UNIVERSITY OF HULL

Data Structure Analysis-Resit

08143 Advance programming

Ieaun Roberts

2 August 2018

This document contains results and reviews found while implementing alternative data structures to solve word search puzzles. Multiple methods to store grid structures and dictionaries for operational use can be found here in along with observations on each methods efficiency.

Contents

Introduction

A data structure can be defined as a storage format that can be used to allow the creator to store, access and modify different types of data in completely different ways which comparatively may seem either more efficient, a lesser strain on hardware or better suited for a specific task.

For this assessment the goal was to implement 4 different data structures. Namely these are Vectors, 2 dimensional Arrays, Linked-Lists and Binary Trees. Each structure is different in its own way but building upon the last in practicality.

In the document i report back my findings regarding the relationships and efficiency between the different data structures.

Structures

Overview

In this project Source code was defined for 3 of the 4 data structures outlined in the ACW specifications. The 3 Successfully implemented structures are:

1. Simple Dictionary Structure
2. Simple Grid Structure
3. Advanced Grid Structure

Bellow each structures operation is broken down and discussed.

Simple

Simple Dictionary Structure

In order to populate our Dictionary array we must read in each item from the "Dictionary.txt" text file. Initially a vector is created to store an unsorted dictionary, a new IfStream called (Fin) is created to stream data from the text file and a variable called (J) is created for use in sorting of dictionary items.

The ifStream object is then given the dictionary name through variable (dictionaryName) and opens it. Now the program enters a loop which ends when either:

- End of Dictionary file is reached
- Max amount of words (30) have been added to the dictionary

In the aforementioned loop each dictionary entry is obtained using the 'getline' function and added to our dictionary array which is a global variable called (Dictionary), here the loop variable (I) is used to add each word into position (i) in the dictionary array. That dictionary item is then also added into the dictionary vector and sorted alphabetically. After each word is added the global variable (MaxWords) which indicated how many words are in the dictionary is incremented.

After all words are added to the sorted vector they are then reinserted into the (Dictionary) array and the stream is closed.

```
void WordSearch::ReadSimpleDictionary() {
    vector<string> Dictionaryarray;
    int j = 0;
    ifstream Fin;
    Fin.open(dictionaryName);
    if (Fin.fail())
    { cerr << "Cannot open simple dictionary file" << endl;}
    else
    {
        for (int i = 0; i < 30; i++)
        {
            if (Fin.eof())
            {
                break;
            }
            else
            {
                getline(Fin, Dictionary[i]);
                Dictionaryarray.push_back(Dictionary[i]);
                sort(Dictionaryarray.begin(), Dictionaryarray.end());          //sorts vector array
                MaxWords++;
            }
        }
        for (vector<string>::const_iterator i = Dictionaryarray.begin(); i != Dictionaryarray.end(); ++i)
        {
            Dictionary[j] = *i;
            j++;
        }
    }
   Fin.close();
}
```

Simple Grid Structure

In order to populate our 2 Dimensional character array for the simple grid named (simplegrid) we create a new stream object called (Fin) to store the puzzle contents which is given the name of the Puzzle to open using (puzzleName). A String is created to store the first item in the grid file which while be the size of the grid (gridsize) and a Char (c) to ensure only valid letters are added to the (simplegrid) array.

```
void WordSearch::ReadSimplePuzzle() {
    ifstream fin;
    fin.open(puzzleName);
    string str;
    char c;
    if (fin)
    {
        getline(fin, str);
        gridsize = atoi(str.c_str());
        for (int Rows = 0; Rows < gridsize; Rows++)
        {
            for (int Cols = 0; Cols < gridsize;)
            {
                fin.get(c);
                if (c != ' ' && c != '\n')
                {
                    simpleGrid[Rows][Cols] = c;
                    Cols++;
                }
            }
        }
    }
    else
    {
        cout << "Error while oppening simple file" << endl;
    }
    fin.close();
}
```

An if statement is used to clarify that the text file exists and then the grid size is obtained. A loop now begins which uses (c) to add all letter items that are not an empty space or the end of a line to the (simplegrid) array. Variables (Rows) and (Cols) are used to insert the characters in the correct positions in the array.

After completion of the loop the stream is then closed.

Solve Puzzle Simple

The SolvePuzzleSimple() method uses variables (row) and (col) to cycle through the 2D (simpleGrid) array. A variable is created to hold the current grid letter by making use of (simplegrid) and (row)/(col) to give each letters position. We then enter a loop which continues until each word in the dictionary has been checked against the current letter.

