universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

André Cardoso 50%
andremacardoso@ua.pt
108269

Tiago Figueiredo 50%
tiago.a.figueiredo@ua.pt
107263

09 de Janeiro de 2023

Conteúdo

1 Introdução

Uma word ladder é uma sequência de palavras em que cada palavra difere em uma e só uma letra da palavra anterior. Por exemplo, na língua Portuguesa é possível ir da palavra tudo para a palavra nada em quatro passos. *tudo* → *todo* → *nodo* → *nado* → *nada*. Como tal para resolver o problema proposto de criar um algoritmo em *C/C++* que permita encontrar foi feita uma implementação de uma *Hash Table* em *C++*, usada depois para permitir a implementação de grafos e do *union find*, tornando possível tal algoritmo.

2 Funções Usadas

Esta secção contém uma lista, com a respetiva descrição de todas as funções usadas para a criação do algoritmo.

2.1 Hash Table

Sendo a linguagem de progamação escolhida para a resolução de este problema *C++*, a *Hash table* foi implementada através de duas classes, uma que contém os parâmetros de cada nó da *Hash Table*, e uma que contém a implementação da *Hash Table*.

2.1.1 Constructor

O construtor inicializa todas as variáveis da classe quando um objeto do tipo da classe é criado. Começa por definir o tamanho da hash table como sendo 65536, cria um array de ponteiros chamado *words* com o tamanho da tabela. Inicializa também a variável *entries* a 0, a variável *connected_components* também a 0 e a *variável load_factor* como sendo 0,75. Define, por fim, todos os elementos do array *words* como sendo ponteiros nullos, o que indica que todos os elementos da tabela estão vazios.

2.1.2 Destructor

O destrutor dá um loop pelo array *words* eliminando cada elemento não nulo, acabando por eliminar o array, no final.

2.1.3 add

Esta função é usada para adicionar uma nova palavra à hash table. A função recebe um único argumento, uma string chamada *word* que representa a palavra a ser adicionada à tabela.

A função começa chamando a função hash() para obter o índice em que a palavra deve ser inserida. Em seguida, verifica se a palavra já está presente na tabela verificando se o elemento no índice calculado não é nulo e se a

palavra armazenada nesse elemento é a mesma da palavra a ser adicionada. Se a palavra já estiver presente, a função retorna sem adicionar a palavra à tabela. A função, caso a palavra não corresponder a nenhuma já existente, verifica se o número de entradas na tabela mais 1 é maior ou igual ao tamanho da tabela multiplicado pelo fator de carga. Se isso for verdade, significa que a tabela está a ficar cheia e é chamada a função resize() para aumentar o tamanho da tabela. Se o elemento no índice calculado for nulo, a função chama a função create() para criar um novo nó com a palavra e a inserir no índice caso contrário, se a palavra armazenada nesse elemento não for a mesma da palavra a ser adicionada, a função entra em um loop while. O loop executa até encontrar um espaço vazio na tabela podendo também parar se a palavra já se encontrar na tabela. Para encontrar o próximo espaço disponível, ele usa *linear probing*. Linear probing é uma técnica de resolução de colisões onde o próximo espaço é encontrado incrementando o índice um por um até ser encontrada uma posição vazia. Se um espaço vazio for encontrado, a função chama a função create(), finalmente criando um novo nó com a palavra e inserindo-a no índice.

2.1.4 get

Esta função é utilizada para obter um nó específico dado uma palavra. Ela começa chamando a função *hash* para obter o índice onde a palavra deve ser encontrada. Se o elemento no índice calculado for nulo, a função retorna um *nullptr*, indicando que a palavra não está presente. Se o elemento não é nulo, ele verifica se a palavra armazenada é a mesma da palavra que se deseja procurar. Se for, a função retorna o endereço desse elemento, caso contrário a função entra num loop while usando linear probing, como na função de cima, para encontrar a palavra ou um espaço vazio. A função retorna o endereço do elemento que coném a palavra, caso este seja encontrado ou *nullptr*.

2.1.5 add_edge

Esta função é utilizada para adicionar uma aresta entre dois nós na estrutura de dados que representa um grafo. Ela recebe dois argumentos, "from"e "to", que são ponteiros para os nós entre os quais a aresta será adicionada.

