которой реализован класс FCF, если компилятор сообщает о том, что этот класс является неопределенным типом. Компилятору C# сборки, на которые имеются внешние ссылки, указываются с помощью ключей/r[eference].

Конечно, одной главы (и даже книги) не хватит, чтобы рассмотреть FCL полностью. FCL — это API .NET Framework и что это необычайно обширная библиотека классов. Чем больше вы будете узнавать ее, тем больше она вам будет нравиться, тем выше вы оцените громадную работу, результатом которой стала FCL.

2.5 Моделирование схем отжига

Для сравнения различных схем метода отжига моделировалось поведение их оценок. Вопрос моделирования различных схем отжига разбивается на две части:

•промоделировать соответствующее распределение,

•промоделировать схему изменения температуры.

2.6.1. Больцмановский отжиг

Нормальное распределение моделировалось с помощью центральной предельной теоремы. Обозначим α_i независимые реализации равномерного распределения на [0; 1]. Тогда $\mathbf{E}\alpha_i = 0, \mathbf{D}\alpha_i = 1/12$ и

$$\frac{\sum_{i=1}^{n} \alpha_i}{\sqrt{n/12}} \xrightarrow[n\to\infty]{} \mathbf{N}(0;1)$$

Таким образом, можно сложить п независимых реализаций α_i, и разделив их сумму на $\sqrt{n/12}$, мы будем иметь достаточно хорошее приближение к нормальному распределению. Как показываетпрактика, достаточно взять n = 24. Для моделирования требуемого **N**(0;T) достаточно домножить получившееся число на $\sqrt{T}$.

В случае размерности, большей единицы, по каждой из координат вводились независимые возмущения с таким распределением.

Для моделирования убывания температуры использовалась непосредственно формула $T_0/\ln k, k = 2,3, \ldots$

2.6.2.Отжиг Коши

Распределение Коши моделировалось методом обратных функций. Пусть **X** — случайная величина, имеющая распределение Коши с плотностью

$$g(x) = \frac{1}{\pi}\frac{T}{x^2 + T^2}$$

Вычислим **P**{**X** < y}:

$$\mathbf{P}\{\mathbf{X} < y\} = \int_{-\infty}^{y} \frac{1}{\pi}\frac{T}{x^2 + T^2}\,dx = \frac{1}{\pi}\left(\text{arctg}\left(\frac{y}{T}\right) + \frac{\pi}{2}\right)$$

Следовательно, если a — реализация равномерно распределенной на [0; 1] случайной величины, то величина

$$T\ \text{tg}\ (\pi\alpha - \tfrac{\pi}{2}) \qquad (2.1)$$

имеет требуемое распределение.

В случае задач с размерностью D > 1 использовалось произведение D одномерных распределений Коши, т. е. вносилось возмущение по каждой из координат, вычисленное по формуле (2.1) для независимых α_i.

Для температуры в этом случае используется формула

$$T(k) = \frac{T_0}{k^{1/D}}$$

причем в случае $D = 1$ она вычислялась как T_0/k, в случае $D = 2$ — как $T_0/\sqrt{k}$ для увеличения скорости работы программ.

2.6.3.Сверхбыстрый отжиг

Для сверхбыстрого отжига использовалась также полученная методом обратных функций формула (2). Закон изменения температуры также имел отличия при $D = 1$ и $D = 2$. Если при $D = 2$ отличие, аналогично предыдущему случаю, сводилось к вычислению $k^{1/2}$ как $\sqrt{k}$, то при $D = 1$ возможна более существенная оптимизация, т. к.

$$\frac{T(k+1)}{T(k)} = \frac{e^{-c_i/(k+1)}}{e^{-c_i/k}} = e^{-c_i} = const$$

Значение этой константы вычислялось заранее, и при каждом уменьшении температуры производилось умножение на нее вместо вычисления экспоненты и деления, что примерно в 200 раз быстрее.

Для предотвращения деления на 0 в формулах температура была ограничена снизу значением 10^{-40000}.

2.6.4.Методы “тушения”

Кроме вышеописанных методов, были смоделированы четыре метода тушения:

• Больцмановский отжиг с уменьшением температуры как в методе Коши,

•Больцмановский отжиг с экспоненциальным уменьшением температуры ($T_{k+1} = cT_k,\ c = 0{,}99$),

•метод Коши с экспоненциальным уменьшением температуры($T_{k+1} = cT_k,\ c = 0{,}99$),

•сверхбыстрое тушение.

Сверхбыстрое тушение получается из метода сверхбыстрого отжига с использованием следующих формул:

$$T_i(k) = T_{(i;0)} \exp\left(-c_i k^{Q/D}\right), \qquad c_i > 0,$$
$$c_i = m_i \exp(-n_i/D)$$

где Q — так называемый множитель тушения. При тестировании множитель тушения брался равным 2. [1]

ГЛАВА 3. РАЗРАБОТКА И ТЕСТИРОВАНИЕ КОМПОНЕНТА

3.1 Алгоритм имитации отжига

Свойства структуры зависят от коэффициента охлаждения после того, как субстанция была нагрета до точки плавления. Если структура охлаждалась медленно, будут сформированы крупные кристаллы, что очень полезно для строения субстанции. Если субстанция охлаждается скачкообразно, образуется слабая структура.

Чтобы расплавить материал, требуется большое количество энергии. При понижении температуры уменьшается и количество энергии. Чтобы яснее представить процесс восстановления, рассмотрим следующий пример. «Взбалтывание» при высокой температуре сопровождается высокой молекулярной активностью в физической системе. Представьте себе, что вы взбалтываете емкость, в которой находится какая-то поверхность сложной формы. Внутри емкости также имеется шарик, который пытается найти точку равновесия. При высокой температуре шарик может свободно перемещаться по поверхности, а при низкой температуре «взбалтывание» становится менее интенсивным и передвижения шарика сокращаются. Задача заключается в том, чтобы найти точку минимального перемещения при сильном «взбалтывании». При снижении температуры уменьшается вероятность того, что шарик выйдет из точки равновесия. Именно в таком виде процесс поиска заимствуется из восстановления. [4]

Следует рассмотреть, как метафора охлаждения растаявшей субстанции используется для решения проблемы. Алгоритм отжига очень прост и может быть разделен на пять этапов.

Рис.3.1. Схема алгоритма отжига

Начальное решение

Для большинства проблем начальное решение будет случайным. На самом первом шаге оно помещается в текущее решение (Current solution). Другая возможность заключается в том, чтобы загрузить в качестве начального решения уже существующее, возможно, то самое, которое было найдено во время предыдущего поиска. Это предоставляет алгоритму базу, на основании которой выполняется поиск оптимального решения проблемы.

