1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ**

1. **«Оптимизация приложения «Эрудит»»**
2. по дисциплине «Языки программирования»

Выполнили   
студенты гр. 5131001/20003 Черникова Василиса Михайловна

<*подпись*>

1. Старший преподаватель Семьянов Павел Валентинович

<*подпись*>

1. Санкт-Петербург
2. 2024

Содержание

[ВВЕДЕНИЕ 3](#_Toc161411170)

[ОСНОВНАЯ ЧАСТЬ РАБОТЫ 4](#_Toc161411171)

[**1.** **ОПТИМИЗАЦИЯ НА УРОВНЕ АЛГОРИТМА** 4](#_Toc161411172)

[**1)** **Подбор структуры для оптимального хранения данных.** 4](#_Toc161411173)

[**2)** **Замена типа сортировки.** 8](#_Toc161411174)

[**3)** **Устранение логических ошибок в коде.** 8](#_Toc161411175)

[**4)** **Замена рекурсивной функции** 8](#_Toc161411176)

[**5)** **Результаты и выводы** 12](#_Toc161411177)

[**2.** **ОПТИМИЗАЦИЯ НА МАШИННО-НЕЗАВИСИМОМ УРОВНЕ** 12](#_Toc161411178)

[**1)** **Отказ от цикла** 12](#_Toc161411179)

[**2)** **Вынос инвариантного кода** 12](#_Toc161411180)

[**3)** **Оптимизация переходов** 13](#_Toc161411181)

[**4)** **Удаление недостижимого кода.** 14](#_Toc161411182)

[**5)** **Развертка циклов.** 14](#_Toc161411183)

[**3.** **ОПИТИМИЗАЦИЯ НА МАШИННО-ЗАВИСИМОМ УРОВНЕ** 15](#_Toc161411184)

[**1)** **Замена индекса на указатель** 15](#_Toc161411185)

[**2)** **Замена типа int на char** 16](#_Toc161411186)

[**3)** **Оптимизация с помощью внутренних возможностей VS** 16](#_Toc161411187)

[**4.** **ТЕСТИРОВАНИЕ И РЕЗУЛЬТАТЫ ПРОГРАММЫ** 16](#_Toc161411188)

[ЗАКЛЮЧЕНИЕ 17](#_Toc161411189)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ (ИСТОЧНИКОВ) 18](#_Toc161411190)

# ВВЕДЕНИЕ

В данной курсовой работе рассмотрен процесс оптимизации игры «Эрудит», который значительно повысил его скорость выполнения. Оптимизация является важным этапом разработки, который позволяет улучшить производительность и удобство использования программы.

**Цель работы:** приобретение навыков в команде при оптимизировании совместно разработанного проекта на 1 курсе.

**Задачи:   
1.** Оптимизировать проект на уровне алгоритма  
**2.** Оптимизировать на машинно-независимом уровнем   
**3.** Оптимизировать на машинно-зависимом уровне

**4.** Проанализировать время выполнения программы до и после проведенных оптимизаций и сделать выводы

**ОСНОВНАЯ ЧАСТЬ РАБОТЫ**

1. **ОПТИМИЗАЦИЯ НА УРОВНЕ АЛГОРИТМА**
2. **Подбор структуры для оптимального хранения данных.**

