1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт компьютерных наук и кибербезопасности**

**Курсовая работа**

1. «Оптимизация программы решения криптарифма»
2. по дисциплине «Языки программирования»
3. Выполнили
4. студенты гр. 5131001/30003 Шевчук Н.Е.
5. Мальцев М.А.

1. Преподаватель Семьянов П.В.
3. Санкт-Петербург

2024

ОГЛАВЛЕНИЕ

[Введение 3](#_Toc185487455)

[1. Цель 3](#_Toc185487456)

[2. Задачи 3](#_Toc185487457)

[3. Теоретическая справка 3](#_Toc185487458)

[4. Ход работы 4](#_Toc185487459)

[4.1. Анализ текущего решения 4](#_Toc185487460)

[4.2. Оптимизация структур данных 4](#_Toc185487461)

[4.3. Оптимизация алгоритма 5](#_Toc185487462)

[4.4. Машинно-независимая оптимизация 6](#_Toc185487463)

[4.5. Машинно-зависимая оптимизация 9](#_Toc185487464)

[5. Анализ результатов 12](#_Toc185487465)

[6. Вывод 13](#_Toc185487466)

[7. Список источников 14](#_Toc185487467)

Введение

Оптимизация программы – это процесс улучшения ее производительности, эффективности и скорости работы. Для достижения эффективной оптимизации следует выбирать методы, наиболее подходящие для конкретных задач и требований. Результаты работы по оптимизации могут быть значительными и повлиять на успех проекта в целом. В данной курсовой работе мы рассмотрим процесс оптимизации решения криптарифма, значительно повлиявшей на скорость выполнения кода.

# Цель

Оптимизировать программу, выполняющую решение криптарифма, введённого пользователем. Сравнить время выполнения кода на разных этапах оптимизации.

# Задачи

1. Замерить время выполнения изначальной версии программы при различных входных данных.
2. Оптимизировать алгоритм и замерить время выполнения программы после оптимизации алгоритма решения.
3. Реализовать машинно-независимую оптимизацию и замерить время выполнения программы после машинно-независимой оптимизации.
4. Реализовать машинно-зависимую оптимизацию и замерить время выполнения программы после машинно-зависимой оптимизации.
5. Сравнить и проанализировать результаты предыдущих пунктов. Предоставить их в графическом виде.

# Теоретическая справка

Криптарифм - это математический ребус, в котором зашифрован пример на выполнение одного из арифметических действий. При этом одинаковые цифры шифруются одной и той же буквой, а разным цифрам соответствуют различные буквы. Считается, что никакое число не должно начинаться с нуля. Кроме того, еще одно требование к правильному криптарифму: он должен иметь единственную возможную расшифровку.



Рисунок . Стандартный криптарифм

# Ход работы

## Анализ текущего решения

Текущий алгоритм работы – перебор с возвратом по уникальным буквам из криптарифма. Сначала составляется массив из уникальных букв криптарифма. Затем в функции try каждой букве из массива abc присваивается значение цифры и проверяется верно ли изначальное арифметическое равенство. Далее в созданном массиве val\_abs пытаемся увеличить значения цифр так, чтобы арифметическое равенство осталось верным и не было повторяющихся цифр.

Время выполнения изначального алгоритма на примере “ROBERT – GERALD=DONALD”: .

## Оптимизация структур данных

Для удобства хранения букв и разбиения их по столбцам были созданы структуры Variable и Column. Variable содержит поля letter – уникальная буква криптарифма, value – присвоенная ей цифра-решение. Column содержит указатели на variables трех букв столбца. При обработке введённых криптарифмов заполняются статические массивы columns и variables, для удобства получения букв определённого столбца. Так как разбиение по столбцам происходит только один раз в течении выполнения программы, то время, затраченное на него незначительно.