Оценка решения

Оценка решения состоит из декодировки текущего решения и выполнения нужного действия, позволяющего понять его целесообразность для решения данной проблемы. Обратите внимание, что закодированное решение может просто состоять из набора переменных. Они будут декодированы из существующего решения, а затем эффективность решения будет оценена на основании того, насколько успешно удалось решить данную задачу.

Случайный поиск решения

Поиск решения начинается с копирования текущего решения в рабочее решение (Working solution). Затем мы произвольно модифицируем рабочее решение. Как именно модифицируется рабочее решение, зависит от того, каким образом оно представляется (кодируется). Представьте себе кодировку задачи коммивояжера, в которой каждый элемент представляет собой город. Чтобы выполнить поиск по рабочему решению, мы берем два элемента и переставляем их. Это позволяет сохранить целостность решения, так как при этом не происходит повторения или пропуска города.

После выполнения поиска рабочего решения мы оцениваем решение, как было описано ранее. Поиск нового решения основан на методе Монте-Карло (то есть случайным образом).

Критерий допуска

На этом этапе алгоритма у нас имеется два решения. Первое - это наше оригинальное решение, которое называется текущим решением, а второе - найденное решение, которое именуется рабочим решением. С каждым решением связана определенная энергия, представляющая собой его эффективность (допустим, что чем ниже энергия, тем более эффективно решение).

Затем рабочее решение сравнивается с текущим решением. Если рабочее решение имеет меньшую энергию, чем текущее решение (то есть является более предпочтительным), то мы копируем рабочее решение в текущее решение и переходим к этапу снижения температуры.

Однако если рабочее решение хуже, чем текущее решение, мы определяем критерий допуска, чтобы выяснить, что следует сделать с текущим рабочим решением. Вероятность допуска основывается на уравнении 3.1 :

$$P(\delta E) = \exp(-\delta E/T) \qquad (3.1)$$

Значение этой формулы визуально показано на рис. 3.2. При высокой температуре (свыше 60 °C) плохие решения принимаются чаще, чем отбрасываются. Если энергия меньше, вероятность принятия решения выше. При снижении температуры вероятность принятия худшего решения также снижается. При этом более высокий уровень энергии также способствует уменьшению вероятности принятия худшего решения.

При высоких температурах симулированное восстановление позволяет принимать худшие решения для того, чтобы произвести более полный поиск решений. При снижении температуры диапазон поиска также уменьшается, пока не достигается равенство при температуре 0°.

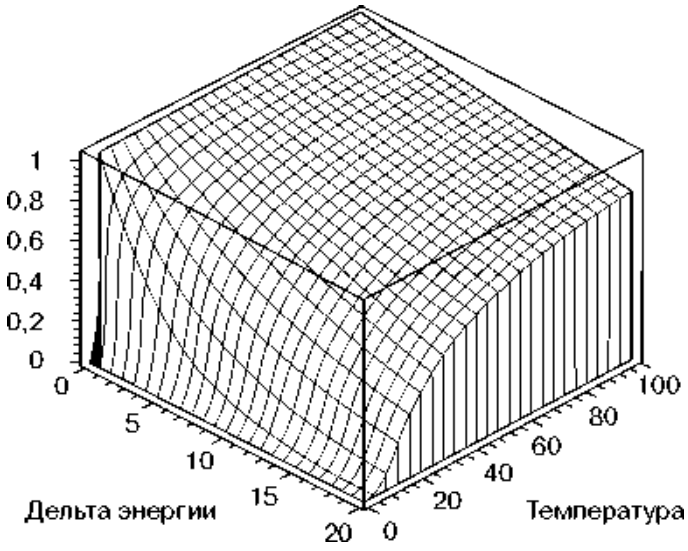

Рис.3.2. Визуализация вероятности допуска

Снижение температуры

После ряда итераций по алгоритму при данной температуре мы ненамного снижаем ее. Существует множество вариантов снижения температуры. В данном примере используется простая геометрическая функция (см. уравнение 3.2):

$$T_{i+1} = \alpha T_i \qquad (3.2)$$

Константа a меньше единицы. Возможны и другие стратегии снижения температуры, включая линейные и нелинейные функции.

Повтор

При одной температуре выполняется несколько итераций. После завершения итераций температура будет понижена. Процесс продолжится, пока температура не достигнет нуля.

Пример итерации

Чтобы проиллюстрировать алгоритм, проследим несколько итераций. Обратите внимание, что если рабочее решение имеет меньший уровень энергии (то есть является лучшим решением) по сравнению с текущим решением, то всегда используется именно оно. Критерий допуска вступает в силу только при условии, что рабочее решение хуже, чем текущее. [4]

Предположим, что температура окружающей среды равна 50°, а энергия текущего решения составляет 10. Мы копируем текущее решение в рабочее решение и выполняем поиск. После оценки энергии устанавливаем, что энергия нового рабочего решения равна 20. В этом случае энергия рабочего решения выше, чем энергия начального решения. Поэтому мы используем критерий допуска:

Энергия текущего решения равна 10.

Энергия рабочего решения равна 20.

Дельта энергии для этого примера (энергия рабочего решения минус энергия текущего решения) равна 10. Подставив это значение и температуру 50 в уравнение 3.1, получаем вероятность:

$$P = \exp(-10/50) = 0{,}818731.$$

Таким образом, на этом примере мы видим, что вероятность принятия худшего решения достаточно велика. Теперь рассмотрим пример с более низкой температурой. Предположим, что температура равна 2, а энергия имеет следующие показатели:

Энергия текущего решения равна 3.

Энергия рабочего решения равна 7.

Дельта энергии в этом примере равна 4. Подставив это значение и температуру в уравнение 3.1, получаем вероятность:

$$P = \exp(-4/2) = 0{,}135335.$$

Данный пример показывает, что вероятность выбора рабочего решения для последующих итераций очень невелика. Это базовая форма алгоритма.

3.2 Разработка библиотеки для реализации метода отжига

Для создания библиотеки классов в среде MS Visual Studio 2015 следует выбрать пункт указанный на рисунке 3.3.

