Crossword Solving Algorithm

Thomas Laverghetta (tlave002@odu.edu),
Old Dominion University – Computational Modeling and Simulation Engineering

Please read over Assignment 3 System Requirements document (which accompanies this report) first to gain understanding of program overview.

Algorithms

In this section, I will discuss the algorithms and data structures used to solve crossword puzzles. The algorithm used to solve the crossword is backtracking. Backtracking is a general algorithm for finding all solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions (adding words to crossword), and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution. Therefore, allowing me to add words to crossword puzzle, test if word does not fail crossword rules, and either adding another word or trying another word. Figure 1 illustrates backtracking process – i.e., starting initial node then adding additional values until reaching leaf node (potential solution) before going back-up and trying a different combination. Although, before deploying backtracking, pre-word processing and data structures are needed to allow backtracking to word efficiently.

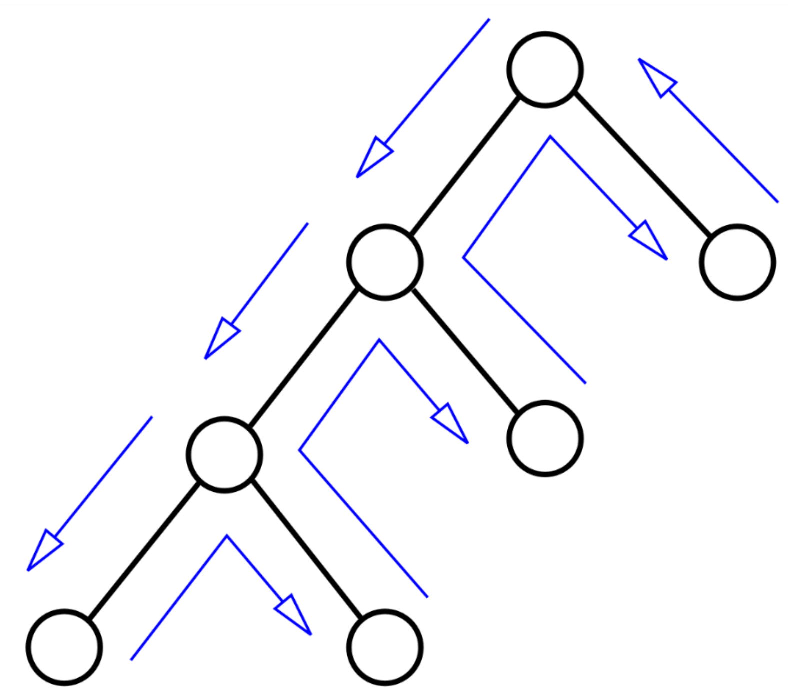


Figure 1. Backtracking Algorithm

There are three pre-word processes done before backtracking and two data structures to encapsulate pre-word processing. I will start with the data-structures then processes.

There are two data-structures used: crossword-element and crossword-element-set. A crossword-element represents a word within crossword which include the word itself and identifier (where the word belongs in crossword). A crossword-element-set represents all possible words that can be placed within crossword-element (word can fit in crossword-element). Hence, a crossword-element-set encapsulates a set of possible words and element identifier. Figure 2 illustrates the data-structures.

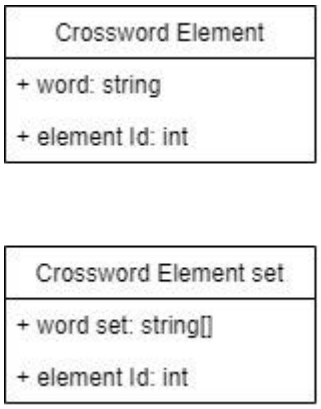


Figure 2. Data structures

The three processes: crossword element requirements loader, intersection finding, and dictionary filtering.

