Authors:

Brian Mitzel	893038547
Sorasit Wanichpan	897260477
Abeer Salam	899423594

Course:

CPSC 335

Assignment:

Project 2

Date:

March 21, 2014

I. Traveling Salesperson Problem

A. The *traveling salesperson* problem is:

input: a weighted graph $G=(V,E)$ where each edge $e \in E$ has a weight w_e
output: a Hamiltonian cycle of minimum total weight; or None if no such cycle exists
size: $n=|V|$, $m=|E|$

We will use the *tsp* algorithm given in Section 4.16 as a template to find a Hamiltonian cycle of minimum total weight for a weighted graph *G*. To generate the candidates, we will modify the algorithm to use lazy permutation to avoid space limitations.

Draft 1: Lazy TSP exhaustive optimization algorithm:

```
def lazy_tsp(G):
    best = None
        for path in <GENERATE CANDIDATES LAZILY>(G.get_vertices()):
            cycle = path + [path[0]]
            if verify_tsp(G, cycle):
                if best is None or cycle_weight(cycle) < cycle_weight(best):
                    best = cycle
    return best

def cycle_weight(G, cycle):
    total = 0
    for i in range(len(cycle)-1):
        total += G.edge_weight(cycle[i], cycle[i+1])
    return total

def verify_tsp(G, cycle):
    for i in range (len(cycle)-1):
        if not G.contains_edge(cycle[i], cycle[i+1]):
            return False
    return True
```

Utilizing the candidate generation module included with the project files, we will instantiate the permutation factory and generate candidates based upon the total number of vertices in *G*. The factory will generate a permutation candidate one at a time.

Draft 2: Lazy TSP exhaustive optimization algorithm:

```
def lazy_tsp(G):
    best = None

    #Instantiate the lazy permutation factory
    factory = PermutationFactory(G.number_of_vertices())

    #Lazily generates permutations
    while factory.has_next():
        perm = factory.next() #Permutation Candidate
        cycle = perm + [perm[0]] #Close the cycle, first-last vertex

        if verify_tsp(G, cycle):
            if best is None or cycle_weight(cycle) < cycle_weight(best):
                best = cycle

    return best

```

```python
def cycle_weight(G, cycle):
    total = 0
    for i in range(len(cycle)-1):
        total += G.edge_weight(cycle[i], cycle[i+1])
    return total

def verify_tsp(G, cycle):
    for i in range (len(cycle)-1):
        if not G.contains_edge(cycle[i], cycle[i+1]):
            return False
    return True
```

B. TSP Python Source Code

Traveling Salesman Problem

```
#Brian Mitzel       893038547
#Sorasit Wanichpan 897260477
#Abeer Salam        899423594
#CPSC 335
#Project 2 v1
#python3 tsp.py <weighted_graph.xml.zip>

import sys
import time
import tsplib
import candidate

#TSP Algorithm from the lecture notes, modified to use lazy permutation over
eager
def tsp_algo(weighted_graph):
    best = None

    #Generates all the permutations of the list
    factory =
candidate.PermutationFactory(list(range(0,weighted_graph.vertex_count())))

    #Find the best candidate if one exists
    while factory.has_next():
        perm = factory.next()                    #Next permutation
        cycle = perm + [perm[0]]                 #Path of distinct vertices/closed
off by duplicating first vertex
        if verify_tsp(weighted_graph, cycle):   #Verifier
            if best is None or cycle_weight(weighted_graph, cycle) <
cycle_weight(weighted_graph, best):
                best = cycle

    #return the best Hamiltonian cycle candidate
    return best

#From the lecture notes
def cycle_weight(graph, cycle):
    total = 0
    for i in range(len(cycle)-1):
        total += graph.distance(cycle[i], cycle[i+1])
    return total

#From the lecture notes
def verify_tsp(graph, cycle):
    for i in range(len(cycle)-1):
        if not graph.is_edge(cycle[i], cycle[i+1]):
            return False
    return True

def main():
    #Verify the correct number of command line arguments were used
    if len(sys.argv) != 2:
        print('error: you must supply exactly one arguments\n\n' +
              'usage: python3 tsp.py <weighted_graph.xml.zip file>')
        sys.exit(1)

    #Capture the command line arguments
    weighted_graph = sys.argv[1]
```

```
    print('TSP Instance:')                #Get file input
    tspfile = tsplib.load(weighted_graph)#Load the TSP file based upon user input
    print("n = ", tspfile.vertex_count())#Print out the # of vertices
    start = time.perf_counter()          #Get beginning time
    result = tsp_algo(tspfile)                        #Find the Hamiltonian cycle
    end = time.perf_counter()                         #Get end time
    cost = cycle_weight(tspfile, result)              #Compute the cost

    #Prints out the result
    print('Elapsed time: ' + str(end - start))
    print('Optimal Cycle: ' + str(result))
    print('Optimal Cost: ' + str(cost))

if __name__ == "__main__":
    main()
```

C. TSP Program Output

For *TSP* instance: clique4.xml.zip

```
>python tsp.py clique4.xml.zip
TSP Instance:
n =  4
Elapsed time: 0.0028451025504184655
Optimal Cycle: [0, 3, 1, 2, 0]
Optimal Cost: 12.0
```

For *TSP* instance: burma14.xml.zip*

```
>python tsp.py burma14.xml.zip
TSP Instance:
n =  14
<break>
```

**Did not finish in 5 minutes*

For *TSP* instance: br17.xml.zip*

```
>python tsp.py br17.xml.zip
TSP Instance:
n =  17
<break>
```