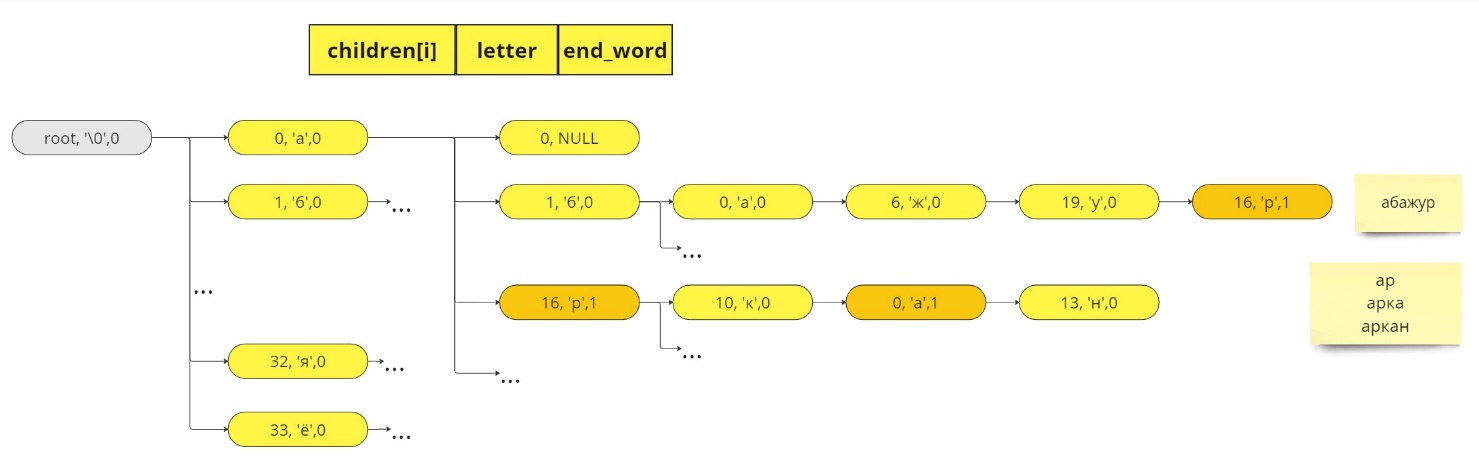
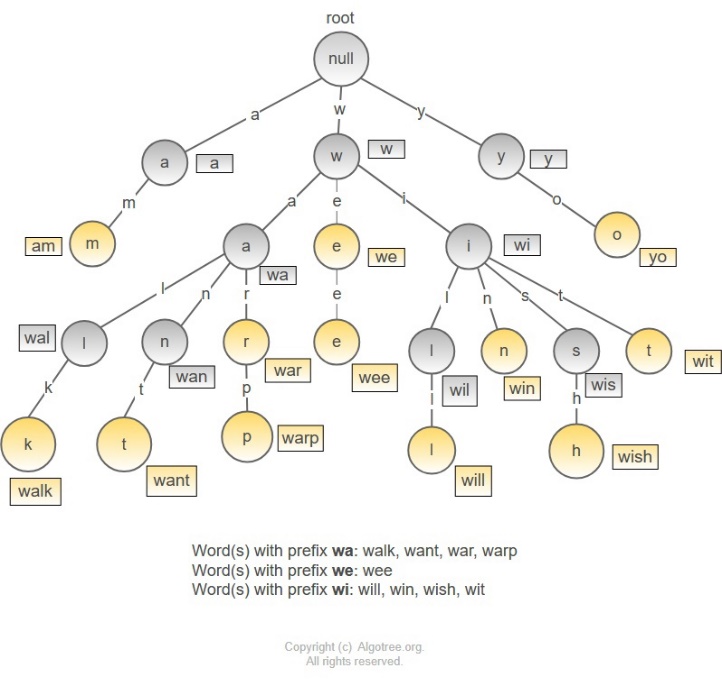
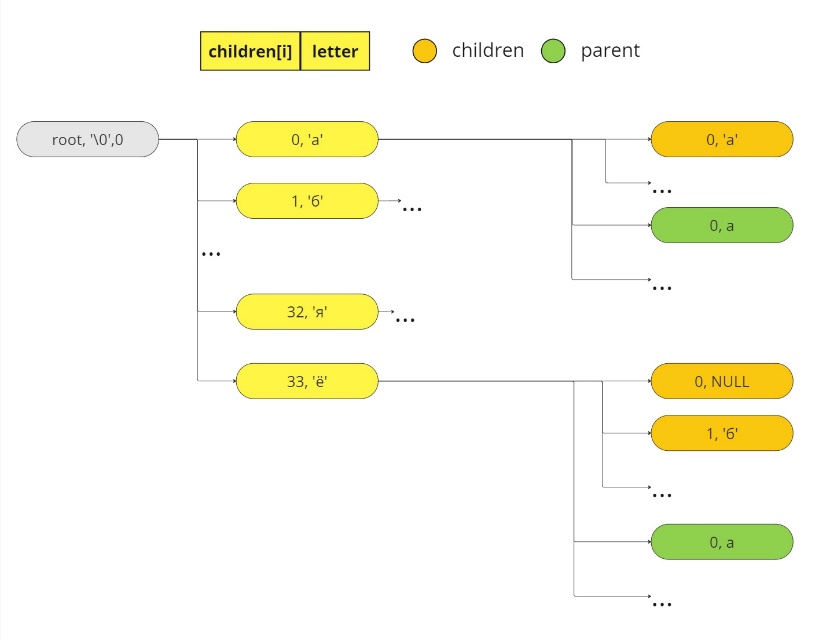
Изначально в программе словарь хранился в виде 14 файлов слов длины от 2 до 15 букв соответственно. При анализе программы стало понятно, что для поиска слова в словаре не выгодно обращаться каждый раз к нужному файлу, поскольку это включает в себя лишние действия: открытие, чтение, закрытие.   
 Принято решение выгрузить словарь в новую структуру, которая будет хранить все слова по буквам. Самым оптимальным решением выбрана структура дерева. Ниже приведена её схема. (См. рис. 1)

Рисунок 1. Схема хранения словаря и структуры для хранения статистики.

Для оптимизации алгоритма потребовалось добавление новой структуры данных для хранения статистики двухбуквенных сочетаний. Она реализована в виде трехуровневого дерева. Его инициализация происходит вместе со словарем.

