

問題解析: 給定一個大小不超過 8*8 的填字表, 並給出可填入的單字, 找出可能的組合。

作法: 用 dfs 跑主要搜尋, frontier 紀錄發現的點(stack), explored 紀錄跑到的點(stack), 每個 node 紀錄該填的位置及填入的字, 每個 node 都有自己的 domain。

(所有作法都先用長度刪除 domain, 因此 domain 內只有長度符合的字, 不會有超出格子的問題)

(且 dfs 填字完後, 若發現有無法填的, 即退回上個點, forward checking, 但因為一開始沒發現, 不小心寫進去 dfs 了, 所以它關不掉 QQ)

實驗過程:

1. 初始暴力解法:

任意選擇想填入哪個字及內容(在 code 裡是照順序), 填入字後判斷剩餘字的 domain(後填的字不可以改動前面填過的字), 再任意填, 再判斷, 直到找出答案為止。(0 是不能填的地方)

實驗結果:

```
final:
y o u r 0 0 0 0
o 0 r 0 0 0 0 0
u 0 g o 0 0 0 0
t o e 0 0 0 0 0
h 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 506e: 5
expendNode: 514
```

```
final:
0 0 0 w 0 0 0 0
0 y 0 i 0 0 0 0
y o u t h 0 0 0
0 u 0 n 0 0 0 0
0 t r e n d 0 0
0 h 0 s 0 u 0 0
0 0 0 s t e p 0
0 0 0 0 0 0 0 0

f: 619e: 6
expendNode: 631
```

```
0 0 y o u r 0 0
i 0 o 0 0 a 0 0
c o u s i n 0 0
e 0 t 0 0 g 0 0
0 0 h u g e 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 585e: 6
expendNode: 3204
```

```
w r o n g 0 u 0
o 0 0 0 r u s h
r 0 0 0 a 0 0 a
r e s i d e n t
i 0 u 0 u 0 0 e
e x p l a i n 0
d 0 e 0 t 0 o r
0 u r g e 0 0 0

f: 963e: 12
expendNode: 35854
```

說明:

f 是 frontier 最後剩餘在 stack 的數量

e 是 explored 最後剩餘在 stack 的數量

expendNode 是產生的 node 總數

2. BackTracking Search(逆序)

使用了 Minimum remaining values (MRV) heuristic, 在每次產生新 node 後, 判斷各 domain 大小, 從剩最少可填的字開始填(但填入哪個字就隨意), 直到找出答案。

實驗結果:

```
c i t y 0 0 0 0
r 0 o 0 0 0 0 0
o 0 w e 0 0 0 0
w i n 0 0 0 0 0
d 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 43e: 5
expendNode: 48
```

```
final:
0 0 0 p 0 0 0 0
0 c 0 o 0 0 0 0
p r o v e 0 0 0
0 o 0 e 0 0 0 0
0 w o r r y 0 0
0 d 0 t 0 o 0 0
0 0 0 y o u r 0
0 0 0 0 0 0 0 0

f: 222e: 6
expendNode: 228
```

```
0 0 o n c e 0 0
y 0 t 0 0 s 0 0
o t h e r s 0 0
u 0 e 0 0 a 0 0
0 0 r e l y 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 177e: 6
expendNode: 188
```

```
p i a n o 0 w 0
r 0 0 0 v i e w
o 0 0 0 e 0 0 i
c h a i r m a n
e 0 p 0 l 0 0 g
s u p p o r t 0
s 0 l 0 o 0 o r
0 w e e k 0 0 0

f: 105e: 12
expendNode: 171
```

說明: 由結果可看出新增的 node 數量比暴力解少許多, 且最後一筆測資也明顯比暴力解快, 大約在幾百個 node 之內可以解決。

但若情況是 worst case, 也就是每個 domain 的大小都一樣的時候, 這個方法就會跟暴力解一樣, 需要其他 heuristic 來縮減 domain 才可以優化。像是 degree heuristic, 把關聯最多的字先填, 就可以減少更多不必要的搜尋。