A função adiciona o nó "to"à lista de adjacência do nó "from"e o nó "from"à lista de adjacência do nó "to", estabelecendo assim a conexão entre eles. A função incrementa o contador de arestas em ambos os nós, indicando que eles têm uma aresta adicional. A função chama outra função chamada "g_union"passando "from"e "to"como argumentos, essa função fará algo relacionado com a estrutura de dados union-find.

2.1.6 BFS

A função BFS é uma implementação de breadth-first search. Ela começa por percorrer todos os nós e marcando-os como não visitados e sem pais (senão incorre o risco de não funcionar caso outra função que modifique os campos visited e parent já tenha sido executada). Em seguida, a função adiciona o nó "from"a uma fila e marca-o como visitado. A função então entra em um loop enquanto a fila não estiver vazia. Dentro do loop, a função pega o primeiro elemento da fila e verifica se é o nó "to". Se for, a função retorna a profundidade atual. Caso contrário, a função percorre todos os nós adjacentes ao nó atual que ainda não foram visitados, marca-os como visitados e adiciona-os à fila. A profundidade é incrementada a cada iteração do loop. Se a profundidade atual é maior que o valor máximo de profundidade especificado e o valor máximo de profundidade é diferente de 0, a função retorna -1. Se o loop termina e o nó "to"ainda não foi encontrado, a função retorna -1.

2.1.7 DFS

A função DFS é uma implementação de depth-first search. Ela funciona de maneira semelhante à função BFS, mas usa uma pilha em vez de uma fila. A função marca todos os nós como não visitados e sem pais e adiciona o nó "from"à pilha. A função então entra em um loop enquanto a pilha não estiver vazia. Dentro do loop, a função pega o topo da pilha e verifica se é o nó "to". Se for, a função segue o caminho de volta ao nó "from"contando a profundidade e retorna-a. Caso contrário, a função percorre todos os nós adjacentes ao nó atual que ainda não foram visitados, marca-os como visitados e adiciona-os à pilha. Se o loop termina e o nó "to"ainda não foi encontrado, a função retorna -1.

2.1.8 list_connected_components

A função list_connected_components é usada para listar todos os nós que fazem parte do mesmo componente conectado de um nó específico dado uma palavra. Ela chama a função get() para obter o nó correspondente à palavra fornecida, se ele não for encontrado, a função imprime "Palavra não encontrada"e retorna. Caso contrário, ela chama outra função chamada "find", passando o nó encontrado como argumento, essa função retorna um representante de um conjunto na estrutura de dados union-find. Em seguida, a função percorre todos os nós na tabela hash e adiciona a um vetor de componentes todos os nós cujo representante é o mesmo do nó encontrado anteriormente. Finalmente, a função imprime "Pertencente ao mesmo componente conectado como [palavra fornecida]:"e imprime todas as palavras armazenadas nos nós do vetor de componentes.

2.1.9 find

2.1.10 g_union

2.1.11 print_adjacency_list

2.1.12 hash

2.1.13 unhash

2.2 Estatísticas da Hash Table

2.2.1 get_load_factor e get_collisions

As funções get_load_factor e get_collisions são usadas para obter estatísticas sobre a tabela hash. A primeira função retorna o fator de carga atual, que é o número de entradas na tabela dividido pelo tamanho da tabela. A segunda função retorna o número de colisões na tabela, que é o número de entradas que estão em uma posição diferente daquela calculada pela função de hash.

2.2.2 get_distribution

A função get_distribution é usada para obter a distribuição de entradas na tabela hash. Ela percorre toda a tabela e adiciona "true"a um vetor de bools, que permite com alguns compiladores guardar o valor em um bit, se uma entrada está presente na posição atual ou "false"se não estiver. A função retorna esse vetor.

2.3 Estatísticas do Grafo

2.3.1 get_connected_components

A função get_connected_components conta o número de componentes conectados no grafo. Ele percorre a tabela hash e verifica, para cada nó, se é um representante de seu conjunto de união (usando a propriedade representative). Se for, incrementa o contador de componentes. O número de componentes conectados é retornado no final.

