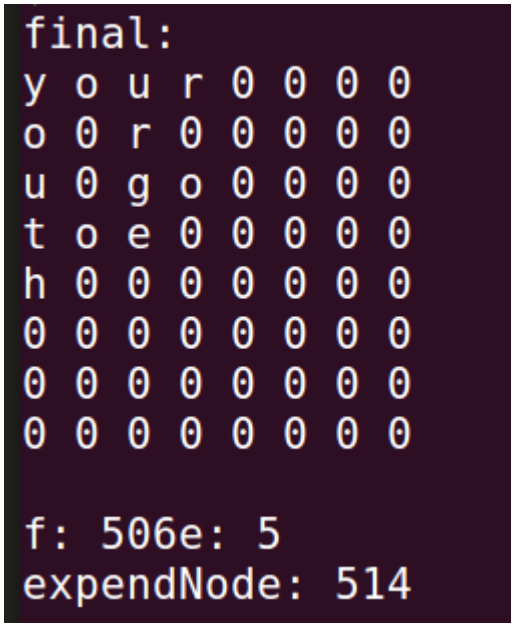
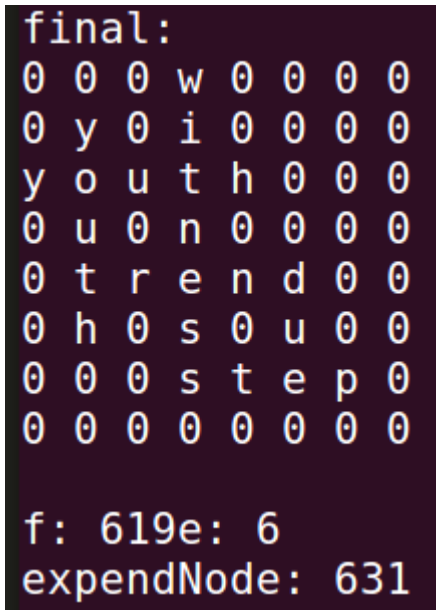
Programming Assignment #1 0616223 劉柏宇

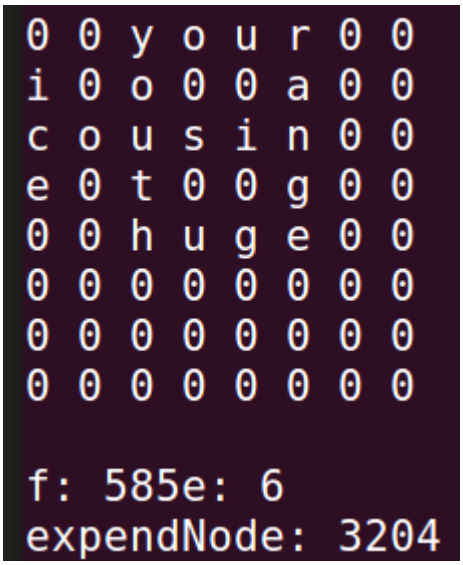
問題解析: 給定一個大小不超過 8*8 的填字表, 並給出可填入的單字, 找出可能的組合。

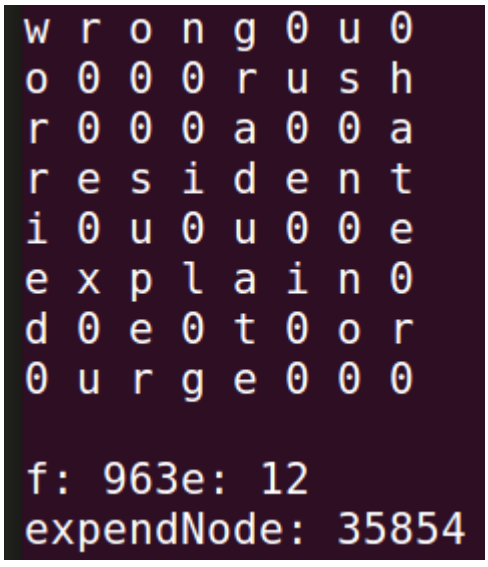
作法: 用 dfs 跑主要搜尋,frontier 紀錄發現的點(stack), explored 紀錄跑到的點(stack), 每個 node 紀錄該填的位置及填入的字, 每個 node 都有自己的 domain。
(所有作法都先用長度刪除 domain, 因此 domain 內只有長度符合的字, 不會有超出格子的問題)
(且 dfs 填字完後, 若發現有無法填的, 即退回上個點, forward checking, 但因為一開始沒發現,不小心寫進去 dfs 了, 所以它關不掉 QQ)