3. BackTracking Search(正序)

將原本從 z~a 的順序改成從 a~z, 驗證看看平均數量

實驗結果:

```
final:
a w a y 0 0 0 0
h 0 w 0 0 0 0 0
e 0 a d 0 0 0 0
a n y 0 0 0 0 0
d 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 134e: 5
expendNode: 139
```

```
final:
0 0 0 o 0 0 0 0
0 a 0 v 0 0 0 0
a s s e t 0 0 0
0 i 0 r 0 0 0 0
0 d r a m a 0 0
0 e 0 l 0 c 0 0
0 0 0 l a t e 0
0 0 0 0 0 0 0 0

f: 183e: 6
expendNode: 192
```

```
0 0 a b l e 0 0
a 0 d 0 0 x 0 0
c a m e r a 0 0
t 0 i 0 0 c 0 0
0 0 t e n t 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

f: 216e: 6
expendNode: 222
```

```
b a s i c 0 a 0
a 0 0 0 a i d e
l 0 0 0 t 0 0 a
a c a d e m i c
n 0 d 0 g 0 0 h
c o m f o r t 0
e 0 i 0 r 0 o f
0 s t a y 0 0 0

f: 62e: 12
expendNode: 80
```

說明: 由實驗結果可看出 **expend** 出來的 **node** 數量差距不大, 且整體字母比較前面(不過最後一種情況應該是巧合, 剛好造成地圖較大, 卻比較快找到的結果)

4. BackTracking Search

再額外新增 **degree heuristic**, 當要選擇填哪個位置時, 若遇到剩餘 **domain** 大小一樣情況, 根據當前地圖, 選擇影響最大的位置, 例如 **domain** 都是 25 的話, 選擇有三個交叉點的填而不選兩個。

實驗結果:

```
final:
s h a r p 0 a 0
a 0 0 0 l a s t
t 0 0 0 e 0 0 e
i n s t a n c e
s 0 e 0 s 0 0 n
f o r m u l a 0
y 0 v 0 r 0 d o
0 g e n e 0 0 0

f: 67e: 12
expendNode: 101
```

說明: 上圖是第四種跑出來的結果, 其餘三種皆跟上一種一樣, 而且這個反而跑更多 node..., 有可能是因為照順序跑剛好跑到有解, 而增加 degree heuristic 雖然平均起來較快, 但也因此避開了巧合解, 所以 node 數量變多

其他想法:

1. 如果使用 AC-3 演算法的話, 就可以直接從 binary constraint 下手, 不用一個一個嘗試, expend 的 node 數量感覺也會明顯下降, 而且甚至不會避掉巧合(畢竟巧合也在 binary constraint 下), 不過實做上我沒什麼想法...

2. 至於跑出全部的結果我覺得是有可能的, 只要每輸出一個結果, 就把剛填過的最後一個字從該行(列)的 domain 拿掉, 再繼續跑應該就可以了>< (等於是假裝這個結果不行繼續跑)

3. 每次都隨機輸出的話我想到的只有用上面 MRV heuristic 的方式, 在 push node 到 frontier 上時, 亂數 push, 這樣就有可能每次都不同

總結:

我覺得填字問題, 對變數的限制數最高只有 2, 用 AC-3 最為適合, 因為 binary constraint 是這個問題最麻煩的事, 且數量跟其他 constraint 比起來也少了許多。(但感覺好難做不出來)

以我實驗的幾種方法來看, 在單字數量到 3000 的情況下, MRV+forward checking 就可以達到不錯的效果, 但當數量過於龐大的時候, 加入 degree heuristic 穩定 expendNode 的數量似乎比較好。