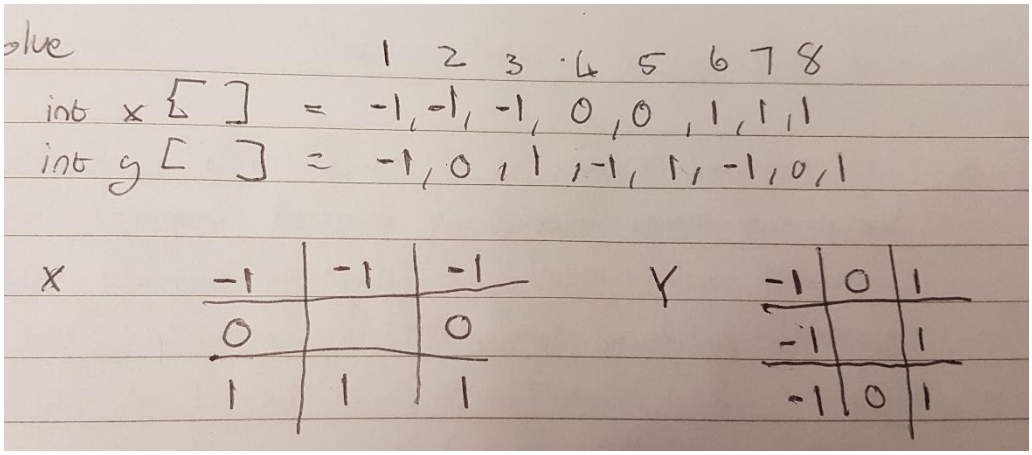
```
void WordSearch::SolvePuzzleSimple() {
    for (int row = 0; row < gridsize; row++)                // cycle through the grid rows .......
    {
        for (int col = 0; col < gridsize; col++)            // cycle through the grid cols ::::::::
        {
            char Currentletter = simpleGrid[row][col];       // sets the current letter to the letter in grid
            cellsVisited++;
            for (int WordCount = 0; WordCount < MaxWords; WordCount++)   //Cycle through the dictionary of words
            {
                DictionaryVisits++;
                string CurrentWord = Dictionary[WordCount];       // Current word equals the current word we are looking at in dictionary

                if (Currentletter < CurrentWord[0])                // if passed the possible matches
                {
                    WordCount = MaxWords;
                    break;
                }
```

```
if (Currentletter == CurrentWord[0])                    // if grid letter = to first letter of current word
{
    for (int direction = 0; direction < 8; direction++)
    {
        int rowdirection = row + x[direction], coldirection = col + y[direction];
        int LengthValidation;
        int Wordlength = CurrentWord.length();
        for (LengthValidation = 1; LengthValidation < Wordlength; LengthValidation++)
        {
            if (rowdirection >= gridsize || rowdirection < 0 || coldirection >= gridsize || coldirection < 0)
            {
                cellsVisited++;
                break;
            }
            if (simpleGrid[rowdirection][coldirection] != CurrentWord[LengthValidation])
            {
                cellsVisited++;
                break;
            }
            rowdirection += x[direction], coldirection += y[direction];
        }
        if (LengthValidation == Wordlength)                  // if all characters have been matched
        {
            MatchedCol[wordsMatched]= col;
            MatchedRow[wordsMatched] = row;
            OrderedMatchedWords[wordsMatched] = CurrentWord;
            MaxOrderedWords++;
            wordsMatched++;
            DictionaryVisits++;
            break;
```

If a grid letter matches the first letter of a dictionary entry then we enter the matching loop which searches in all 8 directions of the grid for a match and uses global variables (x) and (Y) for directional traversals.

```
// For searching in all 8 direction
int x[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
int y[] = { -1, 0, 1, -1, 1, -1, 0, 1 };
```

Where X = -1 and Y = -1 in the program that would indicated a top left grid item/ a diagonal top left traversal.



Variables (rowdirection) and (ColDirection) are used to turn (x) and (y) into valid co-ordinates in the array to search for matches. Two if statements are used to break if out of bounds of grid or if a letter does not match with a word in the dictionary.

If the loop is allowed to continue then the direction is held and the grid continues being traversed in that direction until either a match for a word in the dictionary is found or one of the aforementioned if statements breaks out of the loop indicating the sequence isn't a word in the dictionary.