Сравним части кода старого алгоритма и нового, и посмотрим, как изменилась скорость всей программы.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| int slovar(char sl[15]) {  char dop[17];  int c = strlen(sl);  switch (c)  {  case 2:  fopen\_s(&slov, "2\_new.txt", "r");  break;  case 3:  fopen\_s(&slov, "3\_new.txt", "r");  break;  //и так далее до case 15  while (!feof(slov)) {  fgets(dop, 17, slov);  c++;  if (sl[0] == dop[0]) {  int t = 0;  for (int u = 1; u < strlen(sl); u++) {  t += 1;  if (sl[u] != dop[u]) {  t = -10;  break;  }  }  if (t != -10) {  fclose(slov);  return 1;  }  }  if ((int)sl[0] < (int)dop[0]) {  fclose(slov);  return 0;  }  }  fclose(slov);  return 0;  } | struct TrieNode {  TrieNode\* children  [33];  bool end\_of\_word;  char letter;  TrieNode() {  end\_of\_word = false;  for (short int i = 0; i < 33; i++) {  children[i] = NULL;  }  letter = '\0';  }  };  TrieNode root;  void Insert(char str[17]) {  TrieNode\* current = &root;  for\_stat\* cur\_stat = NULL;  char c;  for (size\_t i = 0; i < strlen(str) - 1; i++) {  c = str[i];  //printf("%c %d\n ", c,c);  if (str[i] == -72) {  c = 0;  }  cur\_stat = (&root\_stat)->children[c + 32];//буква, по которой собираем статистику  if (i > 0)  if (str[i - 1] == -72) {  (cur\_stat->chislo\_p[32])++;  if (cur\_stat->parent\_ss[32] == NULL) {  cur\_stat->parent\_ss[32] = new for\_stat;  cur\_stat->parent\_ss[32]->letter = str[i];  }  }  else {  (cur\_stat->chislo\_p[str[i - 1] + 32])++;  if (cur\_stat->parent\_ss[str[i - 1] + 32] == NULL) {  cur\_stat->parent\_ss[str[i - 1] + 32] = new for\_stat;  cur\_stat->parent\_ss[str[i - 1] + 32]->letter = str[i];  }  }  if (i < strlen(str) - 2)  if (str[i + 1] == -72) {  (cur\_stat->chislo\_c[32])++;  if (cur\_stat->children\_ss[32] == NULL) {  cur\_stat->children\_ss[32] = new for\_stat;  cur\_stat->children\_ss[32]->letter = str[i];  }  //continue;  }  else {  (cur\_stat->chislo\_c[str[i + 1] + 32])++;  if (cur\_stat->children\_ss[str[i + 1] + 32] == NULL) {  cur\_stat->children\_ss[str[i + 1] + 32] = new for\_stat;  cur\_stat->children\_ss[str[i + 1] + 32]->letter = str[i];  }  }  }for (size\_t i = 0; i < strlen(str) - 1; i++) {  int c = str[i];  //printf("%c", c);  if (str[i] == -72) {  c = 0;  }  if (current->children[c + 32] == NULL) {  current->children[c + 32] = new TrieNode;  current->children[c + 32]->letter = str[i];  }  current = current->children[c + 32];  }  current->end\_of\_word = true;}  TrieNode\* Search(char str[16]) {  TrieNode\* current = &root;  for (size\_t i = 0; i < strlen(str); i++) {  int c = str[i];  if (str[i] == 'ё') {  c = 0;  }  if (current->children[c - 'а']) {  current = current->children[c - 'а'];  }  else {  current = NULL;  break;  }  }  return current;  }  TrieNode\* current2;  void slovar() {  fopen\_s(&slov, "slovar.txt", "r");  char dop[17] = "";  while (!feof(slov)) {  fgets(dop, 17, slov);  if (dop[0] != 'x') {  Insert(dop);  }  }  fclose(slov);  } |

Таблица 1. Результат оптимизации алгоритма.

В функции void Insert () и TrieNode\* Search() слово заносится и ищется в дереве отдельно по буквам. Для этого было прописано условие, где позиция буквы определяется с помощью таблицы кодировки ASCII. Для буквы «ё» имеется отдельное условие, так как по данной таблице она не соответствует алфавитному порядку и имеет код 200. Так же в функции void Insert () была установлена проверка на наличие пробелов в словаре, для корректного заполнения дерева.

В новой функции void slovar() теперь вместо 14 файлов считывается только один, который теперь содержит все слова разной длины. Кроме того, с помощью новой структуры не надо работать с множеством файлов во время игры. Словарь теперь хранится в дереве, которое формируется до её начала.

Поиск слов по дереву оказался действительно выгоднее, чем поиск по словарю, который имеет линейную сложность. В новой структуре сложность: O (L ) , где L- длина слова,1< L<16, а в словаре O(N), где N – кол-во слов в словаре(15<=N<9000).

1. **Замена типа сортировки.**

Для выбора наилучшего хода раннее в программе была реализована сортировка пузырьком по количеству баллов. Было принято решение организовать сортировку вставками, поскольку именно она более эффективна для связных списков. Время игрока легкого уровня не изменилось, так как в его случае находятся и сортируются всего 3 слова. Время хода игрока сложного уровня уменьшилось.