2.3.2 get_diameter

A função get_diameter calcula o diâmetro do componente conexo. O diâmetro é o caminho maior caminho mais curto entre dois nós do mesmo componente conexo. Começa por percorrer a hash table e chama a função DFS em cada nó, passando o nó atual como início e o nó especificado como parâmetro como destino. A distância retornada pela DFS é comparada com a distância máxima encontrada até agora. Se for maior, atualiza a distância máxima e salva o nó inicial. No final, se print for verdadeiro, imprime o diâmetro e o caminho encontrado.

2.3.3 get_diameter_node

A função get_diameter_node é semelhante à anterior, mas apenas retorna o nó inicial do caminho com o maior diâmetro.

2.4 Outras Funções

2.4.1 longest

A função "longest"encontra o diâmetro do componente conexo da palavra dada. Ela faz isso chamando o método get_diameter() da tabela de hash correspondente ao tamanho da palavra dada.

2.4.2 connected

A função "connected"verifica se as duas palavras dadas são conectadas (se elas diferem apenas por uma letra). Ela faz isso comparando cada letra das palavras uma a uma e verificando se elas são diferentes.

2.4.3 path_finder

A função "path_finder"é usada para encontrar o caminho mais curto entre duas palavras no grafo. Ele usa a função BFS para encontrar a distância mais curta entre as duas palavras e imprime o caminho encontrado.

2.4.4 connected_components

A função "connected_components"encontra todas as palavras que estão conectadas (que diferem apenas por uma letra) à palavra dada. Ela faz isso chamando o método list_connected_components() da tabela de hash correspondente ao tamanho da palavra dada.

3 Resultados

3.1 Resultados gerais

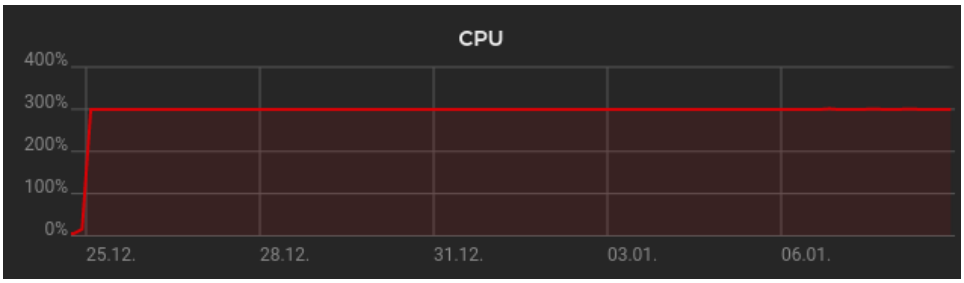
3.2 Testes de Memory Leaks

Para determinar se o programa tinha *memory leaks*, foi usado o valgrind, com as seguintes opções:
–leak-check=full
–show-leak-kinds=all
–track-origins=yes
–verbose
–log-file=valgrind-out.txt
./word_ladder

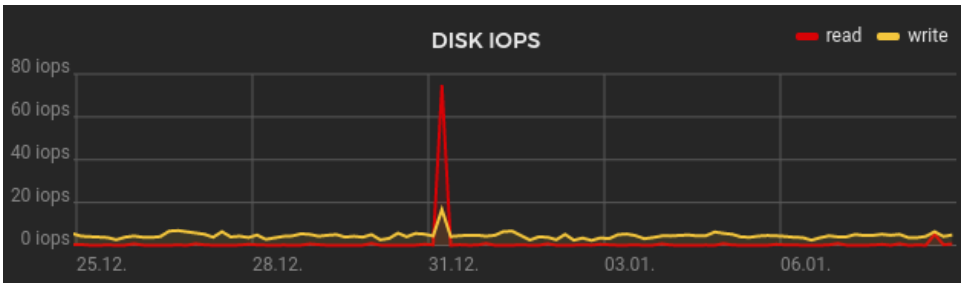
Tendo sido obtido o ficheiro de log encontrado no apêndice *valgrind-out.txt*, comprovando que não existem *memory leaks*.