Code:

```
1 #include<bits/stdc++.h>
2 using namespace std;
3
4 int expendNode = 0;
5 //int ranNode = 0;
6 struct node{
7     int variable; // to store which blank we filled
8     string value; // to store what word we filled
9 };
10
11 void loadWord(vector<string> &dictionary){
12     ifstream f1("English words 3000.txt", ios::in);
13     string word;
14     while(getline(f1, word)){
15         stringstream is(word);
16         string tempword;
17         is >> tempword;
18         dictionary.push_back(tempword);
19     }
20 } // memorize 3000 english word
21
22 void loadDomain(vector<vector<string>> &domain, vector<vector<int>> &position, vector<string> &dictionary,
23 vector<bool> &visited, int depth){
24     for(int i=0; i<depth; i++){
25         if(!visited[i]){
26             for(int j=0; j<2932; j++){
27                 if(position[i][2] == dictionary[j].length()){
28                     domain[i].push_back(dictionary[j]);
29                 }
30             }
31         }
32     }
33 } // find the initial domain for each variable by length
34
35 void makeSize(vector<vector<string>> &domain, vector<int> &domainSize, int depth){
36     for(int i=0; i<depth; i++){
37         domainSize[i] = domain[i].size();
38     }
39 } // find the domain size for each variable
40
```

```

40
41 void loadMap(vector < vector <int> > &position, vector < vector <char> > &map, int depth){
42     for(int i=0; i<depth; i++){
43         if(position[i][3] == 0){
44             for(int j=0; j<position[i][2]; j++){
45                 map[position[i][1]][position[i][0]+j] = '1';
46             }
47         }else{
48             for(int j=0; j<position[i][2]; j++){
49                 map[position[i][1]+j][position[i][0]] = '1';
50             }
51         }
52     }
53 } // set the initial map, '1' for blank that can fill word, '0' for the place can't fill word
54
55 void loadMap2(vector < vector <int> > &position, vector < vector <char> > &map, int depth){
56     for(int i=0; i<depth; i++){
57         if(position[i][3] == 0){
58             for(int j=0; j<position[i][2]; j++){
59                 map[position[i][1]][position[i][0]+j]++;
60             }
61         }else{
62             for(int j=0; j<position[i][2]; j++){
63                 map[position[i][1]+j][position[i][0]]++;
64             }
65         }
66     }
67 } // set the map for binaryConstraint to use, '2' means there has binaryConstraint
68

```

```

69 int findMin(vector < vector <int> > &position, vector < int > &domainSize, vector < vector <char> > &map,
70 vector < bool> visited, vector < int > &binaryCons, int depth){
71     int min = 3001;
72     int index = 0;
73     for(int i=0; i<depth; i++){
74         if(min > domainSize[i] && !visited[i]){
75             min = domainSize[i];
76             index = i;
77         }else if(min == domainSize[i] && !visited[i]){ // if domain size equal, return the one which has more binaryConstraint
78             if(binaryCons[index] > binaryCons[i]){
79                 continue;
80             }else{
81                 index = i;
82             }
83         }
84     }
85     return index;
86 } // find the minimum domain for the current domain, return that variable
87
88 void loadBinaryCons(vector < vector <int> > &position, vector < vector <char> > &map, vector < int > &binaryCons,
89 int depth){
90     for(int i=0; i<depth; i++){
91         if(position[i][3] == 0){
92             for(int j=0; j<position[i][2]; j++){
93                 if(map[position[i][1]][position[i][0]+j] == '2'){
94                     binaryCons[i]++;
95                 }
96             }
97         }else{
98             for(int j=0; j<position[i][2]; j++){
99                 if(map[position[i][1]+j][position[i][0]] == '2'){
100                     binaryCons[i]++;
101                 }
102             }
103         }
104     }
105 } // set how many binaryConstraint that each variable has, "degree heuristic"
106