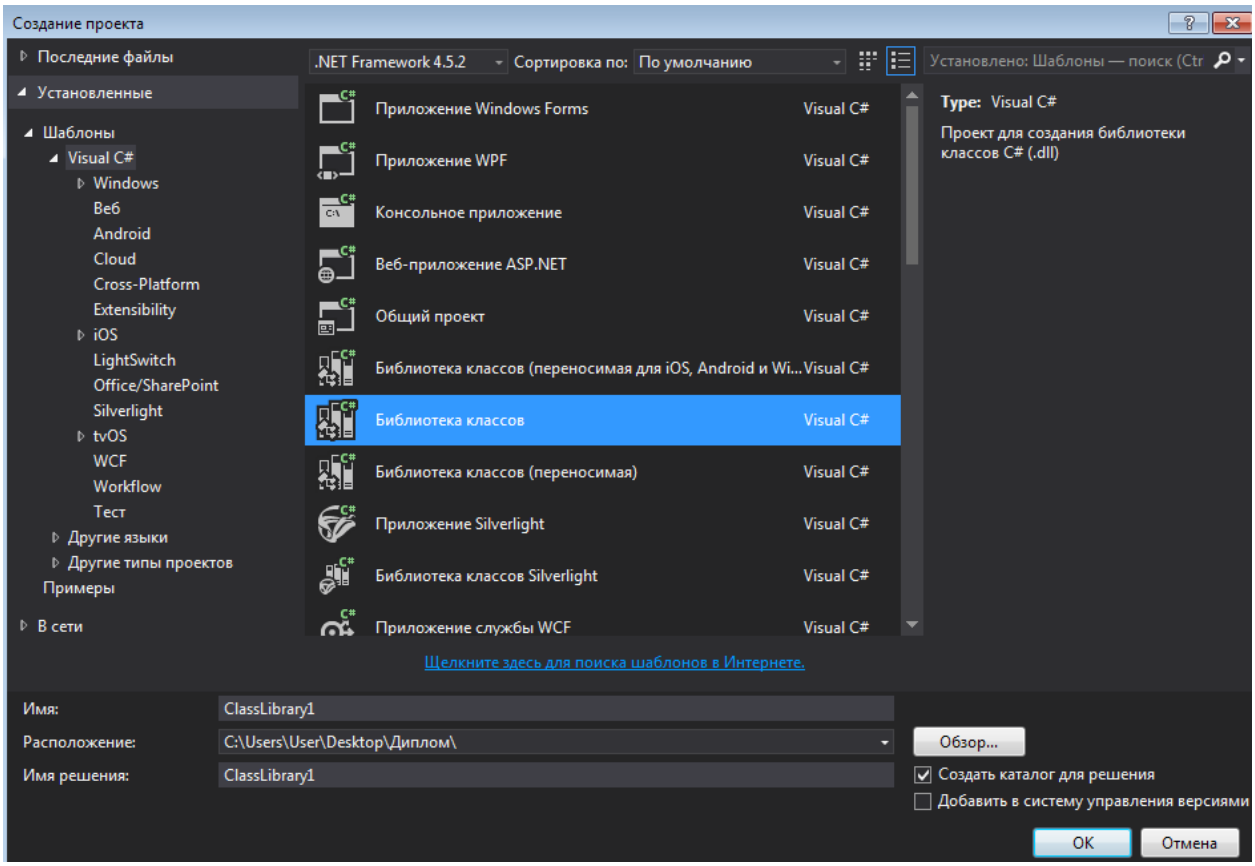

Рис 3.3. Создание библиотеки классов в среде VS 2015

Листинг 3.1. Алгоритм имитации отжига

```
int main()
{
int timer=0;
int step;
int solution=0;
int useNew, accepted;
 float temperature = initial_temperature;
initializeSolution( current );
computeEnergy( current );
best.energy = 100.0;
for (step = 0 ; step < steps_per_change ; step++) {
useNew = 0;
tweakSolution( working );
 computeEnergy( working );
if (working.energy <= current.energy) {
useNew = 1;
} else {
float test => getSRand.Next();
float delta = working.energy - current.energy;
float calc = exp(-delta/temperature);
if (calc > test) {
accepted++;
useNew = 1;
}
}
if (useNew) {
useNew = 0;
copySolution(current, working );
```

Продолжение - Листинг 3.1. Алгоритм отжига

```
if (current.energy < best.energy) {
copySolution( best, current );
solution = 1;
}
} else {
copySolution( working, current);
}
fclose(fp); if (solution) {
emitSolution( best );
} return 0;
}
```

Алгоритм отжига, представленный листингом 3.1, практически повторяет обобщенный алгоритм, показанный на рис.3.1. После двух вводных действий (открытия файла для вывода лога и инициализации генератора случайных чисел) мы инициализируем текущее решение в переменной current и оцениваем энергию решения при помощи функции computeEnergy. Текущее решение копируется в рабочее, и запускается алгоритм.

Внешний цикл алгоритма выполняется до тех пор, пока текущая температура не станет меньше конечной температуры или не сравняется с ней. Это позволяет избежать использования нулевой температуры в функции расчета вероятности.

Внутренний цикл алгоритма работает по методу Монте-Карло. Он выполняет ряд итераций при текущей температуре с целью полного изучения возможностей поиска при данной температуре.

Первый шаг - изменение рабочего решения с помощью функции tweakSolution. Затем рассчитывается энергия рабочего решения, которая сравнивается с текущим решением. Если энергия нового рабочего решения

меньше или равна энергии текущего решения, рабочее решение принимается по умолчанию. В противном случае выполняется уравнение 3.1 (оценка вероятности допуска). Таким образом определяется, будет ли выбрано худшее решение. Дельта энергии рассчитывается как разница между рабочей энергией и текущей. Это означает, что энергия рабочего решения больше, чем энергия текущего решения. В нашем случае просто генерируется случайное число в интервале от 0 до 1, которое затем сравнивается с результатом уравнения 3.1. Если условие допуска выполнено (результат уравнения 3.1 больше случайного значения), то рабочее решение принимается. Затем рабочее решение необходимо скопировать в текущее, так как переменная working, в которой на данный момент хранится рабочее решение, будет повторно изменена при следующей итерации внутреннего цикла.

Если рабочее решение не было принято, текущее решение копируется поверх рабочего. При следующей итерации старое рабочее решение удаляется, программа изменяет текущее решение и пробует снова.

После вывода статистической информации в лог-файл температуру необходимо снизить (выполнив требуемое количество итераций внутреннего цикла). Вспомните (уравнение 3.2), что график температуры представляет собой простую геометрическую функцию. Следует умножить текущую температуру на константу ALPHA и повторить внешний цикл.

В конце алгоритма выводится решение, хранящееся в переменной best, которое было найдено (если оно вообще было найдено, об этом сигнализирует переменная solution). Эта переменная устанавливается во внутреннем цикле после того, как было определено, что обнаружено решение, энергия которого меньше энергии текущего решения best.

3.3 Разработка приложения для тестирования компонента

Пример задачи

Для демонстрации этого алгоритма используется широко известная задача, решить которую пытались с помощью множества алгоритмов поиска. Задача N шахматных ферзей (или NQP) - это задача размещения N ферзей на шахматной доске размером NxN таким образом, чтобы ни один ферзь не угрожал другому (рис. 3.4).