3.3 Resultados ao fim de 14 dias

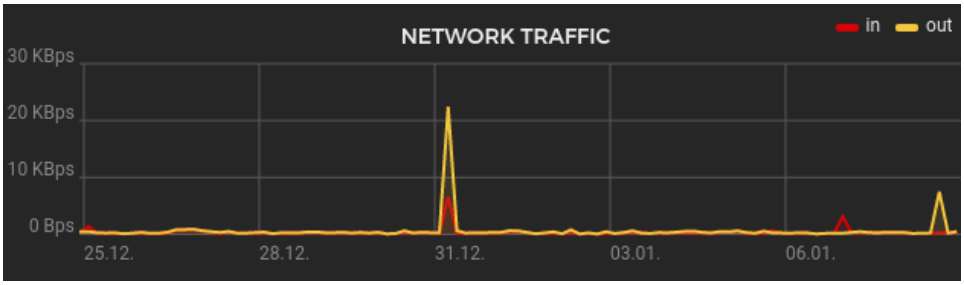
Devido ao facto de que a lógica principal do programa foi acabada com alguma antecedência, o algoritmo correu com o objetivo de encontrar a maior *ladder* possível, para o dicionário fornecido, para cada tamanho de palavra. Infelizmente devido à quantidade de palavras de tamanho superior a oito ou nove letras e inferior a vinte letras, algumas das *ladders* para os comprimentos de palavra nesse intervalo não foram encontrados, tendo a maior sido uma *word ladder* com tamanho 1844 para palavras de tamanho sete.



Devido ao uso de todos os threads disponiveis, três núcleos e 6 threads, de um processador AMD^{TM} *Epyc Rome*, o uso do processador manteve-se a 300% (cada núcleo a 100%), durante a inteira duração.



O programa também demonstrou um uso residual do disco durante a maioria da sua duração, sendo o pico visivel no gráfico, resultante de um pico no uso de rede enquanto este atualizava, visto que havia mais tarefas a correr no mesmo servidor.



4 Referências

[Val] *How do I use valgrind to find memory leaks?* 2011. URL: https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks.

[Ref20] C Reference. *C Reference*. 2020. URL: https://en.cppreference.com/w/. (accessed: 22.12.2022).

[Sil22] Tomás Oliveira e Silva. *Lecture Notes*. 2022. URL: elearning.ua.pt. (accessed: 22.12.2022).

[Fow] *Fowler-Noll-Vo hash function*. URL: https://en.wikipedia.org/wiki/FowlerNollVo_hash_function.

[Wel] Christopher Wellons. *skeeto/hash-prospector*. URL: https://github.com/skeeto/hash-prospector#three-round-functions.