- Crossword element requirements process loads crossword requirements foreach element. A requirement for an element includes size of element (word size constraint), element identifier, and direction of element (vertical or horizontal).
- Intersection finding uses the information from crossword element requirement loader to find element intersections. The element intersections are a set of elements that intersect a specified element, and where elements intersect with specified element. The intersections are used in the backtracking's selecting algorithm (explained in next section).
- Dictionary filtering process loads a dictionary and iterates through the dictionary foreach crossword element to produce crossword-element-sets. Before iterating, crossword-element-sets are created based on crossword element requirement data. Where each crossword-element-set just has element identifier associated with crossword element requirement element identifier and empty word set. Within each iteration, dictionary word will be compared with crossword element requirement length to determine if word will fit within crossword-element. If word conforms, then word will be appended to crossword-element-set word set. Else, continue to next word.

Once dictionary filtering has finished and returned crossword-element-sets, the crossword-element-sets will be sorted in descending order by number of words in each set then back will be popped off and used as initial crossword-element-set to iterate across for backtracking. The pseudocode for main process (includes three processes) is shown below:

```
main (void)
1. Load and save crossword requirements (i.e., elements and their respected sizes)
2. Find and save crossword element intersections (i.e., where elements intersect)
3. Load word dictionary and filter words into crossword element sets (creating crossword element sets foreach crossword element, and foreach element set, save words from the dictionary that meet crossword element requirements (size))
4. sort crossword element sets in descending order by number of words
5. select back crossword element set from sorted sets (init element set) and pop sorted sets
6. foreach crossword element in init element set
7.     if Backtracking (crossword element, sorted set)
8.         break
9.     end if
10. end for
11. if sorted set length == 0
12.     solution found
13. else
14.     no solution found
15. end if
```

Note, & in parameter input is pass by reference.

Backtracking Algorithm

Below is the pseudocode for backtracking algorithm. The algorithm starts by receiving the current elements (CE) in play and remaining element sets (the remaining crossword elements to play) (RES). Using those inputs, it gets the next element set by calling select process. If no set is returned, then backtracking will return false. Else, it will test if RES is zero (i.e., no remaining elements). If true, return CI and true. Else, it iterates through each word in the element set and call backtracking. If that comes back true, return true. Else, continue to next word.

```
Backtracking (& curr elements (CE)[], remaining elements sets (RES)[])
1. Get next crossword element set {SelectNextElementSet(CE.end, RES)}
2. if getting next crossword element returns true (element found):
3.     if RES length equals 0:
4.         Append RES first element to CI
5.         save CI as solution set
6.         return true
7.     end if
8.     foreach element in next crossword element set:
9.         Append element to CE
10.        if Backtracking (CE, RES)
11.            return true
12.        end if
13.        Pop element from CE
14.    end for
15. return false
```

Selecting Algorithm for Task 1

The selecting algorithm for task 1 gets first element set and sets it a tmp value then compares against each current crossword element intersection to determine if tmp is an intersection. If it is an intersection, then it will be tested further to find if there any words within tmp's word set that conform with intersection. Any word that conforms gets saved into a new crossword element set. If there are word that conform, it will set tmp to new crossword element set (it has new set of conforming words from tmp). Else, it will return false since this element set will never conform. If a false is never returned (compared against all intersections), then next element set will be set to tmp and returned. Pseudocode below:

```
SelectNextElementSet (const& curr element set, & RES)
1. create tmp crossword element set and set it back RES
2. foreach element in curr element set:
3.     foreach intersection element (IE) in element:
4.         if IE is tmp:
5.             Allocate new crossword element
6.             foreach word in IE:
7.                 if word does not conflict with newE's word
8.                     Append word to new crossword element word set
9.                 end if
10.            end for
11.            if new crossword element word set length > 0:
12.                set tmp to next crossword element
13.            else:
14.                return false (no solution with given words)
15.            end if
16.        end if
17.    end for
18. end for
19. set next crossword element set to tmp
20. Pop RES
21. return next crossword element
```