```

```

107 void setConstraint(vector < vector <int> > &position, vector < vector <string> > &domain, vector < int > domainSize,
108 vector < vector <char> > &map, vector < bool> visited, int depth){
109     for(int i=0; i<depth; i++){
110         if(!visited[i]){
111             if(position[i][3] == 0){
112                 for(int j=0; j<position[i][2]; j++){
113                     if(map[position[i][1]][position[i][0]+j] != '1'){
114                         for(int k=0, c=0; c<domainSize[i]; k++,c++){
115                             if(domain[i][k][j] != map[position[i][1]][position[i][0]+j]){
116                                 domain[i].erase(domain[i].begin()+k);
117                                 k--;
118                             }
119                         }
120                     }
121                 }
122             }else{
123                 for(int j=0; j<position[i][2]; j++){
124                     if(map[position[i][1]+j][position[i][0]] != '1'){
125                         for(int k=0, c=0; c<domainSize[i]; k++,c++){
126                             if(domain[i][k][j] != map[position[i][1]+j][position[i][0]]){
127                                 domain[i].erase(domain[i].begin()+k);
128                                 k--;
129                             }
130                         }
131                     }
132                 }
133             }
134         }
135     }
136 }
137 } // update the domain for each variable by current map
138

```



```

138
139 void dfsSearch(vector < vector <int> > &position, vector < vector <string> > domain, stack < node > &explored,
140 stack < node > &frontier, vector < vector <char> > &map, vector < int > domainSize, vector < bool > visited,
141 vector < int > &binaryCons, vector < string > &dictionary, int depth, int newNode){
142 if(explored.size() == depth) return;
143 int index;
144 index = findMin(position, domainSize, map, visited, binaryCons, depth);
145 // the variable that we want to fill, "MRV heuristic"
146
147 if(domain[index].size() == 0){
148     int undo = explored.top().variable;
149     visited[undo] = false;
150     explored.pop();
151     loadMap(position, map, depth);
152     stack < node > temp = explored;
153     while(!temp.empty()){
154         int tempindex = temp.top().variable;
155         if(position[tempindex][3] == 0){
156             for(int j=0; j<position[tempindex][2]; j++){
157                 map[position[tempindex][1]][position[tempindex][0]+j] = temp.top().value[j];
158             }
159         }else{
160             for(int j=0; j<position[tempindex][2]; j++){
161                 map[position[tempindex][1]+j][position[tempindex][0]] = temp.top().value[j];
162             }
163         }
164         temp.pop();
165     }
166     return;
167 }
168 // if that word will let the remain blank can't fill word, pop it and use another frontier = "forward checking"
169 // and draw a new map by current explored
170
171 visited[index] = true;
172 // use for dfs to remember which variable has ran

```

```

171 visited[index] = true;
172 // use for dfs to remember which variable has ran
173
174 newNode = 0;
175 // use to memorize how many node will expend if this current node push
176
177 for(int i=domainSize[index]-1; i>=0; i--){
178     node temp;
179     temp.variable = index;
180     temp.value = domain[index][i];
181     frontier.push(temp);
182     newNode++;
183     expendNode++;
184 }
185 // expend node
186
187 while(!frontier.empty() && explored.size()!=depth){
188     if(newNode == 0) {
189         explored.pop();
190         loadMap(position, map, depth);
191         stack < node > temp = explored;
192         while(!temp.empty()){
193             int tempindex = temp.top().variable;
194             if(position[tempindex][3] == 0){
195                 for(int j=0; j<position[tempindex][2]; j++){
196                     map[position[tempindex][1]][position[tempindex][0]+j] = temp.top().value[j];
197                 }
198             }else{
199                 for(int j=0; j<position[tempindex][2]; j++){
200                     map[position[tempindex][1]+j][position[tempindex][0]] = temp.top().value[j];
201                 }
202             }
203             temp.pop();
204         }
205         return;
206     }
207     // newNode == 0 means the frontier it expend is all pop, i need a new explore and return to it's domain
208