5 Apêndice

5.1 word_ladder.cpp

```cpp
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <cmath>
#include <vector>
#include <queue>
#include <stack>
#include <thread>

using namespace std;
#define _max_word_size_ 32

class node {
public:
    node(const string &word) : word(word) {
        parent = nullptr;
        visited = false;
        representative = this;
        vertices = 1;
        edges = 0;
    }

    string word;
    // search relevant data
    node *parent;
    bool visited;
    // graph data structure
    vector<node *> adjacency_list;
    // union-find data structure
    node *representative;
    int vertices;
    int edges;
};

class hashTable {
public:
    unsigned int size;
    node **words;
    unsigned int entries;
    int connected_components;
    double load_factor;

    hashTable() {
        // Makes the dict only need to be resized once.
```

```
46          size = 65536;
47          words = new node *[size];
48          entries = 0;
49          connected_components = 0;
50          load_factor = 0.75;
51          for (unsigned int i = 0; i < size; i++) {
52              words[i] = nullptr;
53          }
54      }
55
56      ~hashTable() {
57          for (unsigned int i = 0; i < size; i++) {
58              if (words[i] != nullptr) {
59                  delete words[i];
60              }
61          }
62          delete[] words;
63      }
64
65      void add(const string &word) {
66          unsigned int index = hash(word);
67          if (words[index] != nullptr && words[index]->word == word) {
68              return;
69          }
70          if (entries + 1 >= size * load_factor) {
71              resize();
72          }
73          if (words[index] == nullptr) {
74              create(index, word);
75          } else if (words[index]->word != word) {
76              while (words[index] != nullptr && words[index]->word !=
                    word) {
77                  // Linear probing is the fastest way.
78                  // Probably because it uses the cache more
                        efficiently.
79                  // And that matters the most when the table is huge
                        and we have memory to spare.
80                  index = (index + 1) % size;
81              }
82              create(index, word);
83          }
84      }
85
86      node *get(const string &word) {
87          unsigned int index = hash(word);
88          if (words[index] == nullptr) {
89              return nullptr;
90          }
91          if (words[index]->word == word) {
```

```
92              return words[index];
93          }
94          while (words[index] != nullptr && words[index]->word !=
                word) {
95              index = (index + 1) % size;
96          }
97          return words[index];
98      }
99
100     // graph functions
101     void add_edge(node *from, node *to) {
102         from->adjacency_list.push_back(to);
103         to->adjacency_list.push_back(from);
104         from->edges++;
105         to->edges++;
106         g_union(from, to);
107     }
108
109     int BFS(node *from, node *to, int maximum_depth = 0) {
110         for (unsigned int i = 0; i < size; i++) {
111             if (words[i] != nullptr) {
112                 words[i]->visited = false;
113                 words[i]->parent = nullptr;
114             }
115         }
116         queue < node * > q;
117         from->visited = true;
118         q.push(from);
119         int depth = 0;
120         while (!q.empty()) {
121             int q_size = q.size();
122             for (int i = 0; i < q_size; i++) {
123                 node *current = q.front();
124                 q.pop();
125                 if (current == to) {
126                     return depth;
127                 }
128                 for (size_t j = 0; j <
                        current->adjacency_list.size(); j++) {
129                     node *adjacent = current->adjacency_list[j];
130                     if (!adjacent->visited) {
131                         adjacent->visited = true;
132                         adjacent->parent = current;
133                         q.push(adjacent);
134                     }
135                 }
136             }
137             depth++;
138             if (depth > maximum_depth && maximum_depth != 0) {
```

```
139                 return -1;
140             }
141         }
142         return -1;
143     }
144
145     int DFS(node *from, node *to) {
146         for (unsigned int i = 0; i < size; i++) {
147             if (words[i] != nullptr) {
148                 words[i]->visited = false;
149                 words[i]->parent = nullptr;
150             }
151         }
152         stack < node * > q;
153         from->visited = true;
154         q.push(from);
155         int depth = 0;
156         while (!q.empty()) {
157             int q_size = q.size();
158             for (int i = 0; i < q_size; i++) {
159                 node *current = q.top();
160                 q.pop();
161                 if (current == to) {