1. **Устранение логических ошибок в коде.**

В функции int analise\_2() проверка на пустоту строки, была удалена и перенесена в функцию int analise(). Не заходим в функцию int analise\_2(), если строка пустая. Проверка происходит до того, как мы заходим в функцию int analise\_2(), таким образом был удален лишний цикл.

1. **Замена рекурсивной функции**

Помимо оптимизаций, описанных выше, была оптимизирована функцию int proverka\_slova() с целью избавиться от рекурсии и ускорить работу программы. В данную функцию отправляется комбинация, состоящая из букв, точек привязки и 'z', составленная в функции int analise\_2(). Затем в функции на места 'z' и точек привязки подставлялись буквы, выданные игроку. Необходимо перебирать и проверять множество вариантов, отсюда и появилась рекурсия.

Было принято решение заранее составлять все размещения букв. Для этого были реализованы две вспомогательные функции void swap() и bool NextSet(), которые составляли все варианты размещений индексов массива букв игрока, а затем по этим индексам буквы подставлялись в пришедшую комбинацию.   
Рассмотрим на примере:

В функцию пришла комбинация ‘кzzzлевk’ и выданные буквы ‘лороваб’ (т.е. есть 4 места для расстановки 7 букв). Из букв ‘лороваб’ уже составляются всевозможные комбинации по 4 буквы, затем вставляются в пропуски. Далее происходит проверка на возможность нахождение каждой из двух букв рядом из дополнительной структуры. После прохождения условий полученное слово отправляется на проверку. По итогу функция выдаст нам один из вариантов хода - слово ‘королева’.

Сравним части кода старого алгоритма и нового, и посмотрим, как изменилась скорость всей программы.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| int proverka\_slova(char sl[15], int uk, int kol, int st, int I, int J, int number, int letter, int num\_z, int ukaz, char mas[7]) {  char t;  int count = 0;  int masi = 0;  setlocale(LC\_ALL, ".1251");  for (int j = 0; j < 7; j++) {  for (int i = uk; i <= J - I; i++) {  if (mas[j] != 'z' && (sl[i] == 'z' || sl[i] == 'k')) {  sl[i] = mas[j];  if ((i == 0) && (mas[j] == 'ь' || mas[j] == 'ъ' || mas[j] == 'ы')) {  break;  }  t = mas[j];  using\_letters[letter] = t;  mas[j] = 'z';  proverka\_slova(sl, i + 1, kol - 1, st, I, J, number, letter + 1, num\_z, ukaz, mas);  if (kol == 1 && ((DOP > 0 && ukaz % 2 == 1) || ukaz % 2 == 0)) {  if (slovar(sl) == 1) {  masi = massa\_sl(number, I, J, sl);  PutIn1(st, number, I, J, masi, using\_letters, sl, num\_z);  DOP -= 1;  if (ukaz % 2 == 1 && DOP == 0) {  using\_letters[letter] = NULL;  mas[j] = t;  sl[i] = 'z';  return 0;  }  }  }  using\_letters[letter] = NULL;  mas[j] = t;  sl[i] = 'z';  }  }  }  return -1;  } | void swap(int i, int j)  {  int s = a[i];  a[i] = a[j];  a[j] = s;  }  bool NextSet(int n, int m)  {  int j;  do  {  j = n - 1;  while (j != -1 && a[j] >= a[j + 1]) j--;  if (j == -1)  return false;  int k = n - 1;  while (a[j] >= a[k]) k--;  swap(j, k);  int l = j + 1, r = n - 1;  while (l < r)  swap(l++, r--);  } while (j > m - 1);  return true;  }  void proverka\_slova(char sl[16], short st, short I, short J, short number, short num\_n\_z, short ukaz, char mas[7]) {  for (short i = 0; i < 7; i++) {  a[i] = i;  }  char sl2[16];  strcpy(sl2, sl);  char m;  short c = 0;  short p = 0;  short T = 1;  short masi;  short num\_z = strlen(sl) - num\_n\_z;  for (char\* sli = &sl[0]; sli <= &sl[J - I]; sli++) {  if (\*sli == 'z' || \*sli == 'k') {  m = mas[a[c]];  if (m == 'z' || (p == 0 && (m == 'ъ' || m == 'ь' || m == 'ы'))) { T = 0; break; }  \*sli = m;  if (sli - sl > 0) {  char c0 = sl[sli - sl];  char c1 = sl[sli - sl - 1];  if (c0 == -72)c = 0;  if (c1 == -72)c1 = 0;  if ((&root\_stat)->children[c0 + 32]->parent\_ss[c1 + 32] == NULL) {  break;  }  }  using\_letters[c] = m;  c++;  }  p++;  }  if (T != 0) {  current2 = Search(sl);  if (current2 != NULL && current2->end\_of\_word == true) {  masi = massa\_sl(number, I, J, sl);  PutIn1(st, number, I, J, masi, using\_letters, sl, num\_n\_z);  DOP -= 1;  if (ukaz % 2 == 1 && DOP == 0) {  printf("DOP 2: %d\n", DOP);  return;  }  }  }  for (int i = 0; i < 7; i++) { using\_letters[i] = NULL; }  while (NextSet(7, num\_z)) {  c = 0;  T = 1;  strcpy(sl, sl2);  for (char\* sli = &sl[0]; sli <= &sl[J - I]; sli++) {  if (\*sli == 'z' || \*sli == 'k') {  m = mas[a[c]];  \*sli = m;  if (sli - sl > 0) {  char c0 = sl[sli - sl];  char c1 = sl[sli - sl - 1];  if (c0 == -72)c = 0;  if (c1 == -72)c1 = 0;  if ((&root\_stat)->children[c0 + 32]->parent\_ss[c1 + 32] == NULL) {  break;  }  }  using\_letters[c] = m;  c++;  }  p++;  }  if (T != 0) {  current2 = Search(sl);  if (current2 != NULL && current2->end\_of\_word == true) {  masi = massa\_sl(number, I, J, sl);  PutIn1(st, number, I, J, masi, using\_letters, sl, num\_n\_z);  DOP -= 1;  if (ukaz % 2 == 1 && DOP == 0) {  printf("DOP 2: %d\n", DOP);  return;  }  }  }  memset(using\_letters, NULL, 7);  }  } |