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:









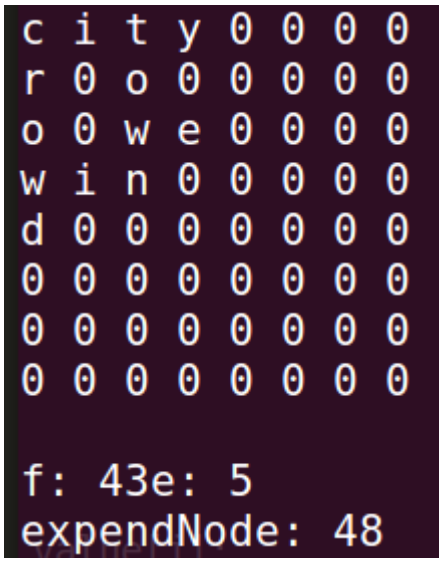
說明:
f 是 frontier 最後剩餘在 stack 的數量

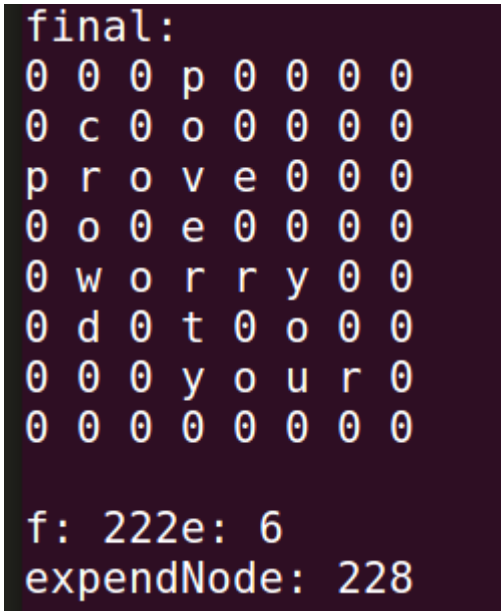
e 是 explored 最後剩餘在 stack 的數量
expendNode 是產生的 node 總數

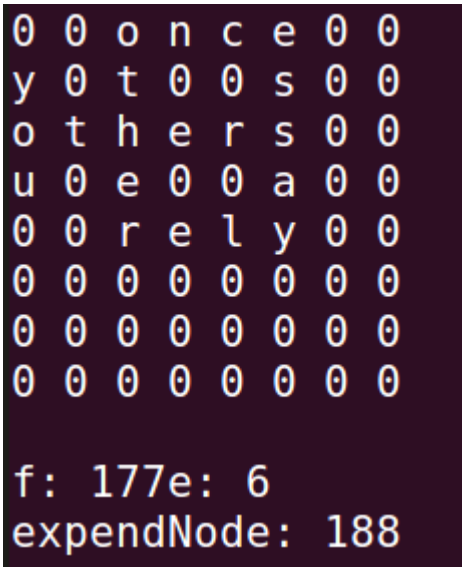
2. BackTracking Search(逆序)

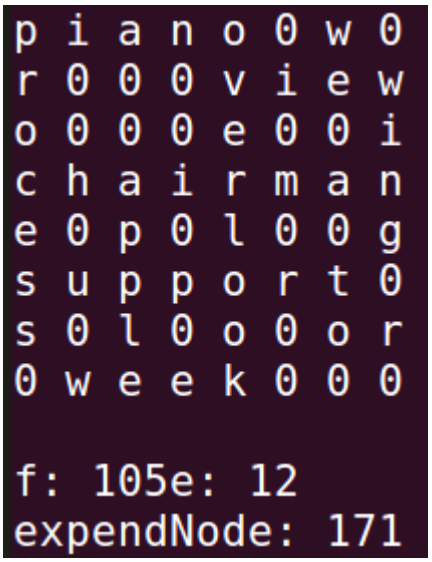
使用了 Minimum remaining values (MRV) heuristic, 在每次產生新 node 後, 判斷各 doamin 大小, 從剩最少可填的字開始填(但填入哪個字就隨意), 直到找出答案。

