Оглавление

1.Историческая справка

Формулировка задачи: «найти способ расставить на шахматной доске восемь ферзей таким образом, чтобы ни один из них не попадал под удар любого другого».

Данная задача известна любому шахматному любителю и её самостоятельное решение считается первым шагом к большим успехам в этой игре. Такое суждение обосновано тем, что для получения верного ответа требуется задействовать память, внимание, логическое и даже творческое мышление. Именно этим она в своё время привлекла внимание самого Карла Гаусса, который смог найти 72 решения из 92 возможных. При этом три четверти решений он получил с помощью поворота доски на 90, 180 и 270 градусов, что демонстрирует необходимость поиска нетривиальных решений на основе уже существующих.

Сложность задачи подтверждает тот факт, что лучшим учёным и шахматистам XIX века потребовалось 26 лет с момента формулировки задачи для нахождения всех решений и доказательства того, что те самые 92 решения являются максимально возможным числом. Однако при увеличении размеров доски n количество решений, естественно, увеличивается. В квадрате со стороной 18 их количество свыше 600 млн. Когда же n достигает 1000, то даже самые современные компьютерные программы перестают справляться с поставленной задачей.

В нынешнем мире интерес к задаче первую очередь исходит именно от программистов. Алгоритм быстрого поиска всех возможных решений этой задачи можно будет адаптировать для и других, более масштабных, проблем, в том числе из области криптографии, где подобный алгоритм в руках недобросовестных людей может быть использован для вредоносных целей. Многие крупные IT- компании предлагают огромные награды за предоставленный алгоритм, одинаково эффективный для любого n, или же доказательство его отсутствия. Также считается, что решение задачи о восьми ферзях эквивалентно решению одной из задач тысячелетия – а именно вопросе о тождестве классов сложности P и NP.

2.Теоритическая справка

Самые примитивные алгоритмы решают эту задачу с помощью банального перебора. Чуть более эффективные - сокращают перебор с помощью учета уже поставленных ферзей на каждой диагонали, вертикали и горизонтали. Самые же «быстрые» алгоритмы, помимо этого, используют различные вариации «поиска с возвратом» (бэктрекинг). Суть его в том, что решение задачи сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше.

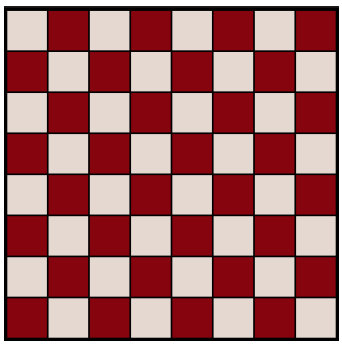
Тем не менее, даже при использовании самой простой реализации этого метода подсчет будет вестись довольно медленно (более 800 секунд) уже при N = 16, что не тянет на удовлетворительный результат.

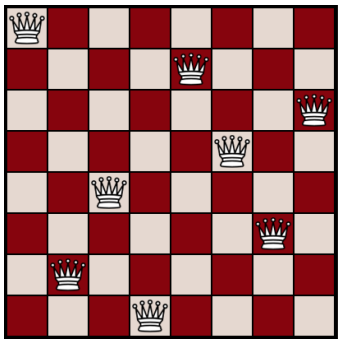
Три типа задачи "о ферзях"

Есть три наиболее популярных постановки задачи о ферзях

Расстановка N ферзей

Задача формулируется очень прямолинейно.
Дано: пустая доска NxN, например 8x8





(в принципе понятно, что достаточно просто указать N, но так наглядней)
Найти: расстановку максимально возможного числа ферзей
Т.е. на вход число — размер доски, а на выход позиции ферзей (одного решения).

