

André Cardoso 50%
andremacardoso@ua.pt
108269

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

09 de Janeiro de 2023

Conteúdo

1	Introdução	3
2	Funções Usadas	3
2.1	Hash Table	3
2.1.1	Constructor	3
2.1.2	Destructor	3
2.1.3	add	3
2.1.4	get	4
2.1.5	add_edge	4
2.1.6	BFS	5
2.1.7	DFS	5
2.1.8	list_connected_components	5
2.1.9	find	6
2.1.10	g_union	6
2.1.11	print_adjacency_list	6
2.1.12	hash	6
2.1.13	unhash	6
2.2	Estatísticas da Hash Table	6
2.2.1	get_load_factor e get_collisions	6
2.2.2	get_distribution	6
2.3	Estatísticas do Grafo	6
2.3.1	get_connected_components	6
2.3.2	get_diameter	6
2.3.3	get_diameter_node	7
2.4	Outras Funções	7
2.4.1	longest	7

2.4.2	connected	7
2.4.3	path_finder	7
2.4.4	connected_components	7
3	Resultados	7
4	Referências	8
5	Apêndice	9
5.1	word_ladder.cpp	9
5.2	makefile	21

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 programação 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 da classe `hashTable` é um construtor padrão que inicializa todas as variáveis da classe quando um objeto da classe é criado. Ele é chamado quando o objeto é criado sem argumentos ou com o construtor padrão.

Ele define o tamanho da tabela hash como sendo 65536. Ele cria um array de ponteiros chamado "words" com o tamanho da tabela. Ele inicializa a variável `entries` para 0, a variável `connected_components` para 0 e a variável `load_factor` para 0,75. Ele define todos os elementos do array `words` como `nullptr`, o que representa que os elementos da tabela estão vazios.

2.1.2 Destructor

O destrutor ele faz um loop pelo array `words` e deleta cada elemento não nulo e finalmente deleta o array `words`.

2.1.3 add

Esta função é usada para adicionar uma nova palavra à tabela hash. 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, ele 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 então 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 ficando cheia e a função chama 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 inserir no índice. Se o elemento no índice calculado não for nulo e a palavra armazenada nesse elemento não for a mesma da palavra a ser adicionada, a função entra em um loop while. O loop while continua até encontrar um espaço vazio na tabela ou a palavra já estiver na tabela. Para encontrar o próximo espaço disponível, ele usa o sondagem linear. A sondagem linear é uma técnica de resolução de colisão onde o próximo espaço é encontrado incrementando o índice em 1 e tomando o resto quando dividido pelo tamanho da tabela. Se um espaço vazio for encontrado, a função chama a função `create()` para criar um novo nó com a palavra e inserir no índice.

Observe que essa implementação usa o endereçamento aberto como técnica de resolução de colisão, mais especificamente a sondagem linear.

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 é nulo, a função retorna `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. Se não é a mesma, a função entra em um loop while e usa o sondagem linear para encontrar a palavra ou um espaço vazio. Se a palavra é encontrada, a função retorna o endereço desse elemento. Se não é encontrada, a função retorna `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 da busca em largura. Ela começa percorrendo todos os nós e marcando-os como não visitados e sem pais. 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 da busca em profundidade. 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 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 grafo. O diâmetro é o caminho mais longo entre dois nós no grafo. 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 caminho mais longo entre uma palavra dada e todas as outras palavras na tabela de hash. 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

4 Referências

- [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).

5 Apêndice

5.1 word_ladder.cpp

```
1  #include <algorithm>
2  #include <fstream>
3  #include <iostream>
4  #include <string>
5  #include <cmath>
6  #include <vector>
7  #include <queue>
8  #include <stack>
9  #include <thread>
10
11 using namespace std;
12 #define _max_word_size_ 32
13
14 class node {
15 public:
16     node(const string &word) : word(word) {
17         parent = nullptr;
18         visited = false;
19         representative = this;
20         vertices = 1;
21         edges = 0;
22     }
23
24     string word;
25     // search relevant data
26     node *parent;
27     bool visited;
28     // graph data structure
29     vector<node *> adjacency_list;
30     // union-find data structure
31     node *representative;
32     int vertices;
33     int edges;
34 };
35
36 class hashTable {
37 public:
38     unsigned int size;
39     node **words;
40     unsigned int entries;
41     int connected_components;
42     double load_factor;
43
44     hashTable() {
45         // 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 !=
77                 word) {
78                 // Linear probing is the fastest way.
79                 // Probably because it uses the cache more
80                 // efficiently.
81                 // And that matters the most when the table is huge
82                 // and we have memory to spare.
83                 index = (index + 1) % size;
84             }
85             create(index, word);
86         }
87     }
88
89     node *get(const string &word) {
90         unsigned int index = hash(word);
91         if (words[index] == nullptr) {
92             return nullptr;
93         }
94         if (words[index]->word == word) {