Таблица 2. Результат оптимизации алгоритма.

Таким образом, убрана рекурсия и существенно сокращена время работы программы.

1. **Результаты и выводы**

По итогам проведенных оптимизаций, среднее время хода существенно сократилось для обоих игроков. Можно сделать вывод, что оптимизация на уровне алгоритма была проведена успешно.

1. **ОПТИМИЗАЦИЯ НА МАШИННО-НЕЗАВИСИМОМ УРОВНЕ**
2. **Отказ от цикла**

Цикл для заполнения поля игры пустыми клетками также был изменен. Функция memset работает быстрее, за счет чего генерация происходит за меньшее время.

|  |  |
| --- | --- |
| Исходный код | Измененный код |
| for (int i = 0; i < 15; i++) {  for (int j = 0; j < 15; j++) {  A[i][j] = 'z';  } | memset(A, 'z', sizeof A); |

Таблица 3. Результат оптимизации алгоритма.

1. **Вынос инвариантного кода**

В начальном коде в функции int analise\_2() одно и то же условие **if** проверялось многократно в цикле, на что затрачивалось большое количество времени. В новом коде это условие было вынесено, поскольку выражение истинно лишь при единственном значении i.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| for (int i = 0; i < 15; i++) {  if (mas[i] == 'z' && i == 0 && mas[i + 1] != 'z') {  mas[i] = 'k';  point\_number[count] = 0;  count += 1;  }  else if (mas[i] == 'z' && i != 0 && i != 14 && ((mas[i - 1] != 'z' && mas[i - 1] != 'k') || (mas[i + 1] != 'z' && mas[i + 1] != 'k'))) {  mas[i] = 'k';  if (mas[i - 1] != 'z') {  point\_number[count] = i;  count += 1;  }  else {  x = i - 1;  while (mas[x] == 'z') {  k += 1;  x -= 1;  }  for (int j = 0; j < k; j++) {  point\_number[count] = i;  }  k = 1;  count += 1;  }  }  else if (mas[i] == 'z' && i == 14 && ((mas[i - 1] != 'z' && mas[i - 1] != 'k'))) {  mas[i] = 'k';  if (mas[i - 1] != 'z') {  point\_number[count] = i;  count += 1;  }  }  } | if (mas[0] == 'z' && mas[1] != 'z') {  mas[0] = 'k';  point\_number[count] = 0;  count ++;  }  for (short int i = 1; i < 14; i++) {  if (mas[i] == 'z' && ((mas[i - 1] != 'z' && mas[i - 1] != 'k') || (mas[i + 1] != 'z' && mas[i + 1] != 'k'))) {  mas[i] = 'k';  if (mas[i - 1] != 'z') {  point\_number[count] = i;  count ++;  }  else {  x = i - 1;  while (mas[x] == 'z') {  k ++;  x --;  }  for (int j = 0; j < k; j++) {  point\_number[count] = i;  }  k = 1;  count ++;  }  }  }  if (mas[14] == 'z' && ((mas[13] != 'z' && mas[13] != 'k'))) {  mas[14] = 'k';  if (mas[13] != 'z') {  point\_number[count] = 14;  count ++;  }  } |

Таблица 4. Результат оптимизации алгоритма.