Также важной для решения является хэш-таблица used\_digits[10], отвечающая за то, была ли уже использована конкретная цифра. Из кода рекурсивной функции при проверке обращение происходит напрямую к необходимой цифре. Так как в такой простейшей хэш-таблице нет коллизий, то доступ к необходимой цифре происходит за O(1), в отличие от изначального варианта, где проверка занятости цифры происходила через линейный поиск в массиве, что соответствовало временной сложности O(n).

|  |  |
| --- | --- |
| Изначальный вариант | Оптимизированный вариант |
| //проверка на занятость цифры  bool inside(char x, int stop, char arr[]) {  for (int y = 0; y < stop; y++) if (arr[y] == x) return true;  return false;  }  //пример вызова проверки  if (!inside(j, i, max\_valAbc)) | //хэш-таблица для проверки на занятость цифры  int used\_digits[10] = {0};  //пример проверки  if (used\_digits[7]) |

## Оптимизация алгоритма

В изначальной версии алгоритм перебора с возвратом был обращен к массиву уникальных букв и для того, чтобы удостовериться в правильности решения проверялось полное арифметическое выражение (которое получается только при максимальной глубине рекурсии), соответствующее криптарифму при подборе цифры для последней из уникальных букв. Очевидно, что при такой реализации проверяется большое количество изначально тупиковых ветвей.

Поэтому в новом алгоритме перебор с возвратом происходит по столбцам, начиная с последнего (т.е. с единиц). Условно, мы представляем криптарифм в виде сложения столбиком и рассматриваем правильно ли подобраны цифры для каждого конкретного столбца, после чего рекурсивно переходим к анализу следующего столбца. Если на каком-то из столбцов не получается корректно подобрать цифры (с учетом уже присвоенных буквам цифр), то происходит откат на предыдущий столбец, т.е. установленные цифры сбрасываются и ищется следующая корректная комбинация. Для корректного учета переноса между разрядами используется отдельная переменная carry\_in, передающаяся между столбцами (т.е. являющаяся входным параметром рекурсивной функции).

В основной рекурсивной функции solve происходит следующее:

1. Начальное условие выхода из рекурсии - столбец последний и сумма его слагаемых не требует переноса в следующий разряд.
2. Присваивание буквам цифр, если в предыдущих столбцах они уже были присвоены.
3. Отдельно рассмотрен случай, когда в последнем столбце нет одного или обоих слагаемых (т.е. длина уменьшаемого больше чем вычитаемое и/или разность. Это сделано для того, чтобы при дальнейшей обработке столбца не возникала ошибка доступа по нулевому указателю.
4. Двумя циклами перебираются цифры для разности и вычитаемого. Если одному из них уже была присвоена цифра, то при проверке используется она. Если букве уменьшаемого уже была присвоена цифра, то их сумма сравнивается с ней. Если не была, то присваивается полученная сумма, если такая цифра еще не была использована.
5. Получив верное равенство *разность+вычитаемое=уменьшаемое*, вызывается эта же функция для следующего столбца. Если она не возвращает true, то установленные цифры сбрасываются и перебор продолжается.

Таким образом, отсеивается множество ветвей решения, гарантированно не приводящих к ответу.

Время выполнения оптимизированного алгоритма на примере “ROBERT – GERALD=DONALD”: .

## Машинно-независимая оптимизация

1. Удаление лишних переменных

При машинно-независимой оптимизации множество флагов типа bool были удалены. Вместо их использования, при необходимости проверки, сравниваются исходные значения типа int.

1. Вынесение инвариантов

Так как основная функция решения solve() рекурсивно подбирает цифры для различных столбцов, то каждый из находящихся в ней двух вложенных циклов вызывается значительное число раз. Следовательно, необходимо вынести из циклов инварианты, для сокращения действий на каждой итерации цикла. Но в данном случае, в зависимости от значения инварианта менялась процедура корректного перебора цифр. Поэтому в такой ситуации, вынос инварианта означает разбиение всего решения на несколько логических вариантов. Инварианты – значения цифр, уже присвоенных буквам, определяются в начале рекурсивной функции, а уже затем, в зависимости от их значения, проверка и перебор цифр идет только по одному варианту.

