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*.

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

```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_
```