1. **Оптимизация переходов**

Была применена оптимизация переходов. В функции int pirog() до модификации происходила проверка каждого условия, несмотря на то что они являются взаимоисключающими. Поэтому многократное повторение if было заменено на конструкцию **if – else if**.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| for (int i = I; i <= J; i++) {  if (num != 0 && A[num - 1][i] != 'z' && t1 == 1) {  return 0;  }  if (num != 0 && A[num - 1][i] != 'z' && t1 == 0) {  if (A[num][i] != 'z') {  t1 = 1;  }  else {  return 0;  }  }  if (num != 0 && A[num - 1][i] == 'z') {  t1 = 0;  }  if (num != 14 && A[num + 1][i] != 'z' && t2 == 1) {  return 0;  }  if (num != 14 && A[num + 1][i] != 'z' && t2 == 0) {  if (A[num][i] != 'z') {  t2 = 1;  }  else {  return 0;  }  }  if (num != 14 && A[num + 1][i] == 'z') {  t2 = 0;  }  if (A[num][i] != 'z') {  t3 += 1;  }  }  if (t3 != num\_n\_z) { return 0; }  return 1;  } | for (int i = I; i <= J; i++) {  if (num != 0 && A[num - 1][i] != 'z' && t1 == 1) {  return 0;  }  else if (num != 0 && A[num - 1][i] != 'z' && t1 == 0) {  if (A[num][i] != 'z') {  t1 = 1;  }  else {  return 0;  }  }  else if (num != 0 && A[num - 1][i] == 'z') {  t1 = 0;  }  if (num != 14 && A[num + 1][i] != 'z' && t2 == 1) {  return 0;  }  else if (num != 14 && A[num + 1][i] != 'z' && t2 == 0) {  if (A[num][i] != 'z') {  t2 = 1;  }  else {  return 0;  }  }  else if (num != 14 && A[num + 1][i] == 'z') {  t2 = 0;  }  if (A[num][i] != 'z') {  t3 ++;  }  }  if (t3 != num\_z) { return 0; }  return 1;  } |

Таблица 5. Результат применения метода машинно-независимой оптимизации.

1. **Удаление недостижимого кода.**

В процессе оптимизации выявлено, что функции int sum\_ost и void cleaning\_mas() не играют никакой роли в программ и нигде не вызываются. Так же были удалены переменные, которые теперь не используются из-за изменения алгоритма. Таким образом код был почищен и стал более читаемый.

1. **Развертка циклов.**

Развертка цикла была применена в функциях int searching(), int analise\_2, void giving\_letters, void change(), void PutIn() для уменьшения количества итераций.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| int searching(char let[8], char mas\_sl[8])  char comp\_dop[8];  strcpy(comp\_dop, mas\_sl);  int t = 0;  for (int i = 0; i < strlen(let); i++) {  t = 0;  for (int j = 0; j < 7; j++) {  if (let[i] == mas\_sl[j]) {  mas\_sl[j] = 'z';  t = 1;  break;  }  }  if (t != 1) {  strcpy(mas\_sl, comp\_dop);  return 0;  }  }  return 1;  } | int searching(char let[8], char mas\_sl[8])  char comp\_dop[8];  strcpy(comp\_dop, mas\_sl);  for (int i = 0; i < strlen(let); i++) {  if (let[i] == mas\_sl[0]) {  mas\_sl[0] = 'z';  break;  }  if (let[i] == mas\_sl[1]) {  mas\_sl[1] = 'z';  break;  }  if (let[i] == mas\_sl[2]) {  mas\_sl[2] = 'z';  break;  }  if (let[i] == mas\_sl[3]) {  mas\_sl[3] = 'z';  break;  }  if (let[i] == mas\_sl[4]) {  mas\_sl[4] = 'z';  break;  }  if (let[i] == mas\_sl[5]) {  mas\_sl[5] = 'z';  break;  }  if (let[i] == mas\_sl[6]) {  mas\_sl[6] = 'z'  break;  }  strcpy(mas\_sl, comp\_dop);  return 0;  }  return 1;  } |

Таблица 6.Пример применения метода машинно-независимой оптимизации.

1. **ОПИТИМИЗАЦИЯ НА МАШИННО-ЗАВИСИМОМ УРОВНЕ**
2. **Замена индекса на указатель**

При использовании индексов для доступа к элементам массива компилятор должен выполнить дополнительные операции по вычислению адреса каждого элемента. При использовании указателей эти операции не требуются, что позволяет сократить время доступа к элементам массива внутри циклов, а оно имеет большое значение для производительности программы.

Таким образом, была применена замена индексов на указатели в функциях int proverka\_slova() и int analise\_2() и значительно ускорена работа программы.

|  |  |
| --- | --- |
| Исходный код | Изменённый код |
| for (short int j = 0; j < 7; j++) {  for (short int i = uk; i <= J - I; i++) {  if (mas[j] != 'z' && (sl[i] == 'z' || sl[i] == 'k')) {  sl[i] = mas[j];  if ((i == 0) && (mas[j] == 'ь' || mas[j] == 'ъ' || mas[j] == 'ы')) {  break;  }  t = mas[j];  using\_letters[letter] = t;  mas[j] = 'z'; | for (char\* masj = &mas[0]; masj < &mas[7]; masj++) {  int i = uk;  for (char\* sli = &sl[uk]; sli <= &sl[J - I]; sli++) {  if (\*masj != 'z' && (\*sli == 'z' || \*sli == 'k')) {  if ((i == 0) && (\*masj == 'ь' || \*masj == 'ъ' || \*masj == 'ы')) {  break;  }  \*sli = \*masj;  t = \*masj;  using\_letters[letter] = t;  \*masj = 'z'; |

Таблица 7. Результат применения метода машинно-зависимой оптимизации.