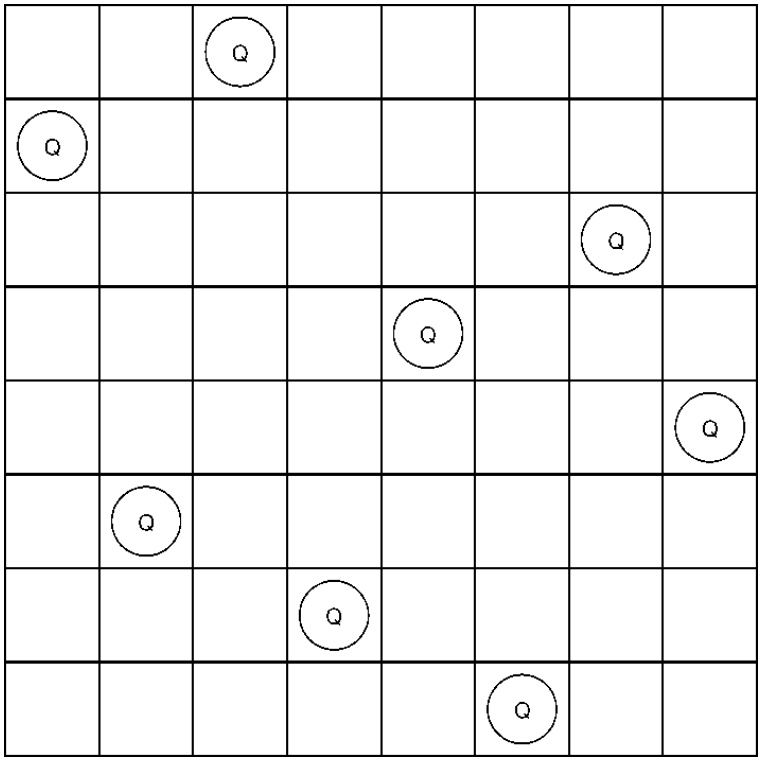

Рис.3.4. Кодировка решения задачи 8 ферзей

Задача 8 ферзей впервые была решена в 1850 г. Карлом Фридрихом Гаубом (Carl Friedrich Gaub). Алгоритм поиска (как видно из даты решения) представлял собой метод проб и ошибок. Затем задача ферзей решалась с помощью метода поиска в глубину (1987), метода «разделяй и властвуй» (1989), генетических алгоритмов (1992) и многими другими способами. В 1990 г. Рок Сосик (Rok Sosic) и Цзюн Гу (Jun Gu) решили проблему для 3000000 ферзей с использованием метода локального поиска и минимизации конфликтов.

Представление решения

Представление решения (кодировка) задачи о N ферзях стандартна: для сужения области поиска используется конечное решение. Обратите внимание

(рис. 3.4), что в каждой строке и каждом столбце может располагаться только один ферзь. Это ограничение намного упрощает создание кодировки, которая управляется алгоритмом отжига.

5	7	1	4	2	8	6	3
1	2	3	4	5	6	7	8

Рис. 3.5. Строчные индексы доски

Так как каждый столбец содержит только одного ферзя, для отображения решения будет использоваться массив из N элементов (рис. 3.5). В элементах этого массива хранятся строчные индексы положения ферзя. Например, на рис. 3.5 столбец 1 содержит значение 5, соответствующее строке, в которую будет помещен ферзь.

Создать произвольное решение очень просто. Сначала нужно инициализировать решение, позволив каждому ферзю занять строку, соответствующую столбцу. Затем необходимо пройти по каждому столбцу и выбрать произвольное число от 1 до N для каждого столбца. Затем два элемента перемещаются (текущий столбец и произвольно выбранный столбец). Когда алгоритм достигает конца, автоматически формируется решение.

Наконец, получив кодировку, мы видим, что на доске нет конфликтов по горизонтали или диагонали. При оценке решения следует учитывать только конфликты по диагонали.

Энергия

Энергия решения определяется как количество конфликтов, которые возникают в кодировке. Задача заключается в том, чтобы найти кодировку, при которой энергия равна нулю (то есть на доске нет конфликтов).

Температура

Для данной проблемы мы начнем поиск решения с температуры 30° и постепенно будем снижать ее до нуля, используя геометрическую формулу (уравнение 3.2). При этом значение α будет равно 0,98. Как будет видно далее, график температуры показывает сначала быстрое снижение, а потом медленное схождение к конечной температуре - нулю.

При каждом изменении температуры мы выполним 100 итераций. Это позволит алгоритму осуществить несколько операций поиска на каждом уровне

Исходный код

Рассмотрим исходный код, который реализует алгоритм отжига для решения задачи N ферзей.

Сначала мы взглянем на константы и типы данных, которые использует алгоритм (см. листинг 3.2).

Листинг 3.2. Типы данных и константы

```
int qeenNumb;
int[] solutionType = new int[qeenNumb];
float energy;
float initial_temperature;
float final_temperature = 0.5;
float alpha = 0.98;
int steps_per_change = 100;
```

Массив solutionType - это наша кодировка задачи о ферзях. Символьная константа *qeenNumb* определяет размер доски (в данном случае решается задача для 20 ферзей). Допускается изменять значение константы *qeenNumb* (до 40 или более), однако при использовании большего значения могут понадобиться изменения в графике охлаждения.

Решение хранится в структуре memberType, которая также включает энергию, рассчитываемую для решения.

Остальная часть листинга 3.2 определяет график охлаждения. Константы initial_temperature и final_temperature задают границы графика, а константа ALPHA используется для определения геометрического охлаждения. Константа steps_per_change устанавливает количество итераций, которые будут выполнены после каждого изменения температуры.

Далее мы рассмотрим вспомогательные функции алгоритма. В листинге 3.3 содержатся функции инициализации кодировки и поиска нового решения. Они используются для создания начального решения и его произвольного изменения.

Листинг 3.3. Функции поиска

```
void tweakSolution(member )
{
int temp, x, y;
x => qeenNumb.Next(); do {
y = => qeenNumb.Next();
} while (x == y);
temp = member->solution[x];
solution[x] = member;
solution[y] = temp;
}
void initializeSolution(member )
{
int i;
for (i = 0 ; i < qeenNumb; i++)        {
solution[i] = i;
}
```

Продолжение - Листинг 3.3. Функции поиска

```
for (i = 0 ; i < qeenNumb; i++)        {
tweakSolution( member );
}}
```

Функция initializeSolution создает решение, при котором все ферзи помещаются на доску. Для каждого ферзя задаются идентичные индексы строки и столбца. Это обозначает отсутствие конфликтов по горизонтали и вертикали. Затем решение изменяется при помощи функции tweakSolution. Позднее функция tweakSolution используется в алгоритме, чтобы изменить рабочее решение, выведенное из текущего решения.

Оценка решения выполняется с помощью функции computeEnergy. Функция идентифицирует все конфликты, которые существуют для текущего решения (листинг 3.4).