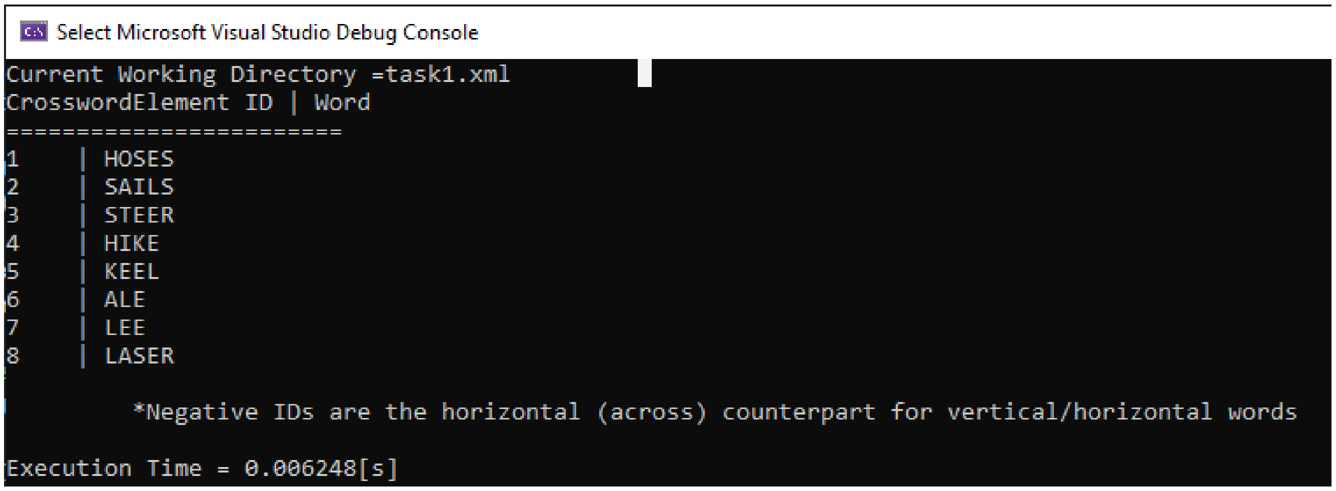
Selecting Algorithm for Task 2

The selecting algorithm for task 2 has memory of previous tests associated with it therefore allowing me to only test the latest element added to the crossword. This algorithm takes the newest element added to the crossword and tests any intersecting elements in RES. The test itself, determines if there are any words that conform with the newest element, and if there is, saves them to a tmp element set array. If any element that intersects does not conform, then the process will return a failure. Once all intersections have been tested, the process will check if any intersections where found in RES (tmp length = 0). If none were found, the process will sort RES in descending order by size of word sets, sets RES back to next crossword element, pops RES back, and returns next crossword element. Else (tmp length > 0), sort tmp by size of word set in ascending order, set next crossword element to tmp front, replace all RES elements corresponding to tmp elements (excluding first element in tmp), and return next crossword element. The pseudocode is below:

```
SelectNextElementSet (const& newest element added (newE), & RES)
1. Allocate temporary list called tmp
2. foreach intersection element (IE) in newE:
3.      if IE in RES:
4.          Allocate new crossword element
5.          foreach word in IE:
6.              if word does not conflict with newE's word
7.                  Append word to new crossword element word set
8.              end if
9.          end for
10.         if new crossword element word set length > 0:
11.             Append new crossword element to tmp
12.         else:
13.             return false (no solution with given words)
14.         end if
15.     end if
16. if tmp length equals zero (either no intersections or all intersections are in CE):
17.     sort RES by size of word set in descending order
18.     set next crossword element set to RES.back
19.     Pop RES
20.     return next crossword element
21. end if
22. sort tmp by size of word set in ascending order
23. set next crossword element to tmp.front
24. foreach element in tmp[1:]:
25      replace RES's element with element
26. end for
27. return next crossword element
```

Results

Using the algorithm described in the previous section, I was able to solve task 1 and 2's crossword puzzles (outlined in system requirements) and their respective execution times for each puzzle. The following were the outputs from puzzles along with execution times. Note, the data is formatted by crossword element identifier (ID) and element word. Element Identifiers that are negative are the horizontal (across) counterpart for vertical/horizontal crossword elements.