1. **Замена типа int на char**

Так как все переменные в нашей программе принимают значения по модулю меньше 32 тысяч было принято решение заменить тип переменных int (4 байта) на char(1 байт) везде, где это возможно. В результате тип данных был заменён более чем у 60 переменных и функций.

1. **Оптимизация с помощью внутренних возможностей VS**

В среде разработки Visual Studio 2019 в свойствах проекта была включена оптимизация кода по скорости /О2 и оптимизация всей программы

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2. Раздел Оптимизация

В разделе Создание кода были включены такие настройки, как Выравнивание членов структур 16 байт (/Zp16), Создание параллельного кода (/Qpar), компоновка на уровне функций.

Изображение выглядит как текст, снимок экрана, компьютер, внутренний

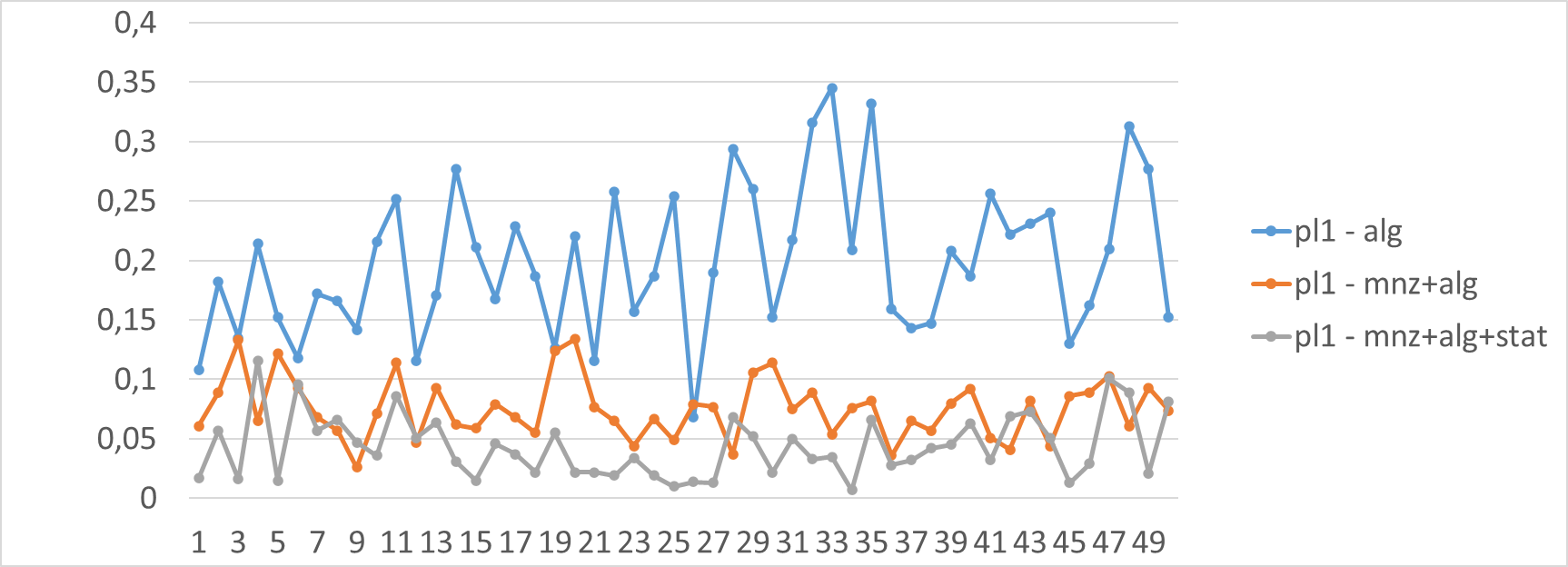
Автоматически созданное описание

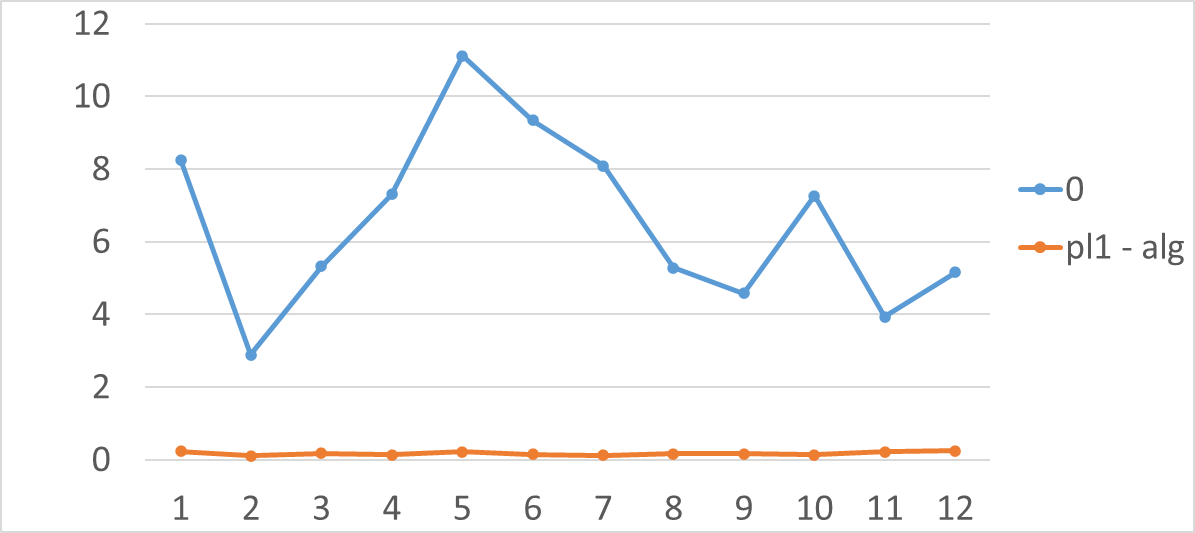
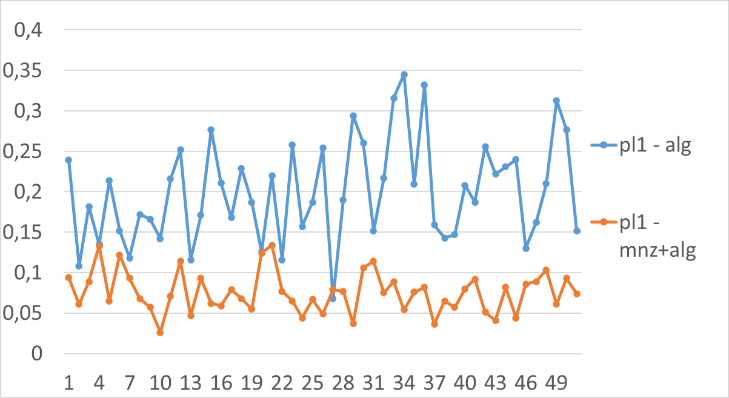
Рисунок 3. Раздел Создание кода

1. **ТЕСТИРОВАНИЕ И РЕЗУЛЬТАТЫ ПРОГРАММЫ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Без оптимизаций, с | Оптимизация на уровне алгоритма, c | Машинно-независимая оптимизация, c | Машинно-зависимая оптимизация, c |
| Легкий уровень | 6,293 | 0,190 | 0,075 | 0,038 |
| Сложный уровень | 15,251 | 0,284 | 0,149 | 0,097 |
| K, % | - | 43,409 | 2,220 | 1,005 |

Таблица 8.Сравнительная таблица среднего времени хода игрока от типа оптимизации. На нижней строке таблицы представлены числа, во сколько раз уменьшилось время работы программы в процентах.



Рисунки 2 - 4. Сравнительные графики времени выполнения программы.

