This report presents the results of a channel advertisement picker using a self-implemented greedy method.

# Greedy Algorithm

Ayushya Amitabh
CSC 22000 – ALGORITHMS
PROF. CHI HIM “TIMMY” LIU

## Table of Contents

## THEORY

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

## IN THIS REPORT

In this report, I compare the results i.e., number of advertisements that can fit into any given channel. The ads will be sorted in four different ways:

1. After all data has been read – by timing only
2. After all data has been read – by profit values
3. While reading data – by timing only
4. While reading data – by profit value

## AFTER READING DATA

### Variables and Functions

```
static vector<int> _starts; //Contains starting times
static vector<int> _ends; //Contains ending times
static vector<int> _values; //Contains raw profit values
static vector<float> _profitValues; //Calculated [profit value / (end time - start time)]
static vector<vector<bool>> _channels; //Contains what channel each ad. can air on
static vector<int> _Ch1,_Ch2,_Ch3,_Ch4; //Positions for each channel's advertisements
static vector<int> _times1, _times2, _times3, _times4; //Final listings for each channel

//Creates vectors using data from input.txt
void read_file(ifstream& inFile);
//Sorts data based on starting times (QUICK SORT)
void sortByStart(int left = 0, int right = _starts.size()-1);
// Used by sortByStart(); Swaps elements of each vector, helps maintain order
void swapContent(int _currentPos, int _newPos);
// Indexing : Creates _Ch1, _Ch2, _Ch3, _Ch4;
void createChannels();
// Creates final listing for each channel [Contatins indexes]
void createListings(vector<int> _chNum, vector<int> &_list);
// Prints values using _Ch1,_Ch2,_Ch3,_Ch4
void printChannelAvailibity();
// Prints values using _times1, _times2, _times3, _times4
void printFinalListings();
// Creates values to be stored in _profitValues
void createProfitRatings();
// Prints all data including the values from _profitValues
void printProfitRatings();
// Creates final listing for each channel based on profit values [Contatins indexes]
void createProfitListings(vector<int> _chNum, vector<int> &_list);
```

## Main Method

```
output.open("allread-output.txt");

//"""""""""""""""GETTING DATA""""""""""""""""//
ifstream inFile;                              // Input file variable, inFile
inFile.open("input.txt");                     // Open file with name, 'input.txt'
if (inFile.fail()) {                          // Checking to make sure file was opened
  //cout << "Fail to open the file" << endl; // If failed to open file
            exit(1);                          // Prevents program from crashing...regular clean up
}                                             //
read_file(inFile);                            // Getting data from the file and creating vectors
inFile.close();                               // Close the file
//"""""""""""""DONE GETTING DATA"""""""""""""//

//"""""""""”””””""""""SORT DATA”””""""""""""""""""""//
sortByStart();       //----------------->> Sorting all data based on the starting times
//"""""""""""""""""""""DONE""""""""""""""""""""""""//

//""""""CREATE AND SHOW CHANNELS""""""//
createChannels();                    //
printChannelAvailibity();            //
//""""""""""""""""DONE""""""""""""""""//

createProfitRatings();
printProfitRatings();

//""""""CREATE AND SHOW LISTINGS""""""//
createListings(_Ch1, _times1);        //
createListings(_Ch2, _times2);        //
createListings(_Ch3, _times3);        //
createListings(_Ch4, _times4);        //
printFinalListings();                 //
//""""""""""""""""DONE""""""""""""""""//

//""""""CREATE AND SHOW LISTINGS""""""//
createProfitListings(_Ch1, _times1);  //
createProfitListings(_Ch2, _times2);  //
createProfitListings(_Ch3, _times3);  //
createProfitListings(_Ch4, _times4);  //
printFinalListings();                 //
//""""""""""""""""DONE""""""""""""""""//
```

## Reading File – void read_file(inFile);

This function reads the file and stores the data in the appropriate vectors.

```
static vector<int> _starts; //Contains starting times
static vector<int> _ends; //Contains ending times
static vector<int> _values; //Contains raw profit values
```