```

```

208
209 int draw = frontier.top().variable;
210 if(position[draw][3] == 0){
211     for(int j=0; j<position[draw][2]; j++){
212         map[position[draw][1]][position[draw][0]+j] = frontier.top().value[j];
213     }
214 }else{
215     for(int j=0; j<position[draw][2]; j++){
216         map[position[draw][1]+j][position[draw][0]] = frontier.top().value[j];
217     }
218 }
219 //draw the word on the map
220
221 vector < vector <string> > newDomain(depth);
222 vector < int > newDomainSize(depth);
223 // new domain and domainSize for dfs
224
225 for(int i=0; i<depth; i++){
226     newDomain[i] = domain[i];
227 }
228 newDomainSize = domainSize;
229
230
231 setConstraint(position, newDomain, newDomainSize, map, visited, depth);
232 //rebuilt each domain
233
234 makeSize(newDomain, newDomainSize, depth); //rebuilt each size
235
236 explored.push(frontier.top());
237 ranNode++;
238 //let the word in the explored
239
240 frontier.pop();
241 //let the word out of frontier
242 newNode--;
243
244 dfsSearch(position, newDomain, explored, frontier, map, newDomainSize, visited, binaryCons, dictionary,
245 depth, newNode);
246 }
247 }

```

```

248
249 int main(){
250     vector < vector <int> > position;
251     //to store puzzle.txt's information: x, y, length, direction
252
253     vector <string> dictionary;
254     //to store 3000 english word
255
256     stack < node > explored;
257     //data structure to store the node we explored, in detail, which blank we fill in what word
258
259     stack < node > frontier;
260     //data structure to store the node we can explored, in detail, which will be the next to refill
261
262     ifstream fin("puzzle.txt", ios::in);
263     string str;
264
265     loadWord(dictionary);
266     // read 3000 english word
267
268     getline(fin,str);
269     getline(fin,str);
270     getline(fin,str);
271     getline(fin,str);
272     // first test for one getline (i don't have time to built it in while, it will explode QQ)
273
274     position.clear();
275     if(!explored.empty()){
276         explored.pop();
277     }
278     if(!frontier.empty()){
279         frontier.pop();
280     }
281     // initialize
282
283     vector < vector <char> > map(8, vector<char> (8, '0'));
284     //to store current drawing on the map, '1' for blank, '0' can't fill
285
286     vector < vector <char> > map2(8, vector<char> (8, '0'));
287     // map to find binaryconstraint
288

```

```

288
289     int i=0;
290     int depth=0;
291     //how many variable to fill
292     istringstream iss(str);
293     vector <int> tmp;
294     char temp;
295     while(iss >> temp){
296         if(i<4){
297             if(temp == 'A'){
298                 tmp.push_back(0);
299                 i++;
300             }else if(temp == 'D'){
301                 tmp.push_back(1);
302                 i++;
303             }else{
304                 tmp.push_back(temp-'0');
305                 i++;
306             }
307         }
308         if(i==4){
309             position.push_back(tmp);
310             tmp.clear();
311             i = 0;
312             depth++;
313         }
314     }
315     // memorize the position for each variable
316
317     int newNode=0;
318
319     vector <bool> visited(depth, false);
320     // to recognize which variable have filled
321
322     vector < vector <string> > domain(depth);
323     // to memorize each variable's domain by length
324
325     loadDomain(domain, position, dictionary, visited, depth);
326     // built domain by length
327
328     vector < int > domainSize(depth, 0);
329     // each variable's domain size
330

```

```

330
331     vector < int > domainSize(depth, 0);
332     // each variable's domain size
333
334     vector < int > binaryCons(depth, 0);
335     // each variable's binaryconstraint's number
336
337     makeSize(domain, domainSize, depth);
338     // make the size of each domain
339
340     loadMap(position, map, depth);
341     // built the initial map
342
343     loadMap2(position, map2, depth);
344     // built the map for binaryconstraint
345
346     loadBinaryCons(position, map2, binaryCons, depth);
347     // count binaryConstraint
348
349     dfsSearch(position, domain, explored, frontier, map, domainSize, visited, binaryCons, dictionary, depth, newNode);
350     // main algorithm
351
352     cout << "final:" << endl;
353     for(int i=0; i<8; i++){
354         for(int j=0; j<8; j++){
355             cout << map[i][j] << " ";
356         }
357         cout << endl;
358     }
359     cout << "f: " << frontier.size() << "e: " << explored.size() << endl;
360     cout << "expendedNode: " << expendedNode << endl;
361     //cout << "ranNode: " << ranNode << endl;
362     // final answer
363
364     return 0;
365 }
366

```