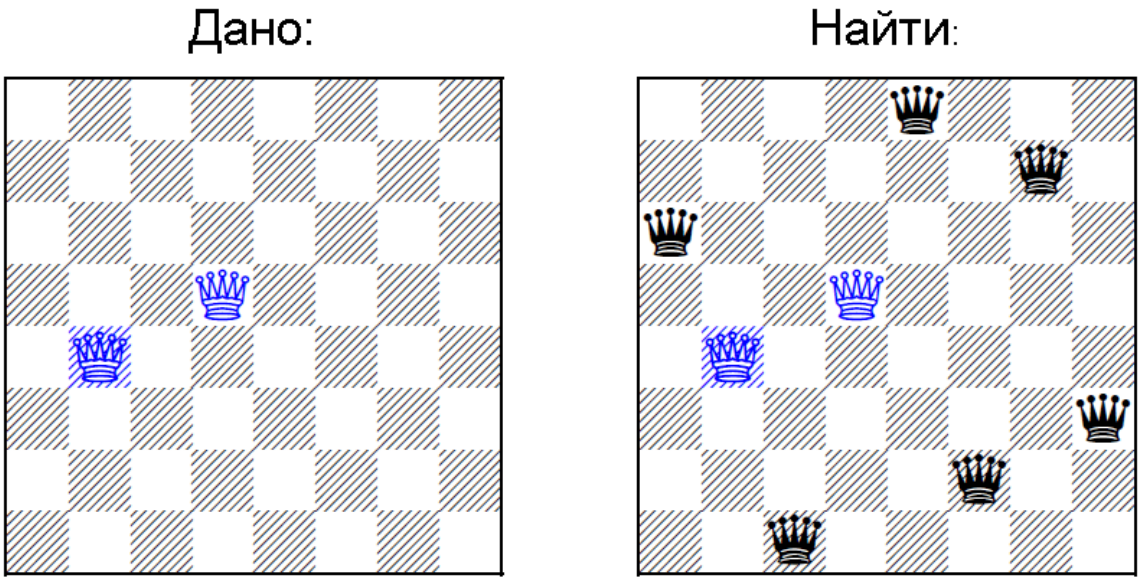
Подсчет числа решений

Задача ставится тоже достаточно просто:
Дано: размер пустой доски N
Найти: H число возможных расстановок N ферзей на доске
Например, размер доски N = 1, тогда число возможных расстановок H = 1.
N = 8 => H = 92.
Решение данной задачи было реализовано для сравнения библиотек.

n	1	2	3	4	5	6	7	8	9	10	11	12
Число решений	1	-	-	2	10	4	40	92	352	724	2680	14200

Дополнение до N ферзей

Вот тут формулировка чуть-чуть коварней:
Дано: размер доски N и M позиций уже установленных ферзей
Найти: позиции оставшихся N — M ферзей



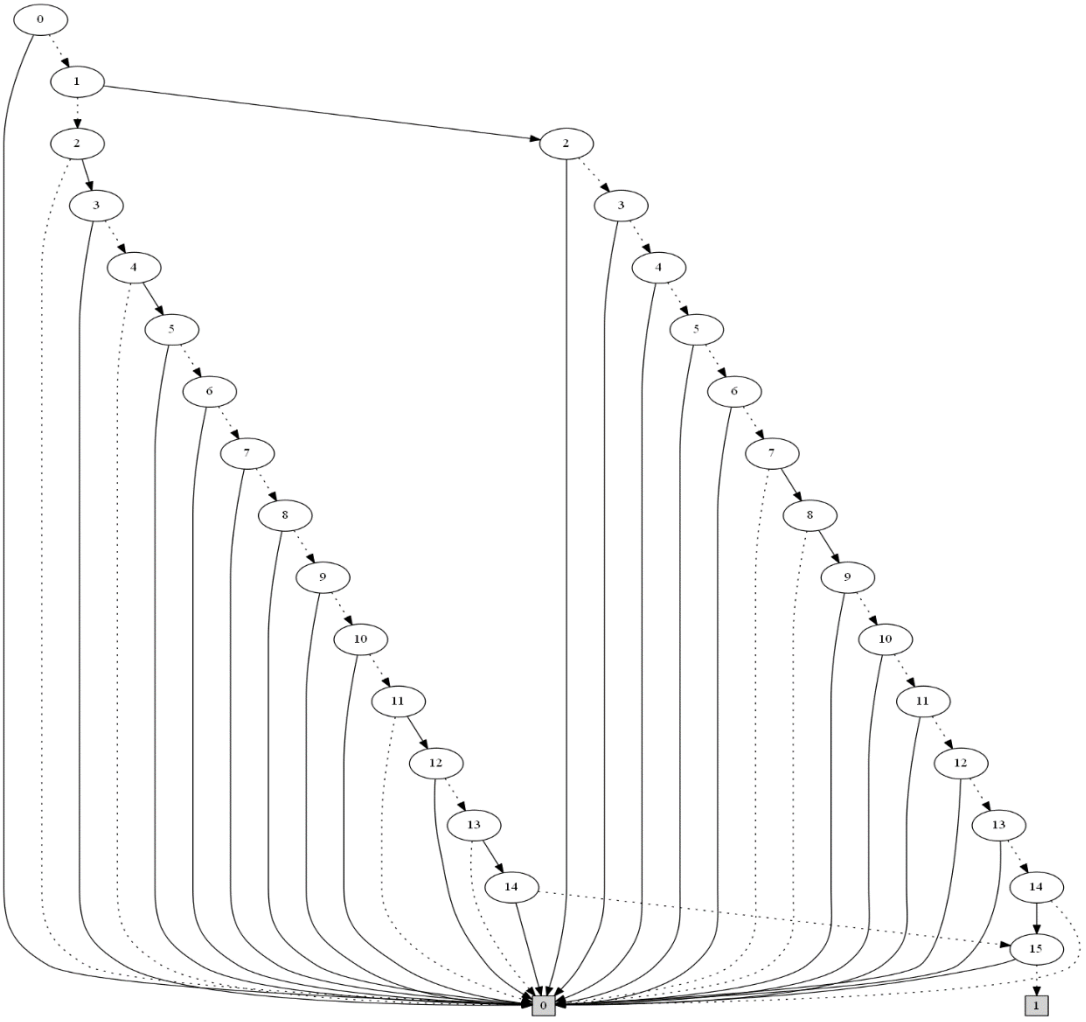
Реализованная программа решает именно последнюю задачу.