If a match is found the appropriate global variables record the co-ordinates of the match and add it to a global variable which compares the matched words against the dictionary to find any unmatched words in the dictionary.

Advanced

Advanced Grid Structure

```
void WordSearch::ReadAdvancedPuzzle() {
    ifstream fin;
    fin.open(puzzleName);
    string str;
    char c;
    LLNode* TempAbove = nullptr;   // Stores the temporary value held above the current node
    LLNode* TempAbovePrevious= nullptr;   // Stores the previous temp above value
    LLNode* Temp = nullptr;       //holds current row location of root head
    if (fin)
    {
        getline(fin, str);
        gridsize = atoi(str.c_str());                //Size of the grid (e.g., 15)
        for (int Rows = 0; Rows < gridsize; Rows++)   ////......
        {
            for (int Cols = 0; Cols < gridsize;)   /////::::::::
            {
                fin.get(c);
                if (c != ' ' && c != '\n')
                {
                    struct LLNode* newNode = new LLNode;                    //(struct LLNode*)malloc(sizeof(struct LLNode));
                    if(headPtr== nullptr)  //if first letter
                    {
                        headPtr = newNode;
                        RootHead = headPtr;   // stores the head address
                        newNode->Data = c;
                        Temp = headPtr;
                    }
                    else if (TempAbove== nullptr)  //if first row of grid
                    {
                        newNode->Data = c;         //set data to letter in grid
                        headPtr->Right = newNode;   //link last item to this item
                        newNode->Left = headPtr;  //link this item to last
                        headPtr = newNode;         // move head to this node
                        if (Cols == gridsize-1)      // if at last col
                        {
                            TempAbove = Temp;
                        }
                    }
                    else if (TempAbove != nullptr)
                    {
                        newNode->Data = c;                              // (1  Data Link )
                        TempAbove->Down = newNode;
```

The advanced grid stores all the grid letters in a multi-linked lists structure. After opening the text file using a similar method to the simple structure each letter is obtained from the file. For each letter a data structure is created which contains the letter and links to all 8 of its neighbour locations which are initialized as null.

```
struct LLNode          // Stores the
{
    char Data;
    LLNode* Left = nullptr;
    LLNode* Right= nullptr;
    LLNode* Up = nullptr;
    LLNode* Down = nullptr;
    LLNode* topLeft = nullptr;
    LLNode* topRight = nullptr;
    LLNode* bottomLeft = nullptr;
    LLNode* bottomRight = nullptr;
};
LLNode* headPtr;       // stores the
LLNode* RootHead;      // stores the
```

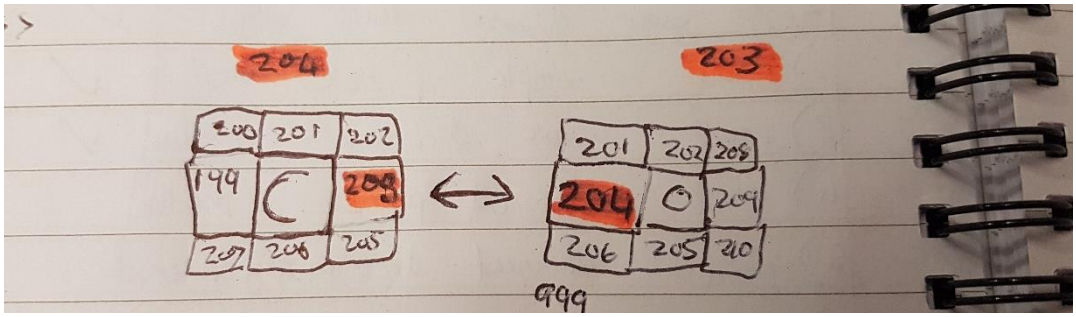
(Headptr) contains the address for the current letter we are using (if first letter in grid) or the previous letter. (Roothead) contains the location of the first letter of the grid/the beginning of the list.

Other variables are also created to hold addresses to structures that contain the letters structure above the current letter (TempAbove), above and to the left of the current letter (TempAbovePrevious) and the address of the first letter of each row (Temp).A graphical explanation of these variables is shown below.

RootHead	B	c	d	E
Temp	G	TempAbovePrevious	TempAbove	j
k	l	HeadPtr	NewNode	o

Various if statement determine the specific conditions for each letters link addresses,

e.g. The first grid letter will not be able to link to any of the other letters in the grid on creation but letter g in the table shown above would be able to link its “up” (Tempabove), ”Topleft” (Tempaboveprevious)/(Roothead), ”left” (Headptr) and “right” (TempAbovePrevious-> right) on creation.