Теперь, если при вызове функции solve() для столбца всем буквам в нем уже были назначены цифры, перебора цифр не происходит. Если назначено одно из слагаемых или буквы слагаемых одинаковые, то перебирается цифра только для одного слагаемого. И только в случае, если цифры для обоих слагаемых еще не назначены происходит перебор по вложенным циклам.

|  |  |
| --- | --- |
| До МНЗ | После МНЗ |
| for (int digit1 = 0; digit1 <= 9; digit1++) {  if (addend1\_value == -1 && used\_digits[digit1]) continue;  if (addend1\_value == -1 && (addend1->letter == addend1\_[0] || addend1->letter == addend2\_[0] || addend1->letter == result\_word[0]) && digit1 == 0) continue;  if (addend1\_value == -1) {  addend1->value = digit1;  used\_digits[digit1] = 1;  }  if (addend1->letter == addend2->letter) {  addend2\_value = is1 ? addend1\_value : digit1;  addend2->value = addend2\_value;  is2 = true;  }  for (int digit2 = 0; digit2 <= 9; digit2++) {  if (addend2\_value == -1 && used\_digits[digit2]) continue;  if (addend2\_value == -1 && (addend2->letter == addend1\_[0] || addend2->letter == addend2\_[0] || addend2->letter == result\_word[0]) && digit2 == 0) continue;  if (addend2\_value == -1) {  addend2->value = digit2;  used\_digits[digit2] = 1;  }  if (addend1->letter == result->letter) {  result\_value = is1 ? addend1\_value : digit1;  result->value = result\_value;  }  if (addend2->letter == result->letter) {  result\_value = is2 ? addend2\_value : digit2;  result->value = result\_value;  }  int column\_sum = (is1 ? addend1\_value : digit1) + (is2 ? addend2\_value : digit2) + carry\_in;  int carry\_out = column\_sum / 10;  int result\_digit = column\_sum % 10;  if (result\_value == -1 && !used\_digits[result\_digit]) {  result->value = result\_digit;  used\_digits[result\_digit] = 1;  }  else if (result\_value != result\_digit) {  if (addend2\_value == -1) {  addend2->value = -1;  used\_digits[digit2] = 0;  }  continue;  }  if (solve(col\_index - 1, carry\_out)) {  return true;  }  if (result\_value == -1) {  result->value = -1;  used\_digits[result\_digit] = 0;  }  if (addend2\_value == -1) {  addend2->value = -1;  used\_digits[digit2] = 0;  }  }  if (addend1\_value == -1) {  addend1->value = -1;  used\_digits[digit1] = 0;  }  } | //слагаемым присвоена цифра, результату - нет  if (addend1\_value != -1 && addend2\_value != -1 && result\_value == -1) {...}  //всем буквам столбца присвоена цифра  else if (addend1\_value != -1 && addend2\_value != -1 && result\_value != -1) {...}  //буквы слагаемых одинаковые  if (addend1->letter == addend2->letter) {  //никакой букве не пресвоена цифра  if (addend1\_value == -1 && addend2\_value == -1 && result\_value == -1) { ... }  //результату присвоена цифра, слагаемым - нет  if (addend1\_value == -1 && addend2\_value == -1 && result\_value != -1) { ... }  }  //буквы слагаемых разные  else {  //слагаемым не присвоены цифры  if (addend1\_value == -1 && addend2\_value == -1) { ... }  //первому слагаемому присвоена цифра, второму - нет  else if (addend1\_value != -1 && addend2\_value == -1) { ... }  //второму слагаемому присвоена цифра, первому - нет  else if (addend1\_value == -1 && addend2\_value != -1) { ... }  } |

1. Развертка циклов

Во всех вариантах, перечисленных выше, были развернуты циклы. В варианте, при котором буквы слагаемых разные и ни одному из них еще не назначена цифра развернут был только внутренний цикл для перебора значений цифры второго слагаемого. Развертка позволяет избежать операции сравнения и инкрементирования (в отличие от объявления цикла) при каждой итерации. Кроме того, проверка, не позволяющая присвоить букве цифру ноль, если эта буква первая в каком-либо из слов криптарифма, теперь осуществляется только при проверке цифры ноль.

Время выполнения программы после машинно-независимой оптимизации на примере “ROBERT – GERALD=DONALD”: .