```

```

92         return words[index];
93     }
94     while (words[index] != nullptr && words[index]->word !=
95         word) {
96         index = (index + 1) % size;
97     }
98     return words[index];
99 }
100
101 // graph functions
102 void add_edge(node *from, node *to) {
103     from->adjacency_list.push_back(to);
104     to->adjacency_list.push_back(from);
105     from->edges++;
106     to->edges++;
107     g_union(from, to);
108 }
109
110 int BFS(node *from, node *to, int maximum_depth = 0) {
111     for (unsigned int i = 0; i < size; i++) {
112         if (words[i] != nullptr) {
113             words[i]->visited = false;
114             words[i]->parent = nullptr;
115         }
116     }
117     queue < node * > q;
118     from->visited = true;
119     q.push(from);
120     int depth = 0;
121     while (!q.empty()) {
122         int q_size = q.size();
123         for (int i = 0; i < q_size; i++) {
124             node *current = q.front();
125             q.pop();
126             if (current == to) {
127                 return depth;
128             }
129             for (size_t j = 0; j <
130                 current->adjacency_list.size(); j++) {
131                 node *adjacent = current->adjacency_list[j];
132                 if (!adjacent->visited) {
133                     adjacent->visited = true;
134                     adjacent->parent = current;
135                     q.push(adjacent);
136                 }
137             }
138             depth++;
139             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 !=
164                     from) {
165                     current = current->parent;
166                     depth++;
167                 }
168                 return depth;
169             }
170             for (size_t j = 0; j <
171                 current->adjacency_list.size(); j++) {
172                 node *adjacent = current->adjacency_list[j];
173                 if (!adjacent->visited) {
174                     adjacent->visited = true;
175                     adjacent->parent = current;
176                     q.push(adjacent);
177                 }
178             }
179         }
180     }
181     return -1;
182 }
183
184 void list_connected_components(const string &word) {
185     vector < node * > components;
186     node *vertex = get(word);
187     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
304             expensive.
305         int coeff = 4;
306         size *= coeff;
307         node **new_words = new node *[size];
308         for (unsigned int i = 0; i < size; i++) {
309             new_words[i] = nullptr;
310         }
311         for (unsigned int i = 0; i < size / coeff; i++) {
312             if (words[i] != nullptr) {
313                 int index = hash(words[i]->word);
314                 if (new_words[index] == nullptr) {
315                     new_words[index] = words[i];
316                 } else {
317                     while (new_words[index] != nullptr) {
318                         index = (index + 1) % size;
319                     }
320                 }
321             }
322         }
323         delete[] words;
324         words = new_words;
325     }
326
327     node *find(node *vertex) {
328         if (vertex->representative != vertex) {
329             vertex->representative = find(vertex->representative);
330         }
331     }

```

```

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 //
366     https://github.com/skeeto/hash-prospector#three-round-functions
367 // Kept for reference.
368 unsigned int hash(int x) {
369     x ^= x >> 17;
370     x *= 0xed5ad4bb;
371     x ^= x >> 11;
372     x *= 0xac4c1b51;
373     x ^= x >> 15;
374     x *= 0x31848bab;
375     x ^= x >> 14;
376     return x;
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
415                 string &end) {
416     if (start.size() != end.size()) {
417         cout << "Cannot compare different sizes." << endl;
418         return;
419     }
420     hashTable *dict = dicts[start.size() - 1];
421     cout << "Trying to go from " << start << " to " << end << endl;
422     node *from = dict->get(end);
423     node *to = dict->get(start);
424     if (from == nullptr || to == nullptr) {
425         cout << "No path found." << endl;
426         return;
427     }

```

```

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" <<
451                 endl;
452             file << "Size: " << dicts[i]->size << endl;
453             file << "Load factor: " << dicts[i]->get_load_factor() <<
454                 endl;
455             file << "Collisions: " << dicts[i]->get_collisions() << endl;
456             #if defined(_detail_) || defined(_full_)
457                 vector<bool> distribution = dicts[i]->get_distribution();
458                 file << "Distribution: " << endl;
459                 for (size_t j = 0; j < distribution.size(); j++)
460                 {
461                     if (distribution[j])
462                         file << j << " ";
463                 }
464                 file << endl;
465             #endif
466             #endif
467             delete dicts[i];
468         }
469         #if defined(_stats_) || defined(_detail_) || defined(_full_)
470             file.close();
471         #endif
472     }
473
474 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(),
501                     dict->words[i]->representative) == reprs.end()) {
502                 reprs.push_back(dict->words[i]->representative);
503                 int depth = dict->get_diameter(dict->words[i], false);
504                 if (depth > largest) {
505                     largest = depth;
506                     max = dict->words[i];
507                 }
508             }
509         }
510     }
511     node *origin = dict->get_diameter_node(max);
512     if (origin == nullptr || max == nullptr) {
513         cout << "No path found." << endl;
514         return;
515     }
516     dict->DFS(origin, max);
517     node *res = max;
518     ofstream file;
519     file.open("longest.txt", ios::app);
520     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(dict);
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_

```