[Each letter structure contains 8 links to adjacent grid positions, here the link between C and O is shown]

Solve Puzzle Advanced

The SolvePuzzleAdvanced() method makes use of the ReadAdvancedPuzzle() and ReadSimpleDictionary() methods. As with the SolvePuzzleSimple() method here we also begin with 2 nested for loops, which increment the row and column values on completion of a single loop.

```
void WordSearch::SolvePuzzleAdvanced() {
    LLNode * ColPosHolder = nullptr;                                    // holds the current position of the first item of each col
    int directionRow = 0;                                       // ints that change according to the direction of a sequence matched
    int directionCol = 0;
    for (int row = 0; row < gridsize; row++)                    // cycle through the grid rows .......
    {
        for (int col = 0; col < gridsize; col++)                // cycle through the grid cols ::::::::
        {
            char CurrentletterCycle = RootHead->Data;           // sets the current letter to the letter in grid
            for (int WordCount = 0; WordCount < MaxWords; WordCount++)   //Cycle through the dictionary of words
            {
                string CurrentWord = Dictionary[WordCount];        // Current word equals the current word we are looking at in dictionary
                if (col == 0)
                {
                    ColPosHolder = RootHead;            // sets it to first col position in row
                }
                if (CurrentletterCycle < CurrentWord[0])                // if passed the possible matches
                {
                    WordCount = MaxWords;
                    break;
                }
                if (CurrentletterCycle == CurrentWord[0])                    // if grid letter = to first letter of current word
                {
                    LLNode * HeadHolder = RootHead;    // pointer so that points at different direction sequences
                    directionCol = col;
                    directionRow = row;
                    for (int direction = 0; direction < 8; direction++)
                    {
```

The variable (Currentlettercycle) holds the current letter in the linked list we are using. Another variable (Currentword) is obtained from the dictionary and is used to compare the current letter to the current word.

A for loop is used to cycle throughout the dictionary, breaking if the words left in the dictionary all begin with letters after the current letter we are working with. If the current letter matches the first letter of a dictionary word then a matching process begins to attempt to match all letters to a word. As seen with the simple solve a for loop begins cycling through each of the 8 directions is used to find a matching sequence.

```
int LengthValidation;
int Wordlength = CurrentWord.length();
for (LengthValidation = 1; LengthValidation < Wordlength; LengthValidation++)
{
    switch (direction)                          //the direction the sequence is fo
    {
    case 0:
        HeadHolder = HeadHolder->topLeft;
        directionCol -= 1;
        directionRow -=  1;
        break;
    case 1:
        HeadHolder = HeadHolder->Up;
        directionRow -=  1;
        break;
    case 2:
        HeadHolder = HeadHolder->topRight;
        directionCol += 1;
        directionRow -=  1;
        break;
```

(Lengthvalidation) is used to check if the current sequence matches the word in the dictionary. The program then continues to loop until the sequence is found or breaks if either:

- Out of bounds of the grid
- Not matched to a word

Similarly to the (x) and (Y) pattern used in simple solve, (DirectionCol) and (DirectionRow) are used to determine a matching direction. E.g. where direction = 1 it will attempt to find a matching sequence using only “Up” links in the linked list.

If the sequence is matched then the appropriate variables are updated, the co-ordinates of the word are captured and the next letter/word is then searched for until the end of the list is found.

Timing and activity

Sample Grid 1 (9x9)

Words matched/unmatched in grid

Word	Found in Simple Dictionary + Simple Puzzle Grid	Found in Simple Dictionary + Advanced Puzzle Grid
Compile	✓	✓
Computer	✓	✓
Debugging	✓	✓
Hello	✓	✓
Language	✓	✓
Graphics	×	×
Loop	✓	✓
Print	✓	✓
Programme	✓	✓
World	✓	✓

```
9
E M M A R G O R P
C L U A U N L L D
O T A O F I L O I
M E U N J G E O K
P W H K G G H P Q
I C O M P U T E R
L L V R Z B A O X
E H O M L E Q G U
T N I R P D C O E
```

Results

	Simple	Advanced
Structures used	**Simple Dictionary + Simple Puzzle Grid**	**Simple Dictionary + Advanced Puzzle Grid**
Number of words matched	9	9
Number of grid cells visited	375	372
Number of dictionary Entries visited	580	580
Time to populate grid structure	0.000523549	0.00052804
Time to solve puzzle	0.000988391	0.0013124

Observations

As the methods used to store the grids are similar in an essence with regards to traversal, at a 9x9 grid size the number of cells visited is almost identical due to the compact amount of grid cells needed to visit to match the words. **Advanced searches fewer grid cells**.