實驗結果:









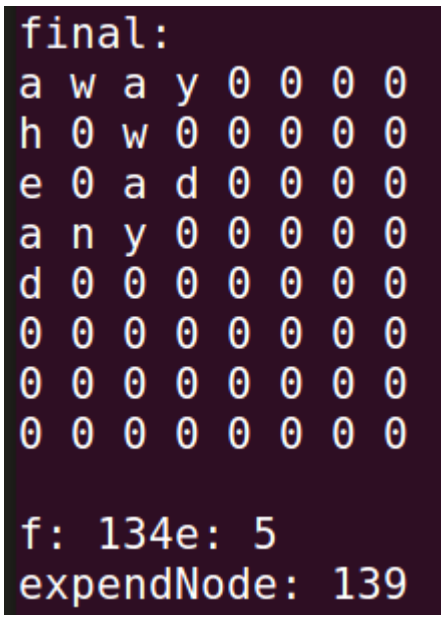
說明: 由結果可看出新增的 node 數量比暴力解少許多, 且最後一筆測資也明顯比暴力解快, 大約在幾百個 node 之內可以解決。

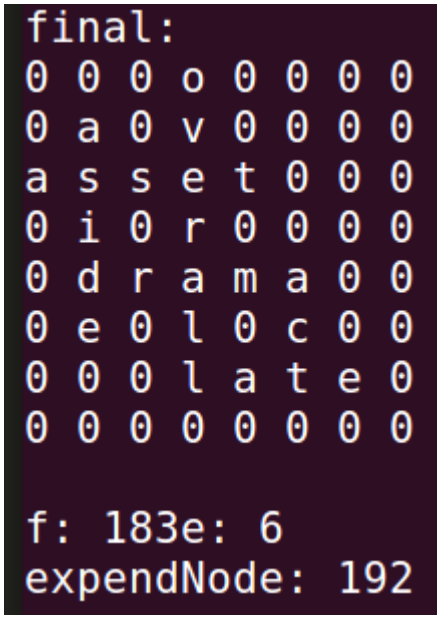
但若情況是 worst case, 也就是每個 domain 的大小都一樣的時候, 這個方法就會跟暴力解一樣, 需要其他 heuristic 來縮減 domain 才可以優化。像是 degree heuristic, 把關聯最多的字先填, 就可以減少更多不必要的搜尋。

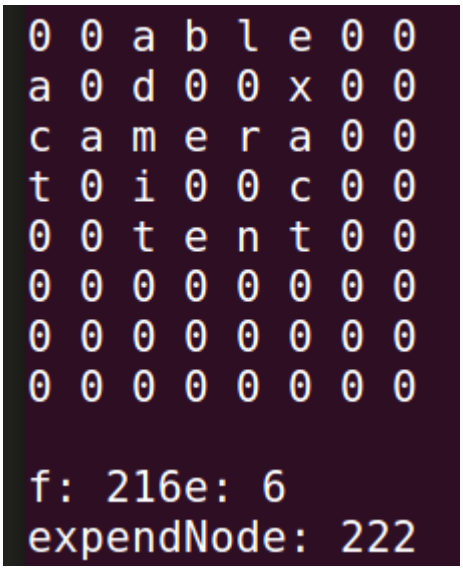
3. BackTracking Search(正序)

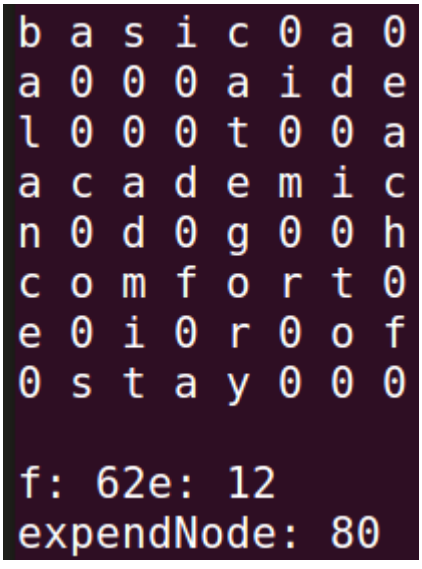
將原本從 z~a 的順序改成從 a~z, 驗證看看平均數量

實驗結果:







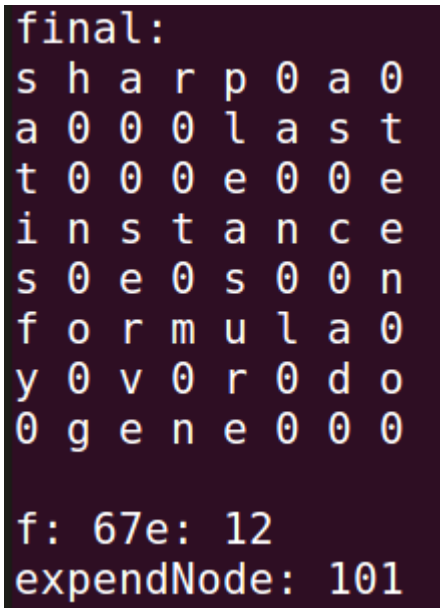


說明: 由實驗結果可看出 expend 出來的 node 數量差距不大, 且整體字母比較前面(不過最後一種情況應該是巧合, 剛好造成地圖較大, 卻比較快找到的結果)

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:



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, 只要每輸出一次結果, 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一樣

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:

puzzle.cpp

```cpp
#include<bits/stdc++.h>
using namespace std;

int expendNode = 0;
//int ranNode = 0;
struct node{
  int variable; // to store which blank we filled
  string value; // to store what word we filled
};

void loadWord(vector< string > &dictionary){
  ifstream fliein("English words 3000.txt", ios::in);
  string word;
  while(getline(fliein, word)){
    istringstream is(word);
    string tempword;
    is >> tempword;
    dictionary.push_back(tempword);
  }
} // memorize 3000 english word

void loadDomain(vector < vector <string> > &domain, vector < vector <int> > &position, vector <string> &dictionary,
  vector <bool> visited, int depth){
  for(int i=0; i<depth; i++){
    if(!visited[i]){
      for(int j=0; j<2932; j++){
        if(position[i][2] == dictionary[j].length()){
          domain[i].push_back(dictionary[j]);
        }
      }
    }
  }
} // find the initial domain for each variable by length

void makeSize(vector < vector <string> > &domain, vector <int > &domainSize, int depth){
  for(int i=0; i<depth; i++){
    domainSize[i] = domain[i].size();
  }
} // find the domain size for each variable
```

```
void loadMap(vector < vector <int> > &position, vector < vector <char> > &map, int depth){
  for(int i=0; i<depth; i++){
    if(position[i][3] == 0){
      for(int j=0; j<position[i][2]; j++){
        map[position[i][1]][position[i][0]+j] = '1';
      }
    }else{
      for(int j=0; j<position[i][2]; j++){
        map[position[i][1]+j][position[i][0]] = '1';
      }
    }
  }
} // set the initial map, '1' for blank that can fill word, '0' for the place can't fill word

void loadMap2(vector < vector <int> > &position, vector < vector <char> > &map, int depth){
  for(int i=0; i<depth; i++){
    if(position[i][3] == 0){
      for(int j=0; j<position[i][2]; j++){
        map[position[i][1]][position[i][0]+j]++;
      }
    }else{
      for(int j=0; j<position[i][2]; j++){
        map[position[i][1]+j][position[i][0]]++;
      }
    }
  }
} // set the map for binaryConstraint to use, '2' means there has binaryConstraint
```

```
int findMin(vector < vector <int> > &position, vector < int > &domainSize, vector < vector <char> > &map,
 vector <bool> visited,  vector < int > &binaryCons, int depth){
  int min = 3001;
  int index = 0;
  for(int i=0; i<depth; i++){
    if(min > domainSize[i] && !visited[i]){
      min = domainSize[i];
      index = i;
    }else if(min == domainSize[i] && !visited[i]){ // if domain size equal, return the one which has more binaryContraint
      if(binaryCons[index] > binaryCons[i]){
        continue;
      }else{
        index = i;
      }
    }
  }
  return index;
} // find the minimum domain for the current domain, return that variable

void loadBinaryCons(vector < vector <int> > &position, vector < vector <char> > &map, vector < int > &binaryCons,
 int depth){
  for(int i=0; i<depth; i++){
    if(position[i][3] == 0){
      for(int j=0; j<position[i][2]; j++){
        if(map[position[i][1]][position[i][0]+j] == '2'){
          binaryCons[i]++;
        }
      }
    }else{
      for(int j=0; j<position[i][2]; j++){
        if(map[position[i][1]+j][position[i][0]] == '2'){
          binaryCons[i]++;
        }
      }
    }
  }
} // set how many binaryConstraint that each variable has, "degree heuristic"
```

```
void setConstraint(vector < vector <int> > &position, vector < vector <string> > &domain, vector < int > domainSize,