Листинг 3.4. Оценка решения

```
void computeEnergy(member )
{
int I;
int j;
int x;
int y;
int tempx,tempy;
 float[,] board = new float[qeenNumb, qeenNumb];
int conflicts;
const int dx[4] = {-1, 1, -1, 1 } ;
const int dy[4] = {-1, 1, 1, -1 } ;
for (i = 0 ; i < qeenNumb; i++)        {
board[i, solution[i]] = 1;
conflicts = 0;
for (i = 0 ; i < qeenNumb; i++)        {
```

Продолжение - Листинг 3.4. Оценка полученного решения

```
x = i;
y = solution[i];
for (j = 0 ; j < 4 ; j++)   {
```

```
tempx = x ;
tempy = y;
while(t==1) {
tempx += dx[j];
tempy += dy[j];
if ((tempx < 0)       || (qeenNumb) ||
(tempy < 0) || (qeenNumb)) break;
if (board[tempx, tempy] == 1) conflicts++;}}}
energy = (float)conflicts;
}
```

Чтобы продемонстрировать результат, построим шахматную доску. Это не является обязательным, но упрощает зрительное восприятие решения проблемы. Обратите внимание на диапазоны dx и dy. Эти диапазоны используются для расчета следующего положения на доске для каждого найденного ферзя. В первом случае dx = -1, а dy = -1, что соответствует перемещению на северозапад. Конечный результат, dx = 1, dy = -1, отвечает перемещению на северовосток.

Мы выбираем по очереди каждого ферзя на доске (ферзь определяется значениями x и y), а затем перемещаемся по все четырем диагоналям в поиске конфликтов, то есть другого ферзя. Если ферзь найден, значение переменной конфликта увеличивается. После завершения поиска конфликты загружаются в структуру с решением в качестве значения энергии.

Следующая функция используется для копирования одного решения в другое. Вспомните (листинг 3.2), что решение кодируется и сохраняется в структуре memberType. Функция copySolution копирует содержимое одной структуры memberType в другую (листинг 3.5).

Листинг 3.5. Копирование полученного решения в другое

```
void copySolution()
{
```

int i;

for (i = 0 ; i < qeenNumb ; i++){

dest.solution[i] = src.solution[i];

}dest.energy = src.energy;}

Последняя вспомогательная функция, которую мы рассмотрим, - это функция emitSolution. Она просто распечатывает представление доски из закодированного решения и выдает его. Печатаемое решение передается как аргумент функции (листинг 3.6). Отображение шахматной доски можно увидеть на рисунке 3.6.

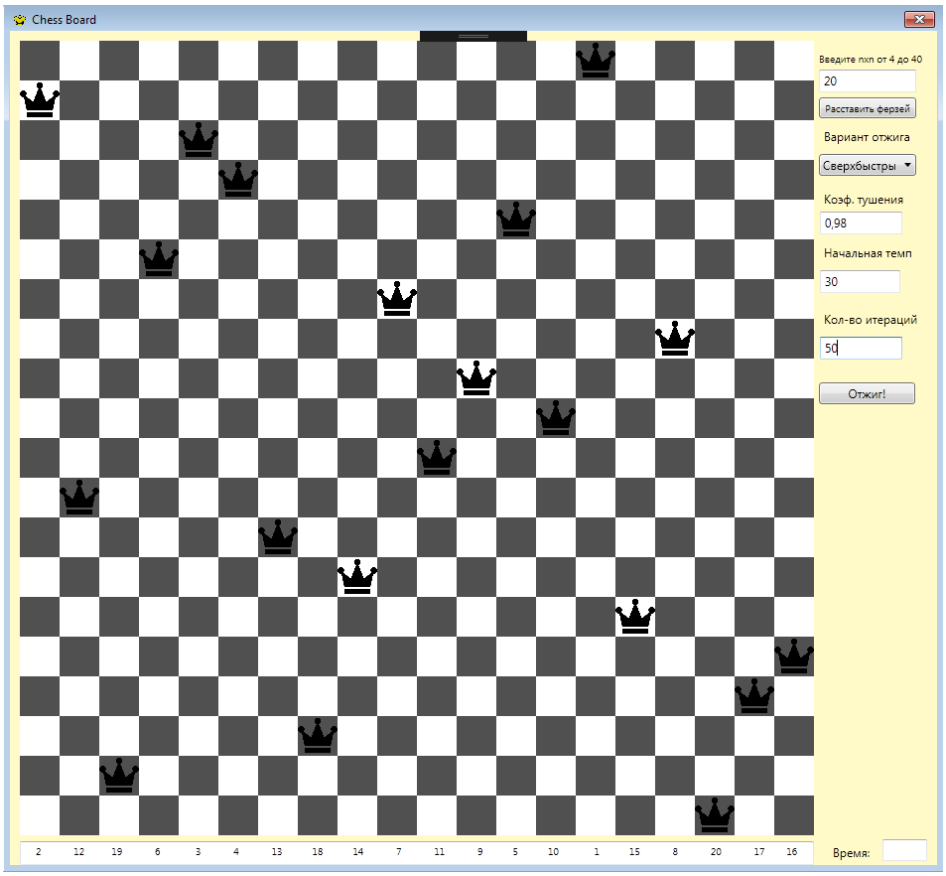

Рис. 3.6. Случайное расположение ферзей на шахматной доске

Листинг 3.6. Отображение решения в виде шахматной доски

Board.RowDefinitions.Clear();

Board.ColumnDefinitions.Clear();

MirrArrBox.Clear(); //очистка бокса для зеркала матрицы

//от 4 до 40. Переменная quinNumb хранит колличество ферзей на поле

```
qeenNumb = Convert.ToInt32(textBox.Text);
if (qeenNumb > 3 && qeenNumb < 41)
{
//////////////////////////////////////////////////////////////
//рандом для расположения едениц в матрице
Random randCoor = new Random();
//матрица с размероностью доски
arrCoorBoard = new int[qeenNumb, qeenNumb];
//образ матрицы
//mirArrCoorBoard = new int[qeenNumb];
//int[] mirArrCoorBoard = new int[]
{2,4,6,8,10,12,14,16,18,20,3,1,7,5,11,9,15,13,19,17};
//заполняем зеркало массива
 for (int i = 1; i < qeenNumb; i++)
 {
mirArrCoorBoard[i] = i;
}
//перемешиваем зеркало массива хз как я сам не понял
mirArrCoorBoard = mirArrCoorBoard.OrderBy(x =>
randCoor.Next()).ToArray();
for (int i = 0; i < qeenNumb; i++)
{
arrCoorBoard[i, mirArrCoorBoard[i]] = 1;
MirrArrBox.Text += (Convert.ToString(mirArrCoorBoard[i] + 1) + ' ');
for (int k = 0; k < qeenNumb; k++)
```