## Машинно-зависимая оптимизация

В настройках проекта в Visual Studio 2022 были найдены опции компилятора, обеспечивающие наиболее быстрый код.

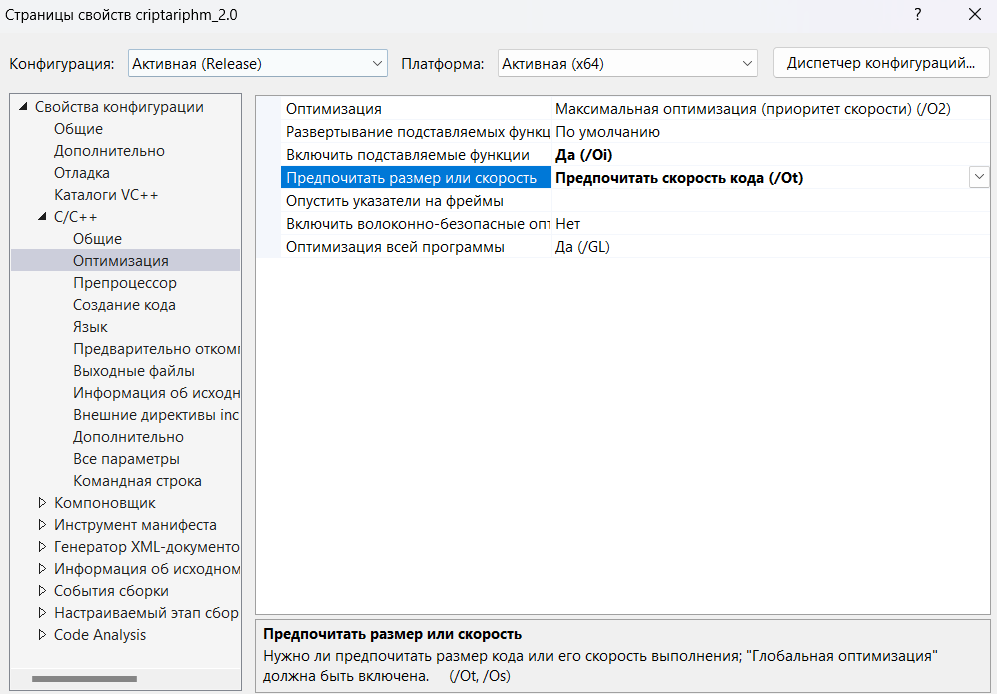


Рисунок . <C/C++> - <Оптимизация>

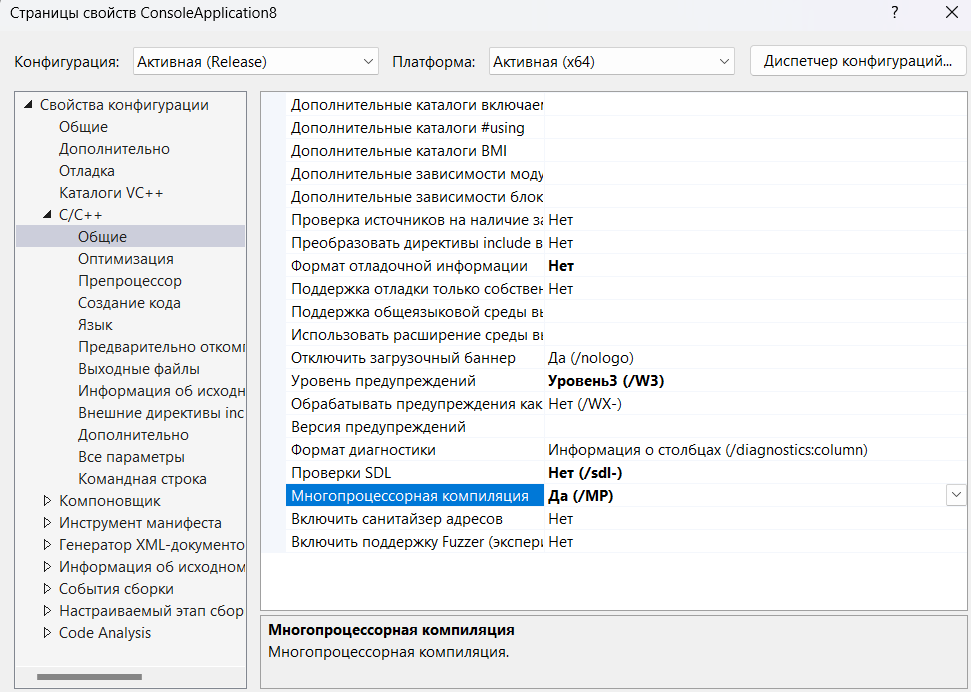


Рисунок . <C/C++> - <Общие>

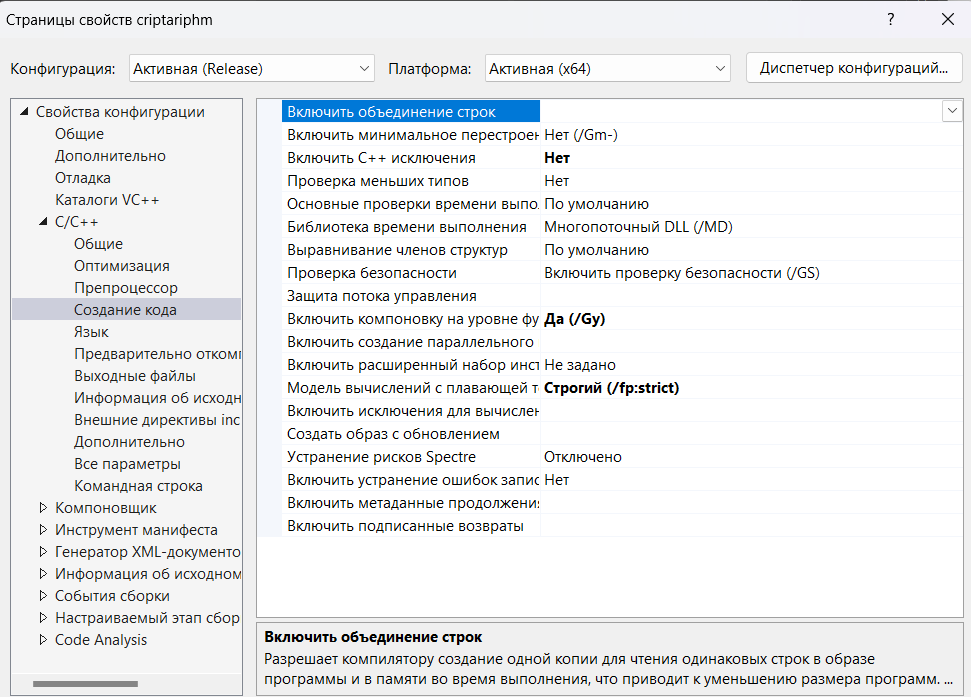


Рисунок . <C/C++> - <Создание кода>

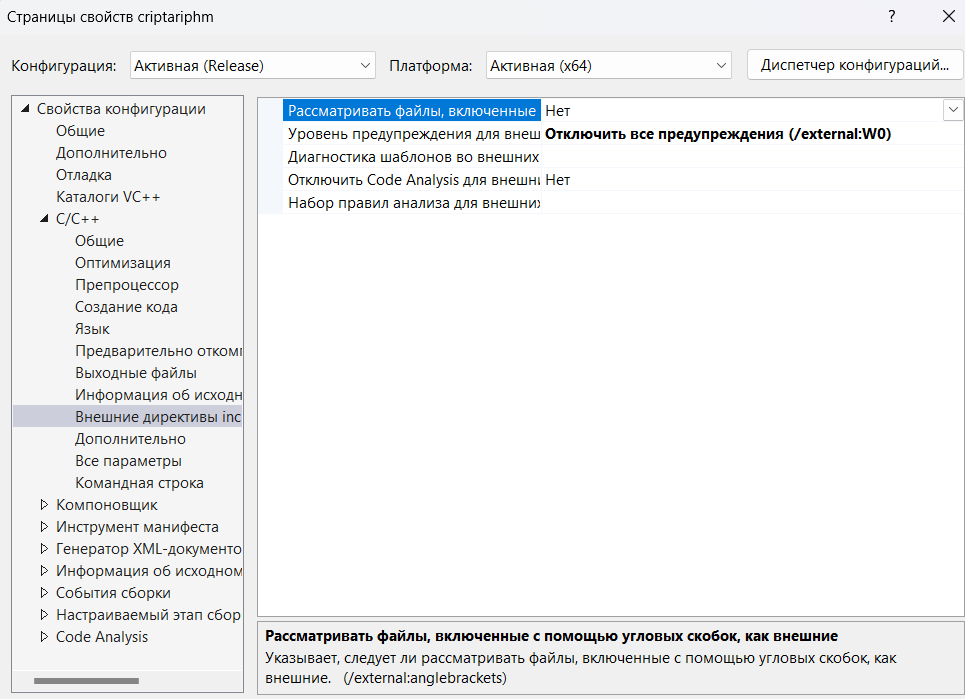


Рисунок . <C/C++> - <Внешние директивы include>

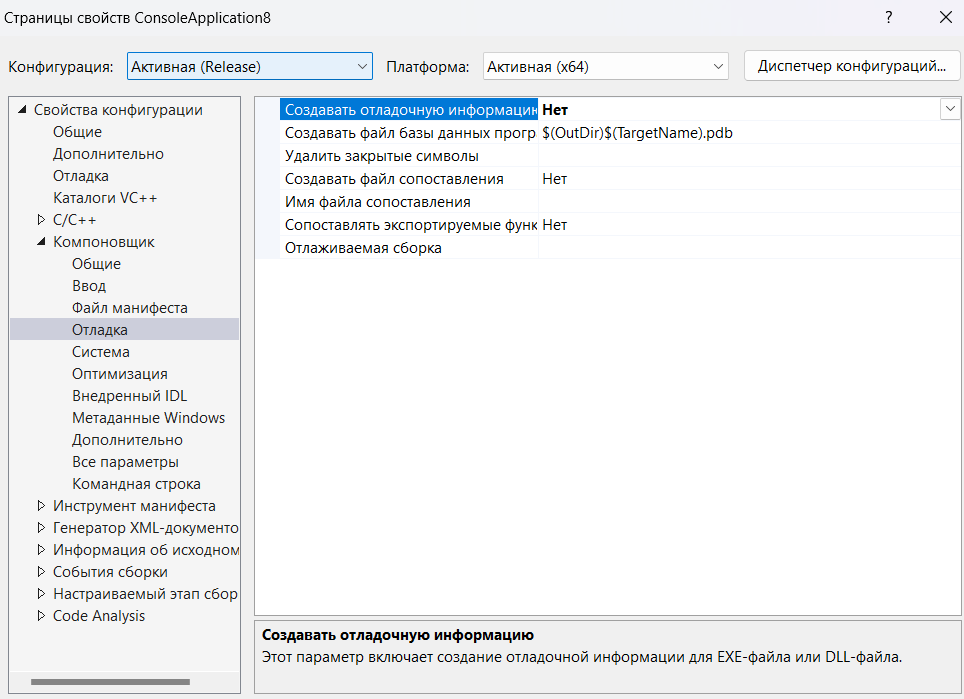


Рисунок . <Компоновщик> - <Отладка>

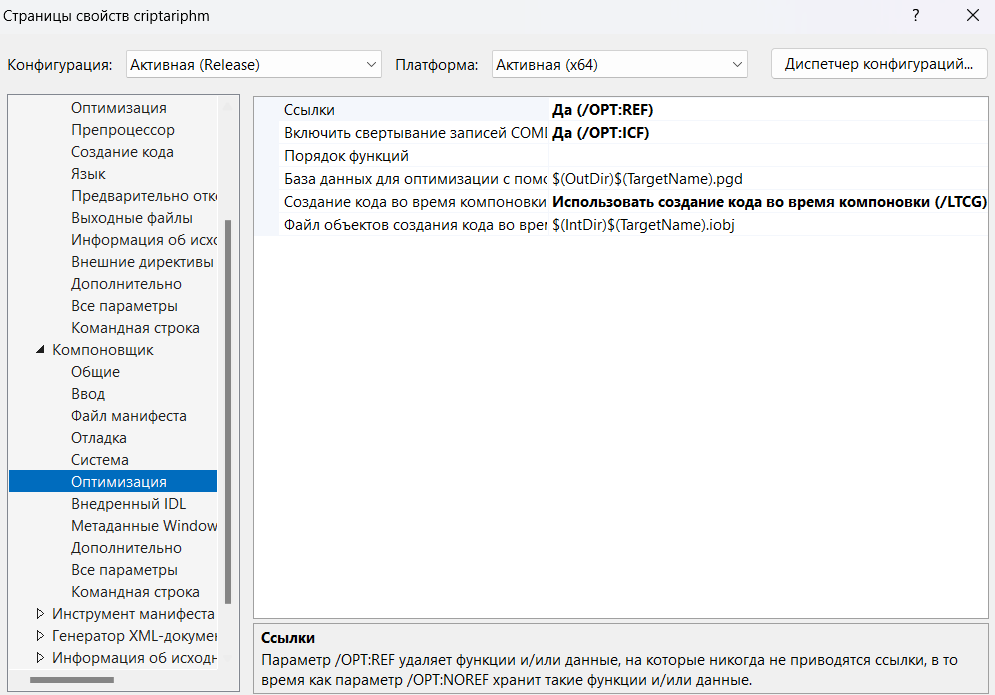


Рисунок . <Компоновщик> - <Оптимизация>

Время выполнения после машинно-зависимой оптимизации на примере “ROBERT – GERALD=DONALD”: .