162                     // god why
163                     while (current->parent != nullptr && current !=
                            from) {
164                         current = current->parent;
165                         depth++;
166                     }
167                     return depth;
168                 }
169                 for (size_t j = 0; j <
                        current->adjacency_list.size(); j++) {
170                     node *adjacent = current->adjacency_list[j];
171                     if (!adjacent->visited) {
172                         adjacent->visited = true;
173                         adjacent->parent = current;
174                         q.push(adjacent);
175                     }
176                 }
177             }
178         }
179         return -1;
180     }
181
182     void list_connected_components(const string &word) {
183         vector < node * > components;
184         node *vertex = get(word);
185         if (vertex == nullptr) {
```

```
186             cout << "Word not found" << endl;
187             return;
188         }
189         node *representative = find(vertex);
190         for (unsigned int i = 0; i < size; i++) {
191             if (words[i] != nullptr && find(words[i]) ==
                    representative) {
192                 components.push_back(words[i]);
193             }
194         }
195         cout << "Belonging to same connected component as " << word
                << "are:" << endl;
196         for (size_t i = 0; i < components.size(); i++) {
197             cout << components[i]->word << "\n";
198         }
199     }
200
201     // hash table statistics
202     double get_load_factor() {
203         return (double) entries / size;
204     }
205
206     int get_collisions() {
207         unsigned int collisions = 0;
208         for (unsigned int i = 0; i < size; i++) {
209             if (words[i] != nullptr) {
210                 if (hash(words[i]->word) != i) {
211                     collisions++;
212                 }
213             }
214         }
215         return collisions;
216     }
217
218     vector<bool> get_distribution() {
219         vector<bool> distribution;
220         for (unsigned int i = 0; i < size; i++) {
221             if (words[i] != nullptr) {
222                 distribution.push_back(true);
223             } else {
224                 distribution.push_back(false);
225             }
226         }
227         return distribution;
228     }
229
230     // graph statistics
231     int get_connected_components() {
232         int components = 0;
```

```
233         for (unsigned int i = 0; i < size; i++) {
234             if (words[i] != nullptr) {
235                 if (words[i]->representative == words[i]) {
236                     components++;
237                 }
238             }
239         }
240         return components;
241     }
242 
243     int get_diameter(node *n, bool print = true) {
244         int diameter = 0;
245         node *max = nullptr;
246         for (unsigned int i = 0; i < size; i++) {
247             if (words[i] != nullptr) {
248                 if (words[i]->adjacency_list.size() == 0) {
249                     continue;
250                 }
251                 int distance = DFS(words[i], n);
252                 if (distance > diameter) {
253                     diameter = distance;
254                     max = words[i];
255                 }
256             }
257         }
258         // DFS data is wiped out every run.
259         DFS(n, max);
260         node *res = max;
261         if (res == nullptr) {
262             return 0;
263         }
264         if (print) {
265             cout << "Diameter: " << diameter << endl;
266             cout << "Path: ";
267             if (res == nullptr) {
268                 cout << "No connected words." << endl;
269             }
270             while (res->parent != nullptr) {
271                 cout << res->word << " -> ";
272                 res = res->parent;
273             }
274             cout << res->word << endl;
275         }
276 
277         return diameter;
278     }
279 
280     node *get_diameter_node(node *n) {
281         int diameter = 0;
```

```
282         node *max = nullptr;
283         for (unsigned int i = 0; i < size; i++) {
284             if (words[i] != nullptr) {
285                 int distance = DFS(words[i], n);
286                 if (distance > diameter) {
287                     diameter = distance;
288                     max = words[i];
289                 }
290             }
291         }
292         return max;
293     }
294 
295 private:
296     void create(int index, const string &word) {
297         entries++;
298         connected_components++;
299         words[index] = new node(word);
300     }
301 
302     void resize() {
303         // High resize coefficient to reduce resizes, which are
                expensive.
304         int coeff = 4;
305         size *= coeff;
306         node **new_words = new node *[size];