Продолжение - Листинг 3.6. Отображение решения в виде шахматной доски

```
{
if (k != mirArrCoorBoard[i])//сохранение ячейки с 1
{
```

```
arrCoorBoard[i, k] = 0;
}}}
//сумма координат ячеек
int sumCoor;
//защита от дурака
// if (qeenNumb > 3 && qeenNumb < 41)
//{
//цикл для заполнения поля сеткой
for (int i = 0; i < qeenNumb; i++)
{
//добавление строки
Board.RowDefinitions.Add(new RowDefinition());
//добавление столбца
Board.ColumnDefinitions.Add(new ColumnDefinition());
for (int j = 0; j < qeenNumb; j++)
{
sumCoor = i + j;
Rectangle Cell = new Rectangle();
//цвет рамки ячейки
//Cell.Stroke = new SolidColorBrush(Color.FromRgb(0, 0, 0));
Uri uri = new Uri("pack://application:,,/bin/Debug/qeen4.png");
BitmapImage bitmap = new BitmapImage(uri);
Image Qeen = new Image();
Qeen.Source = bitmap;
if (sumCoor % 2 == 0)
```

Продолжение - Листинг 3.6. Отображение решения в виде шахматной доски

```
{
Cell.Fill = new SolidColorBrush(Color.FromRgb(80, 80, 80));
}
```

```
else
{
Cell.Fill = new SolidColorBrush(Color.FromRgb(255, 255, 255));
}
Grid.SetColumn(Cell, i);
Grid.SetRow(Cell, j);
Board.Children.Add(Cell);
if (arrCoorBoard[i, j] == 1)
{
Grid.SetColumn(Qeen, i);
Grid.SetRow(Qeen, j);
Board.Children.Add(Qeen);}}}}
else
{
MessageBox.Show("Вы ввели знаечение не из диапазона, либо не ввели ничего.");}}
```

Пример выполнения

Рассмотрим результат запуска алгоритма. В этом примере мы начнем с температуры 100, хотя для решения проблемы достаточно температуры 30. Такая высокая температура задана для иллюстрации работы алгоритма.

На рисунке 3.7 изображены этапа выполнения алгоритма отжига на примере задачи о N ферзях.

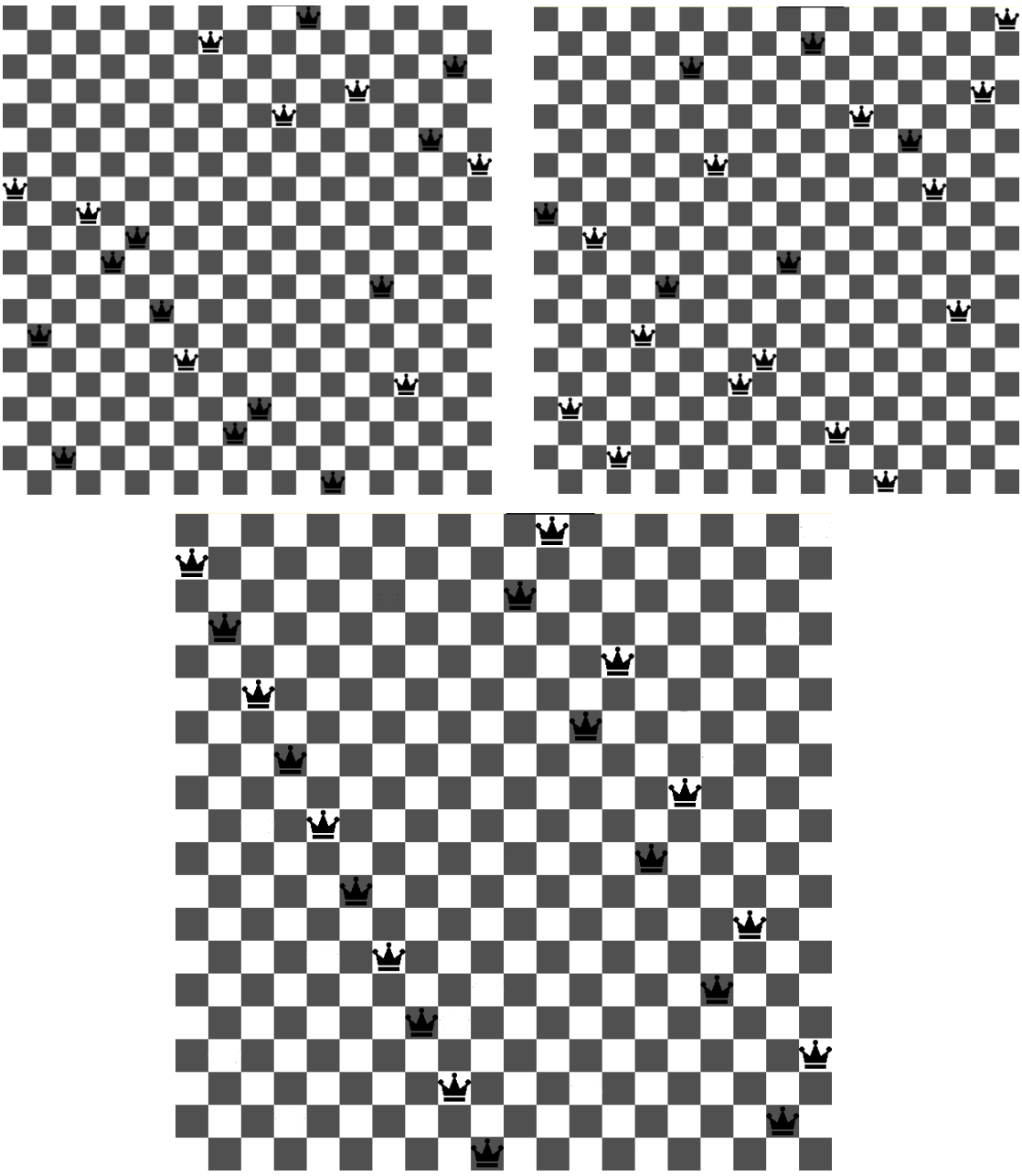

Рис.3.7 Этапы выполнения алгоритма

Энергия во времени

Элементом, значение которого резко уменьшается от 100 до 0, является температура. График охлаждения использует уравнение 3.2. Элемент, значение которого уменьшается не так резко, - это количество принимаемых худших решений (основанное на вероятностном уравнении допуска 3.1). Так как вероятность допуска представляет собой функцию температуры, легко заметить их взаимосвязь. Наконец, третий график иллюстрирует энергию лучшего решения. Как показывает график, идеальное решение находится

только в самом конце поиска. Это справедливо для всех запусков алгоритма (причина - уравнение критерия допуска), что демонстрируется резким спадом кривой «допустимых» худших решений в конце графика (рис.3.8).