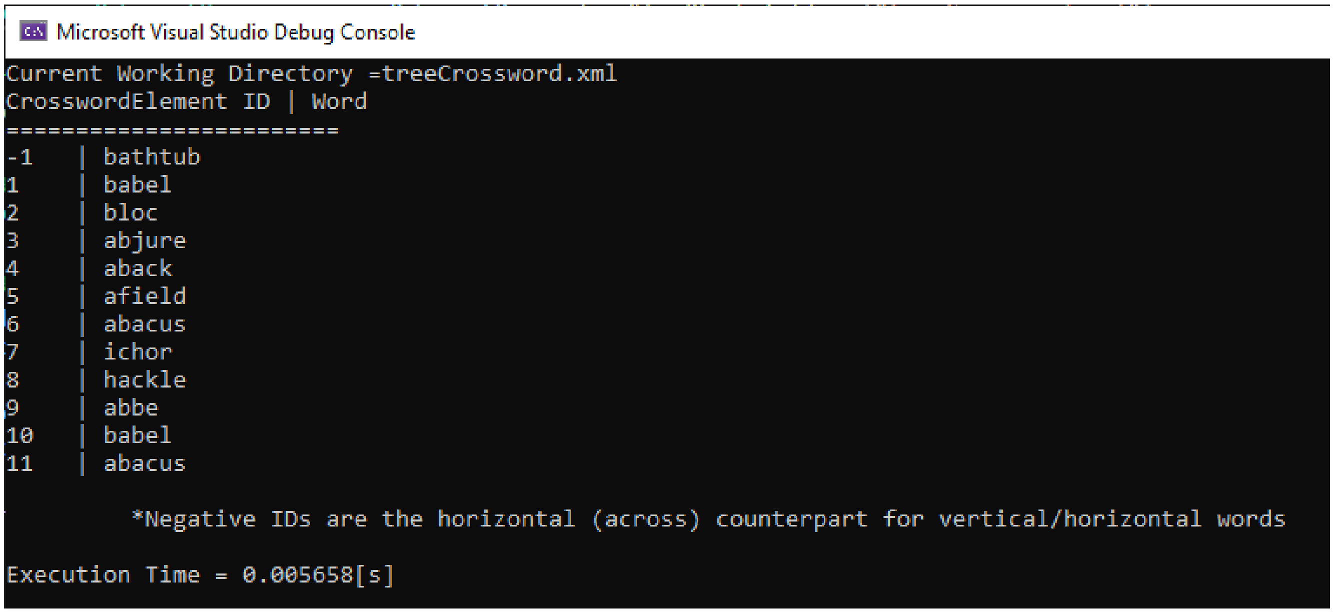


```
Current Working Directory =task1.xml
CrosswordElement ID | Word
========================
1     | HOSES
2     | SAILS
3     | STEER
4     | HIKE
5     | KEEL
6     | ALE
7     | LEE
8     | LASER

         *Negative IDs are the horizontal (across) counterpart for vertical/horizontal words

Execution Time = 0.006248[s]
```