3.Формализация.

Задача была формализована по таким правилам:

1. В каждой строке только 1 ферзь.
2. В каждом столбце только 1 ферзь.
3. На каждой диагонали может быть не более 1 ферзя.
4. X[i][j] – в позиции (i,j) стоит ферзь.
5. Клетки доски пронумерованы от 0 до (n*n-1), начиная с левого верхнего угла. Таким образом на доске клетке с позицией X[i][j] соответствует переменная (i*n+j).

Пример BDD для доски 4x4:



BDD для досок других размеров слишком большие и не вместились в данный отчет. В папке с проектом лежат изображения BDD для досок с другими размерами.

4.Сравнение библиотек.

В процессе написания проекта были протестировано несколько библиотек, реализующих bdd. Сравнивалось время инициализации и построения bdd.

BuDDy. Пакет был сделан как часть ph.d. проекта по моделированию конечных автоматов. Пакет эволюционировал от простого введения в BDD к полномасштабному пакету BDD со всеми стандартными операциями BDD, переупорядочением и богатой документацией. Не поддерживает многопоточность.

CUDD. (Decision Diagram package from University of Colorado). Пакет CUDD управляет двоичными схемами принятия решений (BDD), алгоритмами алгебраических решений (ADD) и нулевыми подавленными схемами принятия решений (ZDD). Данная реализация не поддерживает многопоточность.

CAL. (dedicated to our alma-mater) BDD package provides routines for manipulating BDDs based on breadth-first manipulation algorithm. CAL не обладает большой функциональностью, которую имеет BuDDy. Кроме того, он медленнее, чем BuDDy. Также не поддерживает многопоточность.

Java impl. Это 100% Java реализация BDD. Он основан на исходном коде C для BuDDy. Таким образом, реализация очень уродливая, но она работает. Как и BuDDy, он использует схему подсчета ссылок для сбора мусора.