The implementation of this function:

```
//"""""""""READ FILE FUNCTION"""""""""//
// INPUT : ifstream & (reference)   //
void read_file(ifstream& inFile) {  //
            int start,                  Will store starting time
                        end,                Will store ending time
                        value;              Will store profit value
            bool _channel[4];               Will store True/False for each channel
```

```
        vector<bool> _channelVector(4); Vector containing values of _channel[4]
        while (!inFile.eof()) {        While the end of file is not reached
                inFile >> start          Get start time
                        >> end            Get end time
                        >> value          Get profit value
                        >> _channel[0] >> _channel[1] >> _channel[2] >> _channel[3];
                _starts.push_back(start);Starting time pushed to ** VECTOR<INT> _STARTS**
                _ends.push_back(end);    Ending time pushed to ** VECTOR<INT> _ENDS**
                _values.push_back(value);Profit values pushed to ** VECTOR<INT> _VALUES**
                for (int i = 0; i < 4; i++) _channelVector[i] = _channel[i];
                _channels.push_back(_channelVector);
                Channel availibilty pushed to **STATIC VECTOR<VECTOR<BOOL>> _CHANNELS**
        }
}
```

## Sorting Data – void sortByStart() & void swapContent()

After the channels are created, two functions:

```
void sortByStart(int left = 0, int right = _starts.size()-1);
void swapContent(int _currentPos, int _newPos);
```

work together to sort all the read data

```
//"""""""""""""SORT DATA FUNCTION""""""""""""""""//
//''''''''Uses Quick Sort algorithm''''''''//
// INPUT : int left  &  int right              //
void sortByStart(int left, int right) {        //
        int i = left, j = right;         // Left and Right ends of the array to be sorted
        int pivot = _starts[(left + right) / 2];  // Pivot used by Quick Sort
        while (i <= j) {      // As long as the left and right cursors don't cross
                while (_starts[i] < pivot) {  // Increment left cursor
                        i++;
                }
                while (_starts[j] > pivot) { // Increment right
                        j--;
                }
                if (i <= j) {               // If left cursor is less than right cursor
                        swapContent(i, j);               // Swap values at these cursors...
                        i++;                            // Increase 'i' by 1
                        j--;                            // Decrease 'j' by 1
                }
        }
        if (left < j) sortByStart(left, j); // Recursive call to partition [left half]
        if (i < right) sortByStart(i, right); // Recursive call to partition [right half]
}
//""""""""""""""""""""""""""""""""""""""""""""""""""""""""//

//""""""""""""""""""""SWAP CONTENT FUNCTION""""""""""""""""""""""//
//''''''''''''''Collectively swaps content''''''''''''''//
// INPUT : int _currentPos  &  int _newPos               //
void swapContent(int _currentPos, int _newPos) {//Swaps content of _starts,_ends,_values
        int _tmpStart, _tmpEnd, _tmpValue; // Stores temporary value
        bool _tmpChannels[4]; // Temporarily stores the array of channel listings
        _tmpStart = _starts[_newPos];
        _tmpEnd = _ends[_newPos];
        _tmpValue = _values[_newPos];
        for (int i = 0; i < 4; i++) {
                _tmpChannels[i] = _channels[_newPos][i];
        }
```

```
		_starts[_newPos] = _starts[_currentPos];		// Swaps starting time value
		_ends[_newPos] = _ends[_currentPos];			// Swaps ending time value
		_values[_newPos] = _values[_currentPos];		// Swaps profit value
		for (int i = 0; i < 4; i++) {				// Swaps channel listings
			_channels[_newPos][i] = _channels[_currentPos][i];
		}

		_starts[_currentPos] = _tmpStart;
		_ends[_currentPos] = _tmpEnd;
		_values[_currentPos] = _tmpValue;
		for (int i = 0; i < 4; i++) {
			_channels[_currentPos][i] = _tmpChannels[i];
		}

}

//""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""//
```

## Creating Channels – void createChannels();

After all of the data is read, and sorted I used the createChannels() function. This function checked which channels the ad can be aired on. This function handles:

```
static vector<int> _Ch1,_Ch2,_Ch3,_Ch4; //Positions for each channel's advertisements
```

The implementation of this function:

```
//""""""""""""CREATE CHANNEL FUNCTION""""""""""""""//
//''''''''Creates _ch1, _ch2, _ch3, _ch4''''''''''//
// INPUT : [none]					//
void createChannels() {
	for (int j = 0; j < 4; j++) {
		for (int i = 0; i < _channels.size(); i++) {
			if (_channels[i][j] == 1) {
				if (j == 0) {
					_Ch1.push_back(i);
				}
				else if (j == 1) {
					_Ch2.push_back(i);
				}
				else if (j == 2) {
					_Ch3.push_back(i);
				}
				else if (j == 3) {
					_Ch4.push_back(i);
				}
			}
		}
	}
}
//""""""""""""""""""""""""""""""""""""""""""""""""""""""""//
```

## Creating Ad Listing – Based Only On Timing – void createListings()

After channels are created, the createListings() function will handle inserting appropriate ad listings to static vector<int> _times1, _times2, _times3, _times4; //Final listings for each channel.