307         for (unsigned int i = 0; i < size; i++) {
308             new_words[i] = nullptr;
309         }
310         for (unsigned int i = 0; i < size / coeff; i++) {
311             if (words[i] != nullptr) {
312                 int index = hash(words[i]->word);
313                 if (new_words[index] == nullptr) {
314                     new_words[index] = words[i];
315                 } else {
316                     while (new_words[index] != nullptr) {
317                         index = (index + 1) % size;
318                     }
319                 }
320             }
321         }
322         delete[] words;
323         words = new_words;
324     }
325 
326     node *find(node *vertex) {
327         if (vertex->representative != vertex) {
328             vertex->representative = find(vertex->representative);
329         }
```

```
330         return vertex->representative;
331     }
332 
333     void g_union(node *from, node *to) {
334         node *from_rep = find(from);
335         node *to_rep = find(to);
336         if (from_rep != to_rep) {
337             to_rep->representative = from_rep;
338             connected_components--;
339         }
340     }
341 
342     void print_adjacency_list(node *n) {
343         cout << n->word << " -> ";
344         for (size_t i = 0; i < n->adjacency_list.size(); i++) {
345             cout << n->adjacency_list[i]->word << " ";
346         }
347         cout << endl;
348     }
349 
350 #define FNV_OFFSET 14695981039346656037UL
351 #define FNV_PRIME 1099511628211UL
352 
353     // Return 64-bit FNV-1a hash for key (NUL-terminated).
354     unsigned int hash(const string &word) {
355         uint64_t hash = FNV_OFFSET;
356         const char *key = word.c_str();
357         for (const char *p = key; *p; p++) {
358             hash ^= (uint64_t)(unsigned char)(*p);
359             hash *= FNV_PRIME;
360         }
361         // Ensure hash is adjusted to the size of the table.
362         return (size_t)(hash & (uint64_t)(size - 1));
363     }
364 
365     //
        https://github.com/skeeto/hash-prospector#three-round-functions
366     // Kept for reference.
367     unsigned int hash(int x) {
368         x ^= x >> 17;
369         x *= 0xed5ad4bb;
370         x ^= x >> 11;
371         x *= 0xac4c1b51;
372         x ^= x >> 15;
373         x *= 0x31848bab;
374         x ^= x >> 14;
375         return x;
376     }
377 
```

```
378     unsigned int unhash(int x) {
379         x ^= x >> 14 ^ x >> 28;
380         x *= 0x32b21703;
381         x ^= x >> 15 ^ x >> 30;
382         x *= 0x469e0db1;
383         x ^= x >> 11 ^ x >> 22;
384         x *= 0x79a85073;
385         x ^= x >> 17;
386         return x;
387     }
388 };
389 
390 void longest(hashTable **dicts, const string &word) {
391     hashTable *dict = dicts[word.size() - 1];
392     node *n = dict->get(word);
393     cout << "Longest path to " << word << " is " << endl
394          << dict->get_diameter(n)
395          << " words long." << endl;
396     return;
397 }
398 
399 bool connected(const string &a, const string &b) {
400     if (a.size() != b.size())
401         return false;
402     bool result = false;
403     for (size_t i = 0; i < a.size(); i++) {
404         if (a[i] != b[i]) {
405             // Only one difference is allowed
406             if (result)
407                 return false;
408             result = true;
409         }
410     }
411     return result;
412 }
413 
414 void path_finder(hashTable **dicts, const string &start, const
    string &end) {
415     if (start.size() != end.size()) {
416         cout << "Cannot compare different sizes." << endl;
417         return;
418     }
419     hashTable *dict = dicts[start.size() - 1];
420     cout << "Trying to go from " << start << " to " << end << endl;
421     node *from = dict->get(end);
422     node *to = dict->get(start);
423     if (from == nullptr || to == nullptr) {
424         cout << "No path found." << endl;
425         return;
```

```
426     }
427     int travelled = dict->BFS(from, to);
428     cout << "Travelled " << travelled << " nodes. " << endl;
429     node *res = to;
430     while (res->parent != nullptr) {
431         cout << res->word << " -> ";
432         res = res->parent;
433     }
434     cout << res->word << endl;
435 }
436 
437 void connected_components(hashTable **dicts, const string &word) {
438     hashTable *dict = dicts[word.size() - 1];
439     dict->list_connected_components(word);
440 }
441 
442 void end(hashTable **dicts) {
443 #if defined(_stats_) || defined(_detail_) || defined(_full_)
444     ofstream file;
445     file.open("stats.txt");
446 #endif
447     for (size_t i = 0; i < _max_word_size_; i++) {
448 #if defined(_stats_) || defined(_detail_) || defined(_full_)
449         file << endl;
450         file << "Hash Table for " << i + 1 << " letter words" <<
                endl;
451         file << "Size: " << dicts[i]->size << endl;
452         file << "Load factor: " << dicts[i]->get_load_factor() <<
                endl;