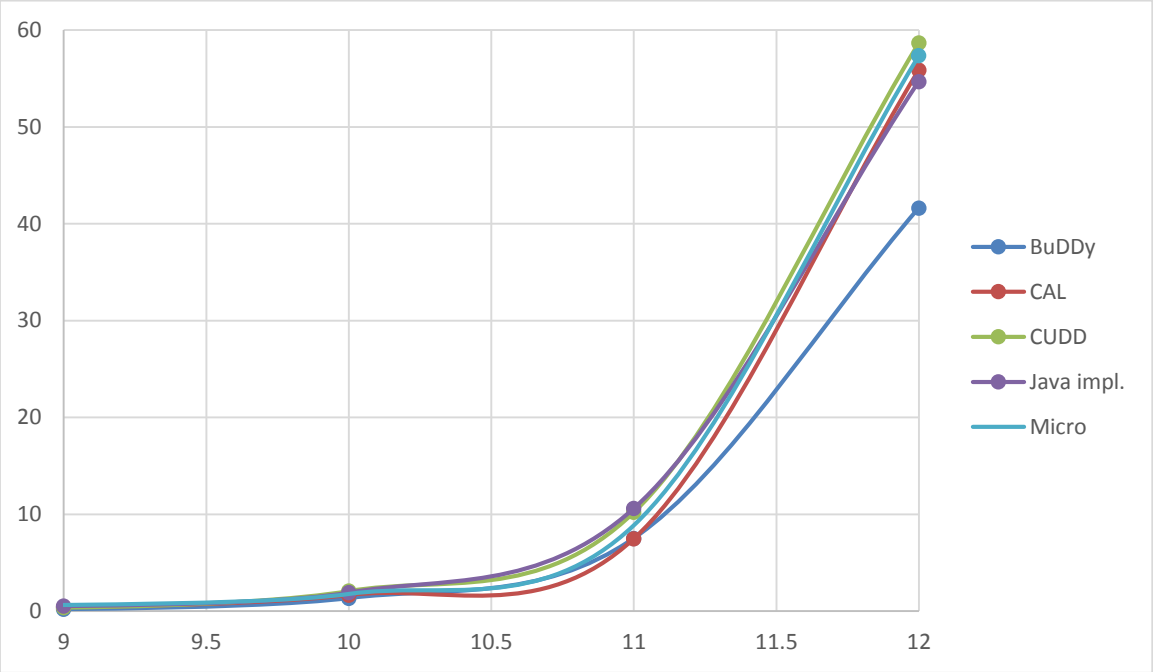
Micro. BDD, где каждый узел занимает только 16 байт в памяти. Это достигается плотной упаковкой битов и ограничением максимального числа переменных BDD до 2 ^ 10 = 1024. Из-за дополнительных вычислений, необходимых для упаковки / распаковки, эта BDD немного медленнее, чем Java impl., но она также использует на 20% меньше памяти.

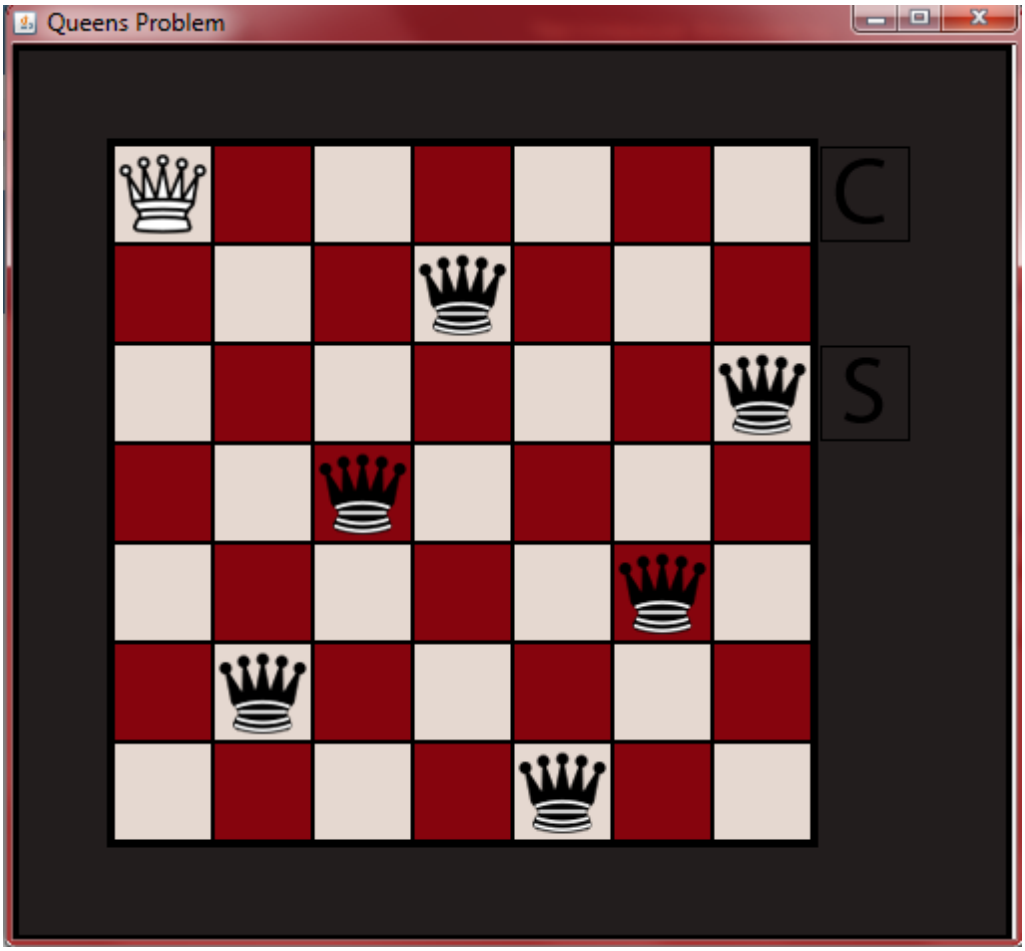
Результаты сравнения в таблице:

Lib\размер доски	9x9	10x10	11x11	12x12
BuDDy	0.203c	1,340c	7,519c	41,637c
CAL	0,437c	1,669c	7,475c	55,881c
CUDD	0,374c	2,075c	10,202c	58,673c
Java impl.	0,530c	1,918c	10,609c	54,682c
Micro	0,624c	1,760c	8,846c	57,346c

Такие временные затраты связаны с медленным выделение памяти и неуправляемым сборщиком мусора Java. После инициализации и построения,

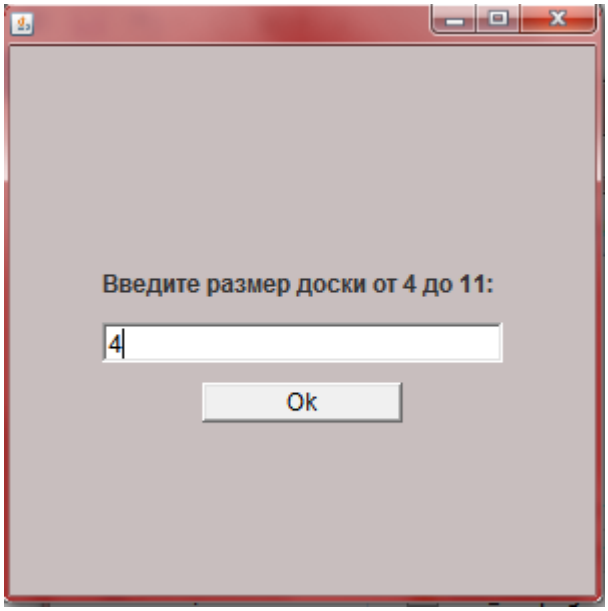
данные библиотеки находят решение любой позиции мгновенно (если позицию возможно решить вообще).