As they both use the same dictionary structure they visit the exact same amount of entries when trying to match a sequence.

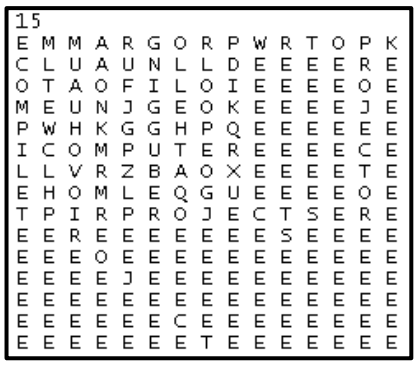
On this scale I was expecting the simple grid structure to be faster as each letter only needs to be streamed from the text file and inserted into the 2d array rather than being inserted into a linked list and then mapping all relationships after being streamed from the text file. **The simple grid populates faster but only barely (0.000003 seconds)**.

When given a smaller definable grid where the size is known before-hand it would appear that there isn't much of a difference in the time it takes for each algorithm to solve the puzzle. Seeing only a 0.004 second difference in speed with the **simple structure being slightly faster when solving the puzzles**.

Sample Grid 2 (15x15)

Words matched/unmatched in grid

Word	Found in Simple Dictionary + Simple Puzzle Grid	Found in Simple Dictionary + Advanced Puzzle Grid
Programme	✓	✓
Project	✓	✓
Projector	✓	✓
Projects	✓	✓
Zebra	✗	✗



Results

Structures used	Simple Dictionary + Simple Puzzle Grid	Simple Dictionary + Advanced Puzzle Grid
Number of words matched	6	6
Number of grid cells visited	439	307
Number of dictionary Entries visited	382	382
Time to populate grid structure	0.000426025	0.000440782
Time to solve puzzle	0.000718275	0.000668551

Observations

Both structures when given sequences to find cannot differentiate words that contain other words from each other and instead match the same word multiple times and then continue to match the word that the root word was a part of. Due to this Project was matched twice by 3 times by each algorithm leading to 6 matches from 4 different words

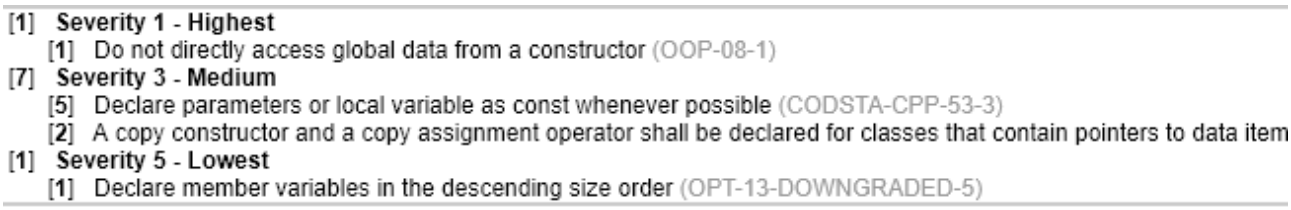
As the grid size increases the linked list data structure appears to be more efficient while breaking out of any sequence matching loops with less cell visits when compared to its simple counterpart.

Both grids populate and puzzles are solved at roughly the same amount of time.

Parasoft

Severity 3 (Declare parameters or local variables as const whenever possible)

I do not feel that this parasoft rule applies to the lines of code it has flagged as all of those variables need to be changed repeatedly as the loop progresses to change to different addresses.



Conclusion

While the results for both the simple and advanced data structures are minuscule in difference when comparing a 9x9 grid to a 15x15 grid, it is evident that the advanced puzzle grid is more efficient regarding its cell traversal of the grid. The advanced Grid structure is also only limited by the hardware running the program and can be populated by much larger amounts of data without making any adjustments to the code of this program where as the simple structure cannot and is limited to either 9x9 or code must be edited to change the 2d arrays size.

Given a larger amount of data the time between the two structures is sure to rise exponentially eventually leading to a significant increase in puzzle solving time for the advanced method. The advanced method uses less processing power and its grid is populated using dynamic memory while linking together all of its adjacent letters individually. This all allows for faster completion of puzzles.