453         file << "Collisions: " << dicts[i]->get_collisions() << endl;
454 #if defined(_detail_) || defined(_full_)
455         vector<bool> distribution = dicts[i]->get_distribution();
456         file << "Distribution: " << endl;
457         for (size_t j = 0; j < distribution.size(); j++)
458         {
459             if (distribution[j])
460                 file << j << " ";
461         }
462         file << endl;
463 #endif
464 #endif
465         delete dicts[i];
466     }
467 #if defined(_stats_) || defined(_detail_) || defined(_full_)
468     file.close();
469 #endif
470 }
471 
472 void graph_builder(hashTable *dict) {
```

```
473     int sizes = 0;
474     // TODO: Optimize this, O(n^1.5) ish isn't good
475     for (size_t i = 0; i < dict->size; i++) {
476         node *from = dict->words[i];
477         if (from == nullptr)
478             continue;
479         if (sizes == 0)
480             sizes = from->word.size();
481         for (size_t j = i + 1; j < dict->size; j++) {
482             node *to = dict->words[j];
483             if (to == nullptr)
484                 continue;
485             if (connected(from->word, to->word)) {
486                 dict->add_edge(from, to);
487             }
488         }
489     }
490     if (sizes != 0)
491         cout << "Processed " << sizes + 1 << " letter words" << endl;
492 }
493
494 void longest_path(hashTable *dict) {
495     int largest = 0;
496     vector < node * > reprs;
497     node *max = nullptr;
498     for (unsigned int i = 0; i < dict->size; i++) {
499         if (dict->words[i] != nullptr) {
500             if (find(reprs.begin(), reprs.end(),
                    dict->words[i]->representative) == reprs.end()) {
501                 reprs.push_back(dict->words[i]->representative);
502                 int depth = dict->get_diameter(dict->words[i], false);
503                 if (depth > largest) {
504                     largest = depth;
505                     max = dict->words[i];
506                 }
507             }
508         }
509     }
510     node *origin = dict->get_diameter_node(max);
511     if (origin == nullptr || max == nullptr) {
512         cout << "No path found." << endl;
513         return;
514     }
515     dict->DFS(origin, max);
516     node *res = max;
517     ofstream file;
518     file.open("longest.txt", ios::app);
519     file << "Longest path for " << max->word.size() << " letter
            words" << endl;
```

```
520     file << "Size: " << largest << endl;
521     while (res->parent != nullptr) {
522         file << res->word << " -> ";
523         res = res->parent;
524     }
525     file << res->word << endl;
526 }
527 
528 int main() {
529     setlocale(LC_ALL, ".UTF8");
530     hashTable *dicts[_max_word_size_];
531     thread threads[_max_word_size_];
532     for (size_t i = 0; i < _max_word_size_; i++) {
533         dicts[i] = new hashTable;
534     }
535     ifstream in("wordlist-big-latest.txt");
536     if (!in) {
537         printf("Error: could not open words file\n");
538     }
539     string word;
540     while (in >> word) {
541         int size = word.size();
542         dicts[size - 1]->add(word);
543     }
544     for (int sizes = 0; sizes < _max_word_size_; sizes++) {
545         hashTable *dict = dicts[sizes];
546         threads[sizes] = thread(graph_builder, dict);
547     }
548     for (int sizes = 0; sizes < _max_word_size_; sizes++) {
549         threads[sizes].join();
550     }
551     path_finder(dicts, "etano", "sitie");
552 #ifdef _full_
553     ofstream file;
554     file.open("longest.txt", ios::trunc);
555     file.close();
556     for (int sizes = 0; sizes < _max_word_size_; sizes++)
557     {
558         hashTable *dict = dicts[sizes];
559         threads[sizes] = thread(longest_path, dict);
560     }
561     for (int sizes = 0; sizes < _max_word_size_; sizes++)
562     {
563         threads[sizes].join();
564     }
565 #endif
566     // connected_components(dicts, "belo");
567     longest(dicts, "etano");
568 
```

```
569     // TODO: See graphs
570     // TODO: Interesting diameters
571     // etano and sitia are opposite extremeties of (one of the)
            main connected component, as they show up in lots of
            diameters
572     end(dicts);
573 }
```

5.2 makefile

```
1  #
2  # makefile to compile the A.02 assignment (word ladder)
3  #
4
5  clean:
6         rm -rf a.out word_ladder *.exe
7
8  word_ladder: word_ladder.cpp
9         g++ -Wall -Wextra -O3 word_ladder.cpp -o word_ladder -lm -march=native
10
11 stats: word_ladder.cpp
12        g++ -Wall -Wextra -O3 word_ladder.cpp -o word_ladder -lm -march=native
              -D_stats_
13
14 detail: word_ladder.cpp
15        g++ -Wall -Wextra -O3 word_ladder.cpp -o word_ladder -lm -march=native
              -D_detail_
16
17 full: word_ladder.cpp
18        g++ -Wall -Wextra -O3 word_ladder.cpp -o word_ladder -lm -march=native
              -D_full_
19
20 debug: word_ladder.cpp
21        g++ -Wall -Wextra -O0 -ggdb3 word_ladder.cpp -o word_ladder -lm
              -march=native -D_full_
```