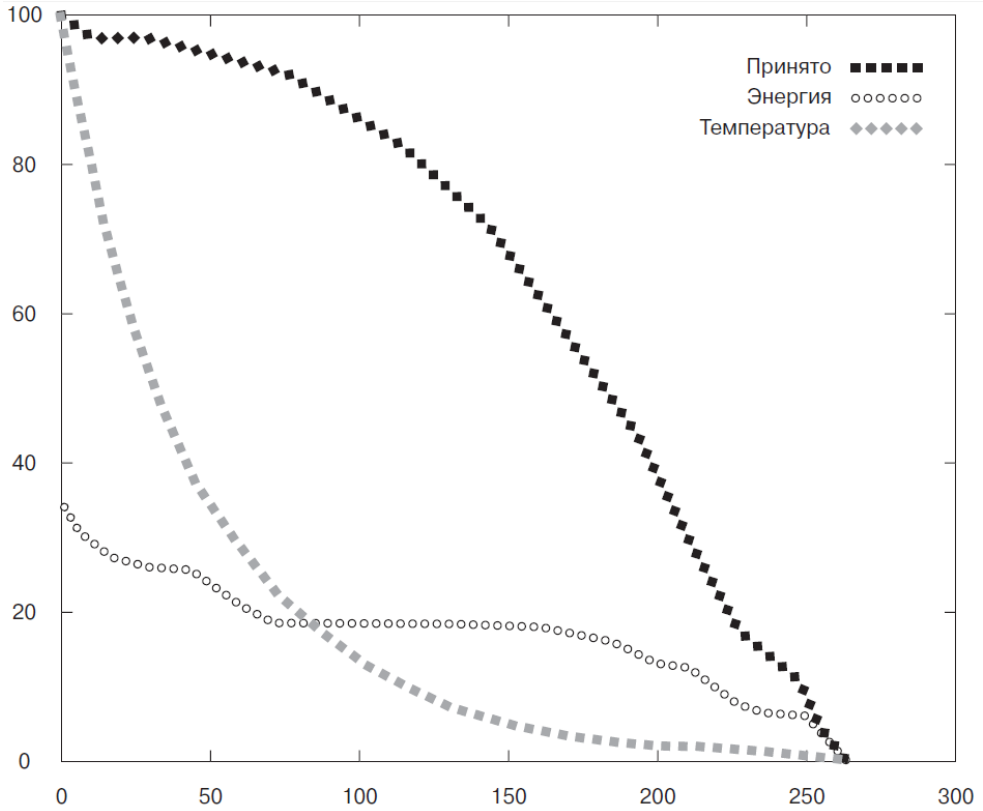

Рис.3.8. Энергия во времени

Оптимизация алгоритма

Вы можете изменять параметры алгоритма в зависимости от сложности проблемы, которую нужно решить. В этом разделе описаны параметры, которые допускается переопределять, а также результаты возможных изменений.

Начальная температура

Начальная температура должна быть достаточно высокой, чтобы сделать возможным выбор из других областей диапазона решений. По утверждению Грэхема Кендалла (Graham Kendall), если известно максимальное расстояние между соседними решениями, то легко рассчитать начальную температуру.

Начальную температуру также можно изменять динамически. Если задать статистику по коэффициенту допуска худших решений и нахождению новых лучших решений, можно повышать температуру до тех пор, пока не будет достигнуто нужное количество допусков (открытий новых решений). Этот процесс аналогичен нагреву субстанции до перехода ее в жидкую форму, после чего уже нет смысла повышать температуру.

Конечная температура

Хотя ноль является удобной конечной температурой, геометрическая функция, которая используется в примере, показывает, что алгоритм будет работать намного дольше, чем это действительно необходимо. Поэтому конечная температура в листинге 3.1 задана как 0,5°. Это значение может изменяться в зависимости от того, какая функция изменения температуры используется.

Функция изменения температуры

Используемую функцию изменения температуры можно модифицировать в зависимости от решаемой задачи. На рис. 3.8 показано изменение температуры во времени при применении геометрической функции. Снижение температуры допускается определять и с помощью многих других функций. Результатом использования этих функций может быть постепенное снижение температуры в первой половине графика или медленное снижение, за которыми следует резкий спад.

Количество итераций при одном значении температуры

При высоких температурах алгоритм отжига выполняет поиск оптимального решения во всем диапазоне решений. При снижении температуры движение уменьшается, и алгоритм ищет локальный оптимум, чтобы улучшить решение. Поэтому количество итераций, заданное для каждой температуры, имеет большое значение. При решении данной задачи (листинг 3.1) указано 100 итераций. Чтобы правильно определить количество итераций, которое оптимально подходит для решения проблемы, необходимо поэкспериментировать.

3.4 Сравнение различных вариаций метода отжига

В качестве сравнения различных вариаций метода отжига была выбрана известная задача о N ферзях (вариант задачи о 8 ферзях). Данная задача была выбрана для тестирования библиотеки для более простого сравнения результатов с другими методами, так как ее решали различными методами оптимизации другие авторы. Благодаря такому выбору появилась возможность сравнить метод имитации отжига с другими методами не реализуя их.

Для сверки вариантов алгоритма отжига были заданы следующие параметры. Размерность доски и количество ферзей было выбрано 20. Хотя в разработанном для тестирования примере имеется возможность выбора размерности от 4 до 40, параметр 20 является наиболее наглядным для сопоставления методов. Коэффициент тушения равен 0.98 наиболее удачный параметр, как показали тесты. При таком коэффициенте отжиг происходит наиболее быстро и одновременно точно. Начальная температура 30, хотя можно было выбрать параметр 50, но от этого не существенно страдало время отжига. Данный параметр следует увеличивать при большей размерности, например, для размерности 40 такая температура будет слишком мала и от данного выбора пострадает точность. Количество итераций равно 50. Здесь также, как и с температурой, для такой размерности этого параметра вполне достаточно, но увеличивая количество ферзей, нужно увеличивать и данный параметр.

Для каждого варианта было проведено по 100 запусков. На рисунке 3.9 можно увидеть отношение запусков методов к невязке. Исходя из полученных результатов можно сделать вывод, что самым универсальным и быстрым вариантом является сверхбыстрый отжиг, так как он заметно опережает все соседние методики. И целесообразно использовать для решения различных оптимизационных задач именно сверхбыстрый отжиг.