Figure 3. Task 1

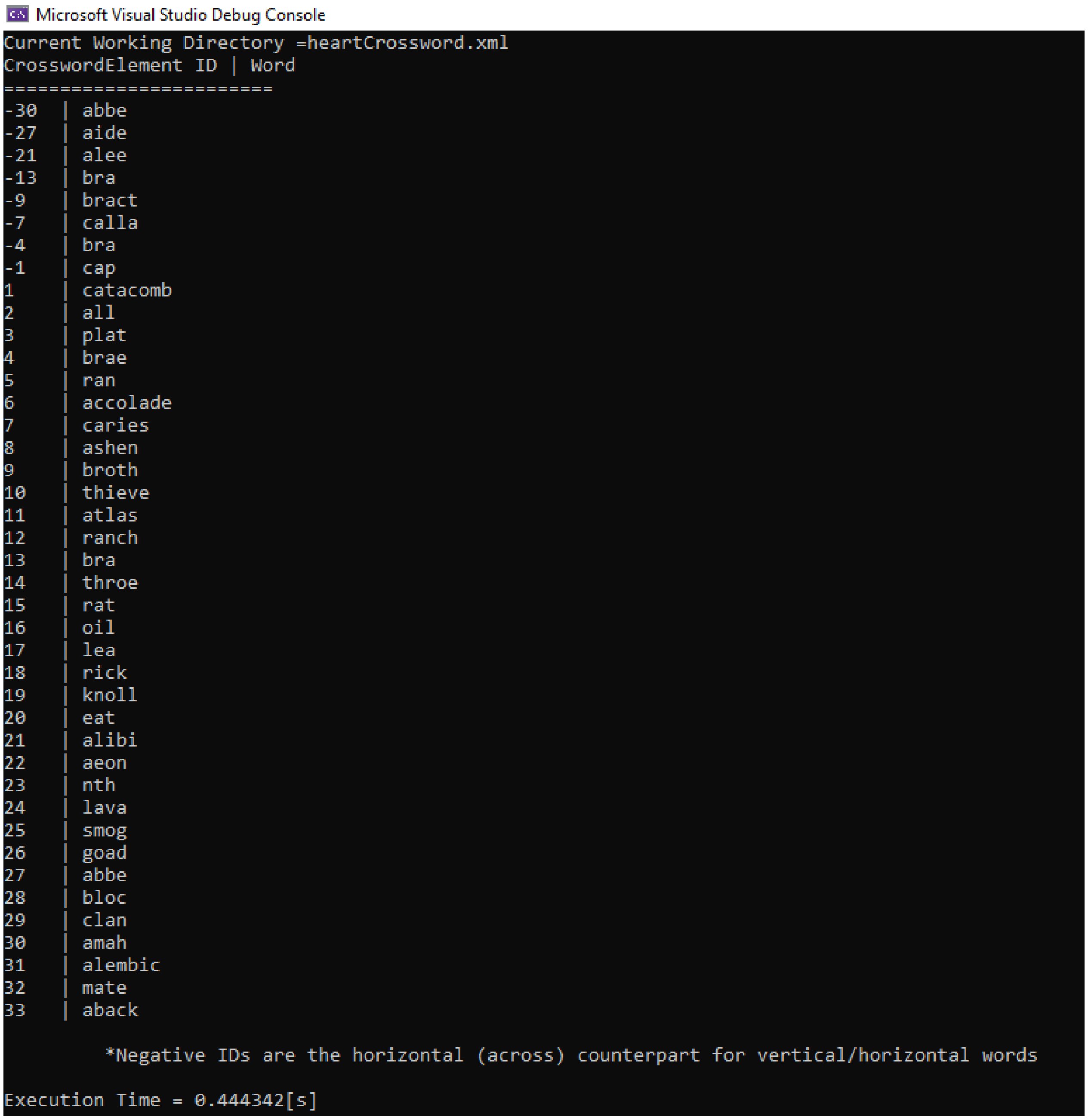


```
Current Working Directory =treeCrossword.xml
CrosswordElement ID | Word
========================
-1    | bathtub
1     | babel
2     | bloc
3     | abjure
4     | aback
5     | afield
6     | abacus
7     | ichor
8     | hackle
9     | abbe
10    | babel
11    | abacus

         *Negative IDs are the horizontal (across) counterpart for vertical/horizontal words

Execution Time = 0.005658[s]
```

Figure 4. Task 2 Crossword Puzzle 1 (Tree Crossword Puzzle)



```
Microsoft Visual Studio Debug Console
Current Working Directory =heartCrossword.xml
CrosswordElement ID | Word
========================
-30   | abbe
-27   | aide
-21   | alee
-13   | bra
-9    | bract
-7    | calla
-4    | bra
-1    | cap
1     | catacomb
2     | all
3     | plat
4     | brae
5     | ran
6     | accolade
7     | caries
8     | ashen
9     | broth
10    | thieve
11    | atlas
12    | ranch
13    | bra
14    | throe
15    | rat
16    | oil
17    | lea
18    | rick
19    | knoll
20    | eat
21    | alibi
22    | aeon
23    | nth
24    | lava
25    | smog
26    | goad
27    | abbe
28    | bloc
29    | clan
30    | amah
31    | alembic
32    | mate
33    | aback

         *Negative IDs are the horizontal (across) counterpart for vertical/horizontal words

Execution Time = 0.444342[s]
```

Figure 5. Task 2 Crossword Puzzle 2 (Heart Crossword Puzzle)