 vector < vector <char> > &map, vector <bool> visited, int depth){
  for(int i=0; i<depth; i++){
    if(!visited[i]){
      if(position[i][3] == 0){
        for(int j=0; j<position[i][2]; j++){
          if(map[position[i][1]][position[i][0]+j] != '1'){
            for(int k=0, c=0; c<domainSize[i]; k++,c++){
              if(domain[i][k][j] != map[position[i][1]][position[i][0]+j]){
                domain[i].erase(domain[i].begin()+k);
                k--;
              }
            }
          }
        }
      }else{
        for(int j=0; j<position[i][2]; j++){

          if(map[position[i][1]+j][position[i][0]] != '1'){
            for(int k=0, c=0; c<domainSize[i]; k++,c++){
              if(domain[i][k][j] != map[position[i][1]+j][position[i][0]]){
                domain[i].erase(domain[i].begin()+k);
                k--;
              }
            }
          }
        }
      }
    }
  }
} // update the domain for each variable by current map
```

```
void dfsSearch(vector < vector <int> > &position, vector < vector <string> > domain, stack < node > &explored,
 stack < node > &frontier, vector < vector <char> > &map, vector < int > domainSize, vector <bool> visited,
 vector < int > &binaryCons, vector <string> &dictionary, int depth, int newNode){
  if(explored.size() == depth) return;
  int index;
  index = findMin(position, domainSize, map, visited, binaryCons, depth);
  // the variable that we want to fill, "MRV heuristic"

  if(domain[index].size() == 0){
    int undo = explored.top().variable;
    visited[undo] = false;
    explored.pop();
    loadMap(position, map, depth);
    stack < node > temp = explored;
    while(!temp.empty()){
      int tempindex = temp.top().variable;
      if(position[tempindex][3] == 0){
        for(int j=0; j<position[tempindex][2]; j++){
          map[position[tempindex][1]][position[tempindex][0]+j] = temp.top().value[j];
        }
      }else{
        for(int j=0; j<position[tempindex][2]; j++){
          map[position[tempindex][1]+j][position[tempindex][0]] = temp.top().value[j];
        }
      }
      temp.pop();
    }
    return;
  }
  // if that word will let the remain blank can't fill word, pop it and use another frontier = "forward checking"
  // and draw a new map by current explored

  visited[index] = true;
  // use for dfs to remember which variable has ran
```

```
  visited[index] = true;
  // use for dfs to remember which variable has ran

  newNode = 0;
  // use to memorize how many node will expend if this current node push

  for(int i=domainSize[index]-1; i>=0; i--){
    node temp;
    temp.variable = index;
    temp.value = domain[index][i];
    frontier.push(temp);
    newNode++;
    expendNode++;
  }
  // expend node

  while(!frontier.empty() && explored.size()!=depth){
    if(newNode == 0) {
      explored.pop();
      loadMap(position, map, depth);
      stack < node > temp = explored;
      while(!temp.empty()){
        int tempindex = temp.top().variable;
        if(position[tempindex][3] == 0){
          for(int j=0; j<position[tempindex][2]; j++){
            map[position[tempindex][1]][position[tempindex][0]+j] = temp.top().value[j];
          }
        }else{
          for(int j=0; j<position[tempindex][2]; j++){
            map[position[tempindex][1]+j][position[tempindex][0]] = temp.top().value[j];
          }
        }
        temp.pop();
      }
      return;
    }
    // newNode == 0 means the frontier it expend is all pop, i need a new explore and return to it's domain
```

```
    int draw = frontier.top().variable;
    if(position[draw][3] == 0){
      for(int j=0; j<position[draw][2]; j++){
        map[position[draw][1]][position[draw][0]+j] = frontier.top().value[j];
      }
    }else{