# Анализ результатов

Рисунок . График зависимости времени выполнения программы от этапа оптимизации.

Этапы оптимизации: 1->2 – Оптимизация алгоритма; 2->3 – Машинно-независимая оптимизация; 3->4 – Машинно-зависимая оптимизация.

Из анализа графика следует, что наиболее весомой оптимизацией оказалась оптимизация алгоритма, а машинно-зависимая оптимизация дала совсем незначительный результат.

Также после всех упомянутых выше оптимизаций программа была протестирована на различных криптарифмах:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Мощность алфавита | Криптарифм | До оптимизации, , c | После , c | Отношение |
| 8 | MONEY-MORE=SEND | 0.2680 | 0.0001525 | 1875.4 |
| RIGHT-WRONG=WRONG | 0.3060 | 0.0001376 | 2223.8 |
| 9 | HAPPY-TIGER=YEAR | 0.6500 | 0.0001929 | 3369.6 |
| DANCER-DANCE=SQUARE | 0.7840 | 0.0001781 | 4387.2 |
| 10 | PARENT-MOTHER=FATHER | 1.3710 | 0.0001399 | 9799.8 |
| TRUDEAU-ELLIOTT=PIERRE | 1.3860 | 0.0001440 | 9625.0 |
| ROBERT-GERALD=DONALD | 1.304 | 0.0001565 | 8332.3 |

Таблица . Зависимость времени выполнения программы от конкретного криптарифма и мощности алфавита

Из анализа таблицы выше следует, что время выполнения оптимизированного кода не зависит напрямую от мощности криптарифма, а только от строения конкретного ребуса. Удалось добиться значительной оптимизации программы по времени выполнения в пределах от 2 до 10 тысяч раз.

# Вывод

В ходе курсовой работы была оптимизирована программа решения криптарифма с помощью оптимизации алгоритма, а также машинно-зависимых и машинно-независимых оптимизаций. Были наглядно показаны различия начальной и оптимизированной версий. Наиболее значительной оптимизацией стала оптимизация алгоритма.

Таким образом, методы оптимизации и ускорения программы могут значительно улучшить ее производительность и эффективность.

# Список источников

1. Филипп Н. Хислей, “Генерация высококачественного кода для программ, написанных на Си”.
2. <https://www.math.uni-bielefeld.de/~sillke/PUZZLES/ALPHAMETIC/> - примеры криптарифмов.