На первом графике представлено среднее время работы программы с 1) оптимизаций алгоритма, 2) добавленной машинно-независимой и 3) статистическим анализом из выборки 50 игр для каждого вида оптимизаций. На втором та же выборка, и, очевидно, оптимизация алгоритма на алгоритмическом уровне значительно уменьшила время работы программы. На третьем графике можно увидеть, что машинно-независимая оптимизация также улучшает программу, но не так эффективно, как алгоритм.

По результатам проведенных оптимизаций выполнение алгоритма было значительно ускорено. Это можно заметить по коэффициентам, которые получились в таблице выше. В результате, очевидно, что оптимизация на уровне алгоритма самая эффективная.

**ЗАКЛЮЧЕНИЕ**

В ходе проделанной работы был оптимизирован алгоритм игры «Эрудит», разработанного в ходе курсовой работы прошлого семестра. До оптимизаций алгоритм работал достаточно долго, ход 1 игрока по времени мог достигать 250 секунд, выполняя много лишних действий.

Были проведены оптимизации на уровне алгоритма, на машинно-независимом и машинно-зависимом уровнях, составлена таблица зависимости времени хода игроков от типа оптимизации, что позволило прийти к выводу, что наиболее значительной оптимизацией стала оптимизация на уровне алгоритма и машинно-независимая оптимизация. Это естественно, ведь машинно-зависимые оптимизации не дают таких результатов, как работа с циклами, условиями и функциями. Оптимизация проекта повысила скорость работы программы с 10,772 миллисекунд до 0,07. Таким образом, оптимизация играет важную роль при разработке любого проекта.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ (ИСТОЧНИКОВ)

1. **Tim A. Wheeler, Mykel J. Kochenderfer** Algorithms for Optimization, 2019. -520 c
2. **Marcello La Rocca** Advanced Algorithms and Data Structures, 2021. - 380 с.
3. Проблемы кибернетики. Выпуск 9, 1963 год. Г.Г. Белоногов, Г.Д. Фролов. Эмпирические данные о распределении букв в русской письменной речи - с. 290
4. <https://www.algotree.org/algorithms/trie/>
5. <https://prog-cpp.ru/placement/?ysclid=lrdfa3ri4l638520619%27nj>