      for(int j=0; j<position[draw][2]; j++){
        map[position[draw][1]+j][position[draw][0]] = frontier.top().value[j];
      }
    }
    //draw the word on the map

    vector < vector <string> > newDomain(depth);
    vector < int > newDomainSize(depth);
    // new domain and domainsize for dfs

    for(int i=0; i<depth; i++){
      newDomain[i] = domain[i];
    }
    newDomainSize = domainSize;


    setConstraint(position, newDomain, newDomainSize, map, visited, depth);
    //rebuilt each domain

    makeSize(newDomain, newDomainSize, depth);  //rebuilt each size

    explored.push(frontier.top());
    ranNode++;
    //let the word in the explored

    frontier.pop();
    //let the word out of frontier
    newNode--;

    dfsSearch(position, newDomain, explored, frontier, map, newDomainSize, visited, binaryCons, dictionary,
     depth, newNode);
  }
}
```

```
int main(){
  vector < vector <int> > position;
  //to store puzzle.txt's information: x, y, length, direction

  vector <string> dictionary;
  //to store 3000 english word

  stack < node > explored;
  //data structure to store the node we explored, in detail, which blank we fill in what word

  stack < node > frontier;
  //data structure to store the node we can explored, in detail, which will be the next to refill

  ifstream fin("puzzle.txt", ios::in);
  string str;

  loadWord(dictionary);
  // read 3000 english word

  getline(fin,str);
  getline(fin,str);
  getline(fin,str);
  getline(fin,str);
  // first test for one getline (i don't have time to built it in while, it will explode QQ)

  position.clear();
  if(!explored.empty()){
    explored.pop();
  }
  if(!frontier.empty()){
    frontier.pop();
  }
  // initialize

  vector < vector <char> > map(8, vector<char> (8, '0'));
  //to store current drawing on the map, '1' for blank, '0' can't fill

  vector < vector <char> > map2(8, vector<char> (8, '0'));
  // map to find binaryconstraint
```

```
  int i=0;
  int depth=0;
  //how many variable to fill
  istringstream iss(str);
  vector <int> tmp;
  char temp;
  while(iss >> temp){
    if(i<4){
      if(temp == 'A'){
        tmp.push_back(0);
        i++;
      }else if(temp == 'D'){
        tmp.push_back(1);
        i++;
      }else{
        tmp.push_back(temp-'0');
        i++;
      }
    }
    if(i==4){
      position.push_back(tmp);
      tmp.clear();
      i = 0;
      depth++;
    }
  }
  // memorize the position for each variable

  int newNode=0;

  vector <bool> visited(depth, false);
  // to recgonize which variable have filled

  vector < vector <string> > domain(depth);
  // to memorize each variable's domain by length

  loadDomain(domain, position, dictionary, visited, depth);
  // built domain by length

  vector < int > domainSize(depth, 0);
  // each variable's domain size
```

```
  vector < int > domainSize(depth, 0);
  // each variable's domain size

  vector < int > binaryCons(depth, 0);
  // each variable's binaryconstraint's number

  makeSize(domain, domainSize, depth);
  // make the size of each domain

  loadMap(position, map, depth);
  // built the initial map

  loadMap2(position, map2, depth);
  // built the map for binaryconstraint

  loadBinaryCons(position, map2, binaryCons, depth);
  // count binaryConstraint

  dfsSearch(position, domain, explored, frontier, map, domainSize, visited, binaryCons, dictionary, depth, newNode);
  // main algorithm



  cout << "final:" << endl;
  for(int i=0; i<8; i++){
      for(int j=0; j<8; j++){
        cout << map[i][j] << " ";
      }
    cout << endl;
  }
  cout << endl;
  cout << "f: " << frontier.size() << "e: " << explored.size() << endl;
  cout << "expendNode: " << expendNode << endl;
  //cout << "ranNode: " << ranNode << endl;
  // final answer

  return 0;
}
```