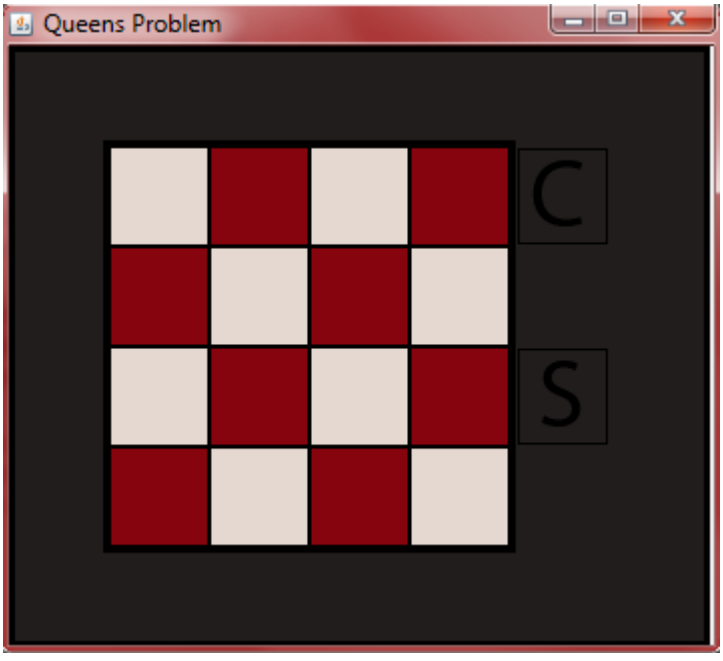


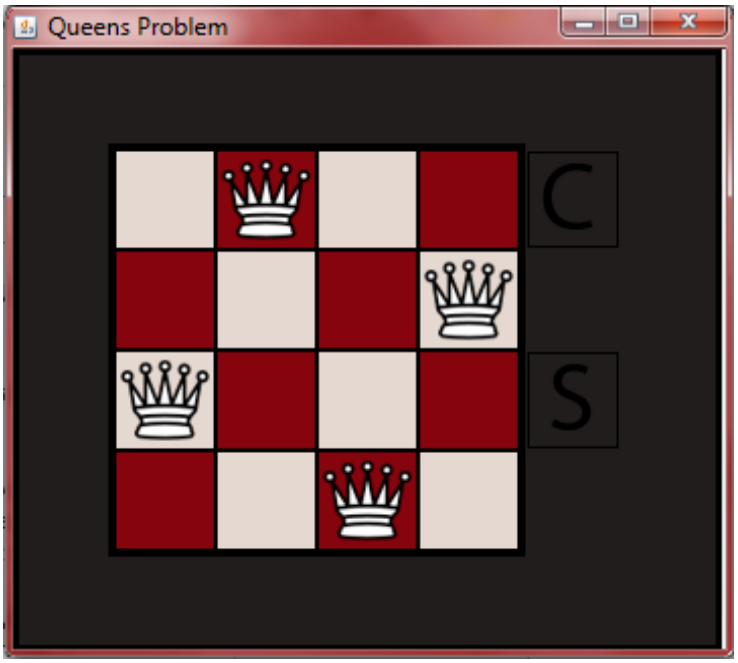
5.Описание работы программы

Как только мы запускаем программу, перед нами всплывает окно для ввода размера доски:

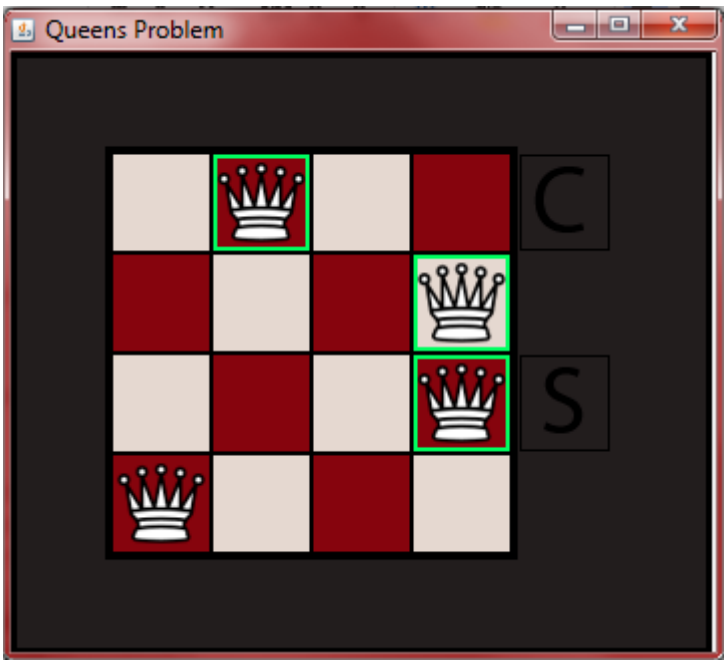


После того как пользователь нажмет кнопку «Ок», программа начнет инициализацию и построение bdd (это может занять некоторое время). Далее перед пользователем появится пустая доска, на которой можно расставлять и убирать ферзей. Чтобы отчистить доску, нажмите кнопку «C». Чтобы решить выставленную на доске позицию, если это возможно, нажмите «S» (если доска была пустой, ты выведется случайное решение данной задачи). Если выбранную позицию невозможно решить, то нажатие кнопки «S», не приведет ни к каким изменениям:





Даже после решения задачи, пользователь может продолжить редактирование доски. Приложение снабжено системой проверки позиции. Если вы поставите ферзя под удар других ферзей, конфликтные ферзи будут подсвечены зеленым.



6.Заключение

В заключение хочется сказать, что в настоящее время известно очень много «переборных» задач. Именно грамотная организация перебора и различные эвристики помогают решать их довольно быстро. Задача о расстановке ферзей является классической, и на ее примере я прекрасно осознал все преимущества хорошо выполненного поиска с возвратом. Язык Java не очень подходит для переборных задач ввиду его непредсказуемой работой с памятью. Как мы видим по тестам производительности наш алгоритм не самый быстрый, но после инициализации и построения BDD позволяет получить ответ по щелчку пальцев.

С помощью BDD мы организовали достаточно эффективную структуру для решения задачи о расстановки ферзей.