Implementation of the function:

```
//"""""""""""""""""""CREATE LISTINGS FUNCTION"""""""""""""""""""""//
//''''''''' Creates final listings based on timings'''''''''//
// INPUT : vector<int> _chNum  & vector<int> &_list            //
void createListings(vector<int> _chNum, vector<int> &_list) {        //
        int _lastEnd = 0;                    // Stores the value of the last ending time
        for (int i = 0; i < _chNum.size(); i++) {          // Will iterate through each
                if (_starts[_chNum[i]] >= _lastEnd) {// If starting time is after/equal
                        _list.push_back(_chNum[i]);    // Will insert index of the listing
                        _lastEnd = _ends[_chNum[i]];// Updates the value of the last ending
                }
                else {
                        //do nothing
                }
        }
}
//"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""//
```

## Creating Ad Listing – Based On Profit Values

To sort ads by their profit values, we first need to create profit ratings for all ads. The profit rating is calculated by finding the average profit per time unit, i.e.:

Profit Value = Given Value / (End Time – Start Time)

If (End Time – Start Time) = 0 ㅍ Profit Value = Given Value

Implementation of createProfitRatings():

```
//"""""""""""""""""""""""""""""""""""PROFIT RATINGS FUNCTION"""""""""""""""""""""""""""""""""""""//
//''''''''''''''''''''Creates profit values (profit / run time)''''''''''''''''''''//
// INPUT : [none]                                                  //
void createProfitRatings() {
                                                        //
        for (int i = 0; i < _starts.size() - 1; i++) {
                                        //
                if (_ends[i] - _starts[i] == 0) _profitValues.push_back(_values[i]);
                else _profitValues.push_back((_values[i] * 1.0) / (_ends[i] - _starts[i]));
        }
}
//""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""//
```

Now we are ready to create channel listings based on the profit ratings. The function createProfitListings() checks the following conditions:

1. If the start time of the ***to-be inserted*** ad is before the end time of an already existing ad – check if it has a greater profit value, if so, replace the already existing ad with the to-be inserted ad.
2. If the ad only takes up one unit of time (Example: Start time = 3 and End time = 3), check if the ***to-be inserted*** ad is not conflicting with an ad that

also takes up the same one-unit time. If it does conflict, check for the greater profit value, keep the greater profit ad.

Since the ads have already been sorted, there are only these are the only two conditions to worry about. Implementation of the function:

```
//""""""""""""""""""""""CREATE LISTINGS FUNCTION""""""""""""""""""""""//
//''''''''''''''''''''''''Creates listings based on profit values''''''''''''''''''''//
// INPUT : vector<int> _chNum & vector<int> &_list                                        //
void createProfitListings(vector<int> _chNum, vector<int> &_list) {
    _list.clear();
    int _lastEnd = 0;
    for (int i = 0; i < _chNum.size() - 1; i++) {
        if (_starts[_chNum[i]] < _lastEnd) {
            int checkIndex = 0;
            if ((i - 1) < 0) { checkIndex = 0; }
            else { checkIndex = i - 1; }
            if (_profitValues[_chNum[i]] > _profitValues[_chNum[checkIndex]]) {
                _list.back() = _chNum[i];
                _lastEnd = _ends[_chNum[i]];
            }
        }
        else if (_starts[_chNum[i]] >= _lastEnd) {
            if (_list.size() == 0) {
                cout << "Size is 0, inserting..." << endl;
                _list.push_back(_chNum[i]);
                _lastEnd = _ends[_chNum[i]];
            }
            else {
                bool none = true;
                if (_ends[_chNum[i]] == _lastEnd) {
                    for (int k = _list.size() - 1; k > 0; k--) {
                        if (_starts[_list[k]] == _lastEnd) {
                            if (_profitValues[_chNum[i]] > _profitValues[_list[k]]) {
                                none = false;
                                _lastEnd = _ends[_chNum[i]];
                                _list[k] = _chNum[i];
                                break;
                            }
                            else {
                                none = false;
                                break;
                            }
                        }
                    }
                }
                if (none) {
                    _list.push_back(_chNum[i]);
                    _lastEnd = _ends[_chNum[i]];
                }
            }
        }
        else {
            //do nothing
        }
    }
}
//""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""//
```