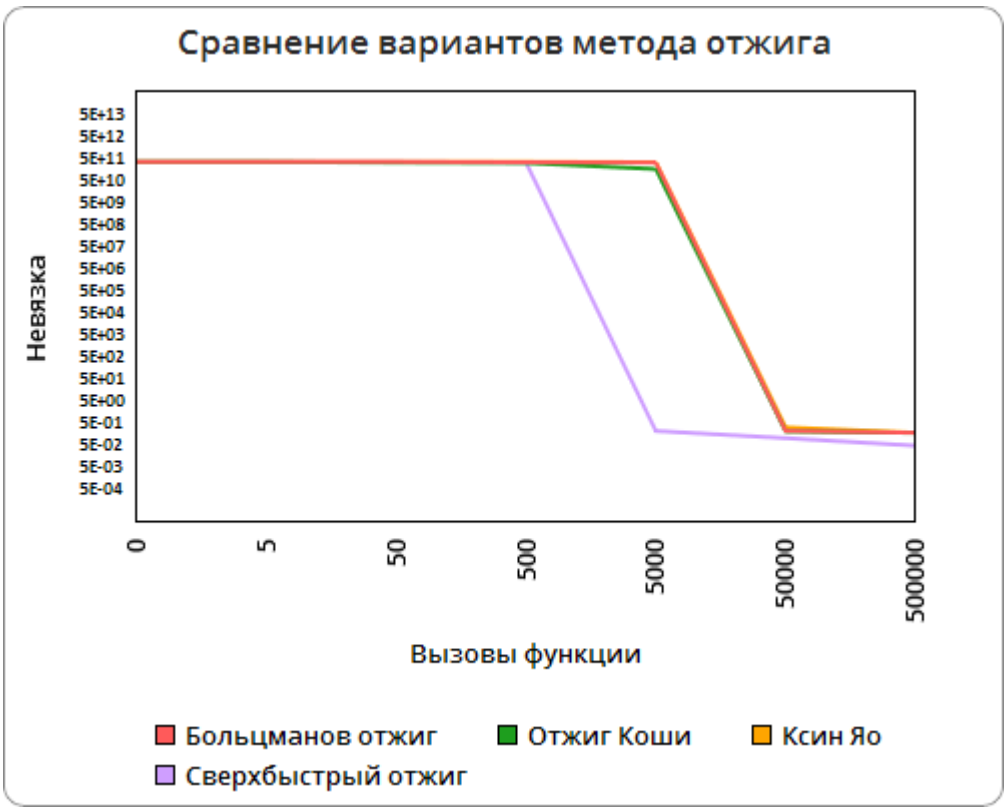

Рис. 3.9. Сравнение вариантов метода отжига

Для более наглядного отображения полученных результатов составлена таблица невязки для метода сверхбыстрого отжига в зависимости от числа вызова функции. Данный отображены в таблице 3.1.

Таблица 3.1

Таблица невязки

Число итераций	Невязка
10000	0.763129571893273058
100000	0.043839954318724353
1000000	0.004662248394661734
10000000	0.000547701824698814
100000000	0.000056790943476683
1000000000	0.000007776614600166

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были выполнены следующие задачи:

- Исследование метода имитации отжига;
- Разработка и внедрения компонента в платформу .NET;
- Создание среды тестирования компонента (задача о N ферзях);
- Сравнительный анализ различных вариаций метода.

Стоит отметить, что была достигнута главная цель данной работы, а именно: разработка компонента реализации метода имитации отжига, для платформы .NET.

Основным превосходством метода имитации отжига перед различными методами оптимизации является, то что при верном подборе входных параметров этот метод является крайне быстрым, также стоит отметить его «супер способность» обходить локальные минимумы рабочей функции, и без проблем продолжать поиск глобального минимума даже при малой энергии.

Все это достигается за счет умного подбора начальных параметров. Самым важным параметром можно назвать T – температуру. Чем она больше, тем больше вероятность проскочить локальный минимум. Из вышесказанного можно сделать вывод, что в правильных руках метод имитации отжига является сильным оружием для решения оптимизационных задач, но если его использовать бездумно, то данный алгоритм никак себя не покажет.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Лопатин А. С. Метод отжига. Санкт-Петербургский государственный университет, 2010. – 17с.

2) Ingber L., Rosen B. Genetic Algorithms and Very Fast Simulated Reannealing: A Comparison // Mathematical and Computer Modelling. 16(11). 1992. P. 87-100.

3) Тим Джонс. Программирование искусственного интеллекта в приложениях - М. М.: ДМК Пресс, 2011. - 312 с.

4) Джош: М. Т, Программирование искусственного интеллекта в приложениях. — М.: ДМК-пресс, 2014. – 720с.

5) Хабрахабр – интересные публикации / [Электронный ресурс] / Режим доступа: https://habrahabr.ru/post/209610/

6) Хабрахабр – интересные публикации / [Электронный ресурс] / Режим доступа: https://habrahabr.ru/post/112189/

7) Хабрахабр – интересные публикации / [Электронный ресурс] / Режим доступа: https://habrahabr.ru/post/308960/

8) А. П. Карпенко. Современные алгоритмы поисковой оптимизации. Алгоритмы, вдохновленные природой: учебное пособие. — Москва: Издательство МГТУ им. Н. Э. Баумана, 2014. — 446с.

9) Просиз Джеф. Программирование для Microsoft .NET /Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2013. — 704 стр.

10) Аттетков А. В., Галкин С. В., Зарубин В. С. Методы оптимизации: Учебник для ВУЗов. - М.: Из-во МГТУ им. Н. Э. Баумана, 2010, 439 с.

11) Воеводин В. В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ- Петербург, 2002, 608 с.
12) Курейчик В. В., Курейчик В. М., Родзин С. И. Теория эволюционных вычислений. М.: ФИЗМАТЛИТ, 2012, 260 с.
13) Ларичев О. И. Теория и методы принятия решений: Учебник для ВУЗов. М.: Университетская книга, Логос, 2006, 392 с.
14) Рутковская Д. Нейронные сети, генетические алгоритмы и нечеткие системы: Пер. с польск. / Рутковская Д., Пилиньский М., Рутковский Л. М.: Горячая линия-Телеком, 2004, 452 с.
15) Forman M. C. Compression of Integral Three Dimensional Television Pictures / Ph. D. Thesis at De Montfort University Leicester. 2000. United Kingdom.
16) Jeong C., Kim M. Fast Parallel Simulated Annealing for Traveling Salesman Problem on SIMD Machines with Linear Interconnections // Parallel Computing. 17. 1991. P. 221-228.
17) Yao X. Call Routing by Simulated Annealing // International Journal of Electronics. Oct. 1995.
18) Джон Скит. C# для профессионалов: тонкости программирования, 3-е издание, новый перевод = C# in Depth, 3rd ed.. — М.: «Вильямс», 2014. — 608 с.
19) Кристиан Нейгел и др. C# 5.0 и платформа .NET 4.5 для профессионалов = Professional C# 5.0 and .NET 4.5. — М.: «Диалектика», 2013. — 1440 с.
20) А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд. Язык программирования C#. Классика Computers Science. 4-е издание = C# Programming Language (Covering C# 4.0), 4th Ed. — СПб.: «Питер», 2012. — 784 с.
21) Э. Стиллмен, Дж. Грин. Изучаем C#. 2-е издание = Head First C#, 2ed. — СПб.: «Питер», 2012. — 704 с.
22) Гик Е. Я. Шахматы и математика. — М.: Наука, 1983. — 176 с.

Выпускная квалификационная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

«___» _____ _____ г.

_____ _____

(подпись) *(Ф.И.О.)*