## After Reading Data – Results

### Only based on timings:

**<u>Channel 1:</u>** 52 **<u>Channel 2:</u>** 62 **<u>Channel 3:</u>** 59 **<u>Channel 4:</u>** 51

### Based on profit ratings

**<u>Channel 1:</u>** 202 **<u>Channel 2:</u>** 180 **<u>Channel 3:</u>** 209 **<u>Channel 4:</u>** 178

## WHILE READING DATA

### Variables and Functions

There is only one function that deals with everything because data needs to processed as it is read.

```
// VOID CREATE CHANNEL --------------------------------------------------------------//
// INPUT: vector<vector<int>> &channel, int start, int end, int value, bool byProfit
//             ^ channel to create        || ^ start || ^ end || ^ profit|| ^ sort by profit?
void createChannel(vector<vector<int>> &channel, int start, int end, int value, bool byProfit = false)
//------------------------------------------------------------------------------------//
vector<vector<int>> _ch1, _ch2, _ch3, _ch4; // Vector with listings for each channel
ofstream output; // Output file for the data
ifstream input; // Input file to read the data
```

### Main Method

```
int main() {
	vector<vector<int>> _ch1, _ch2, _ch3, _ch4;
	ofstream output;
	output.open("whileread-wo-profit-output.txt");
	ifstream input;
	input.open("input.txt");
	if (input.fail()) {
		output << "Fail to open the file" << endl;
		exit(1);
	}
	while (!input.eof()) {
		int start, end, value,
			_end1 = 0, _end2 = 0,
			_end3 = 0, _end4 = 0;
		bool _channel[4];
		input >> start >> end >> value
			>> _channel[0] >> _channel[1]
			>> _channel[2] >> _channel[3];
		if (_channel[0]) {
			createChannel(_ch1, start, end, value);
		}
		if (_channel[1]) {
			createChannel(_ch2, start, end, value);
		}
		if (_channel[2]) {
			createChannel(_ch3, start, end, value);
		}
		if (_channel[3]) {
			createChannel(_ch4, start, end, value);
		}
```

```
        }
        return 0;
}
```

## The Process

Following is a text flow (pseudo-code) of how the `void createChannel()` function handles the data.

```
If the size of the channel is 0:
        Insert this ad
Else:
        Compare to every ad that already exists for:
                If to-be ad's starting time is < ending time:
                        Compare profit values, and insert accordingly
                If to-be ad's ending time is < starting time:
                        Compare profit values, and insert accordingly
                If to-be ad's time surrounds existing ad's time:
                        Compare profit values, and insert accordingly
```

The above process is done when given either of these two cases:

1. The ***to-be inserted*** ad is after already existing ad

**OR**

2. The ***to-be inserted*** ad is before already existing ad

## The Code

```
void createChannel(vector<vector<int>> &channel, int start, int end, int value, bool byProfit =
false) {
   if (channel.size() == 0) {
      vector<int> _temp;
      _temp.push_back(start);
      _temp.push_back(end);
      _temp.push_back(value);
      channel.push_back(_temp);
   }
   else {
      for (int i = 0; i < channel.size(); i++) {
         if (start >= channel[i][1]) {
            bool none = true;
            for (int j = 0; j < channel.size(); j++) {
               if ((end > channel[j][0]) && (end < channel[j][1])) {
                  if (byProfit) {
                     float currProfit = 0, compProfit = 0;
                     if ((end - start) == 0) currProfit = value;
                     else currProfit = value / (end - start);
                     if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
                     else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
                     if (currProfit > compProfit) {
                        vector<int> _temp;
                        _temp.push_back(start);
                        _temp.push_back(end);
                        _temp.push_back(value);
                        channel[j].swap(_temp);
                        none = false;
                        break;
                     }
                     else {
```

```
            none = false;
            break;
        }
    }
    else {
        none = false;
        break;
    }
}
else
{

}
if ((channel[j][0] < start) && (channel[j][1] > start)) {
    if (byProfit) {
        float currProfit = 0, compProfit = 0;
        if ((end - start) == 0) currProfit = value;
        else currProfit = value / (end - start);
        if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
        else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
        if (currProfit > compProfit) {
            vector<int> _temp;
            _temp.push_back(start);
            _temp.push_back(end);
            _temp.push_back(value);
            channel[j].swap(_temp);
            none = false;
            break;
        }
    }
    else {
        none = false;
        break;
    }
}
else
{
}
if (start <= channel[j][0] && end >= channel[j][0]) {
    if (byProfit) {
        float currProfit = 0, compProfit = 0;
        if ((end - start) == 0) currProfit = value;
        else currProfit = value / (end - start);
        if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
        else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
        if (currProfit > compProfit) {
            vector<int> _temp;
            _temp.push_back(start);
            _temp.push_back(end);
            _temp.push_back(value);
            channel[j].swap(_temp);
            none = false;
            break;
        }
    }
    else {
        none = false;
        break;
```

```
            }
        }
        else
        {
        }
    }
    if (none) {
        vector<int> _temp;
        _temp.push_back(start);
        _temp.push_back(end);
        _temp.push_back(value);
        channel.push_back(_temp);
    }
    break;
}

            else if (end <= channel[i][0]) {

    bool none = true;
    for (int j = 0; j < channel.size(); j++) {
        if ((end > channel[j][0]) && (end < channel[j][1])) {
            if (byProfit) {
                float currProfit = 0, compProfit = 0;
                if ((end - start) == 0) currProfit = value;
                else currProfit = value / (end - start);
                if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
                else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
                if (currProfit > compProfit) {
                    vector<int> _temp;
                    _temp.push_back(start);
                    _temp.push_back(end);
                    _temp.push_back(value);
                    channel[j].swap(_temp);
                    none = false;
                    break;
                }
                else {
                    none = false;
                    break;
                }
            }
            else {
                none = false;
                break;
            }
        }
        else
        {
        }
        if ((channel[j][0] < start) && (channel[j][1] > start)) {
            if (byProfit) {
                float currProfit = 0, compProfit = 0;
                if ((end - start) == 0) currProfit = value;
                else currProfit = value / (end - start);
                if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
                else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
                if (currProfit > compProfit) {
                    vector<int> _temp;
                    _temp.push_back(start);
```

```
                                _temp.push_back(end);
                                _temp.push_back(value);
                                channel[j].swap(_temp);
                                none = false;
                                break;
                            }
                        }
                        else {
                            none = false;
                            break;
                        }
                    }
                    else
                    {
                    }
                    if (start <= channel[j][0] && end >= channel[j][0]) {
                        if (byProfit) {
                            float currProfit = 0, compProfit = 0;
                            if ((end - start) == 0) currProfit = value;
                            else currProfit = value / (end - start);
                            if ((channel[j][1] - channel[j][0]) == 0) compProfit = channel[j][2];
                            else compProfit = channel[j][2] / (channel[j][1] - channel[j][0]);
                            if (currProfit > compProfit) {
                                vector<int> _temp;
                                _temp.push_back(start);
                                _temp.push_back(end);
                                _temp.push_back(value);
                                channel[j].swap(_temp);
                                none = false;
                                break;
                            }
                        }
                        else {
                            none = false;
                            break;
                        }
                    }
                    else
                    {
                    }
                }
                if (none) {
                    vector<int> _temp;
                    _temp.push_back(start);
                    _temp.push_back(end);
                    _temp.push_back(value);
                    channel.push_back(_temp);
                }
                break;
            }
        }
    }
}
```

## The Result

### Only based on timings

**<u>Channel 1:</u>** 111 **<u>Channel 2:</u>** 111 **<u>Channel 3:</u>** 112 **<u>Channel 4:</u>** 101

### Based on profit ratings

**Channel 1:** 213 **Channel 2:** 224 **Channel 3:** 215 **Channel 4:** 219

## RESULT ANALYSIS

Creating listings based on profit ratings consistently showed a greater number of ads for each channel. I think this is true because the best ads have these two properties – less running time and greater profit value as a result, the best ads took less time and allowed for insertion of more ads. We could have increased the number of ads inserted – based on only timings – if ads were sorted based on two factors, the starting and ending time. This would require using bucket data structures, where each starting time would have its own bucket and then the buckets would be sorted based on their ending times.