**Did not finish in 5 minutes*

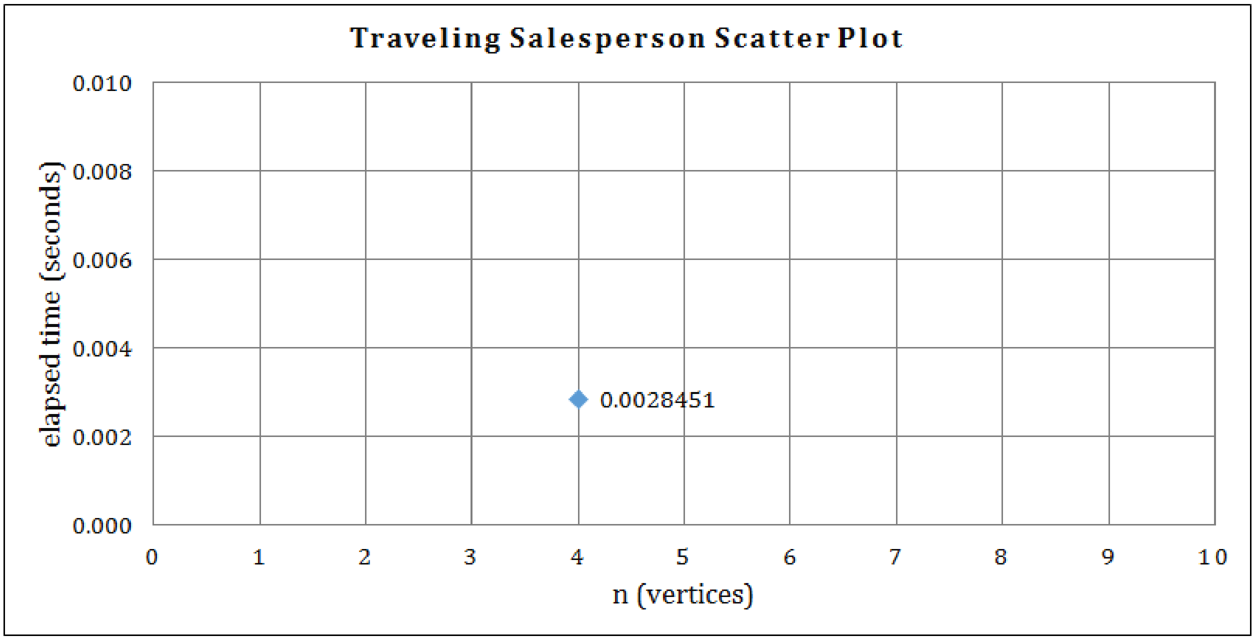
For *TSP* instance: gr17.xml.zip*

```
>python tsp.py gr17.xml.zip
TSP Instance:
n =  17
<break>
```

**Did not finish in 5 minutes.*

We were unable to find any other TSP instance that could be completed under five minutes.

D. TSP Scatter Plot



II. Longest Common Subsequence Problem

A. The *longest common subsequence* problem is:

input: two strings L, R of length n_L, n_R respectively
output: the longest string B such that B is a subsequence of L and also a subsequence of R
size: n_L, n_R

We will use an exhaustive optimization algorithm to find the optimal (longest) common subsequence of L and R. We will generate candidate subsequences lazily in order to avoid space limitations.

Draft 1: Lazy LCS exhaustive optimization algorithm:

```
def lcs(L, R):
	B = “”
	factory = <CANDIDATE FACTORY TYPE>(L, R)

	while factory.has_next():
		candidate = factory.next()
		if <VERIFIER>(L, R, candidate):
			if (B is “” or
					<candidate IS BETTER THAN B>):
				B = candidate

	return B
```

We can use a factory to lazily generate valid subsequences of L and then use a verifier to verify which of those are also subsequences of R. Therefore, we do not need to pass R as an argument to the factory constructor; likewise, we do not need to pass L as an argument to the verifier.

Draft 2: Lazy LCS exhaustive optimization algorithm:

```
def lcs(L, R):
	B = “”
	factory = <CANDIDATE FACTORY TYPE>(L)

	while factory.has_next():
		candidate = factory.next()
		if <VERIFIER>(R, candidate):
			if (B is “” or
					<candidate IS BETTER THAN B>):
				B = candidate

	return B
```

The verifier returns `True` if the candidate subsequence from L is also a subsequence of R, or `False` otherwise.

Subsequence verifier algorithm:

```
def <VERIFIER>(R, candidate):
    initialize the next index to the beginning of R

    for each letter in candidate:
        initialize r to the next index of R
        repeat from r to the end of R
            while letter is not equal to R[r]:
                r = r + 1

        if r < length(R):           #Match found
            increment the next starting index
        else:                       #Match not found
            return False

    return True
```

Using the above algorithms, we have the following pseudocode.

LCS pseudocode:

```
def lcs(L, R):
    B = “”

    #Convert L from a string to a list, and
    #generate all subsets (i.e.: subsequences) of the list
    factory = SubsetFactory(list(L))

    while factory.has_next():
        subsequence = factory.next()
        if verify_subsequence(R, subsequence):
            if (B is “” or
                        length(subsequence) > length(B)):
                    B = to_string(subsequence)

    return B
```

One final optimization can be made to the LCS pseudocode. Currently, our pseudo code converts each common subsequence of L and R that is found into a string before assigning it to B. In order to reduce the operating time, we can instead initially declare B as a null object and assign each common subsequence to it directly. This way, we do not need to convert each common subsequence to a string. Instead, we can simply convert B to a string once before we return it.

Revised LCS pseudocode:

```
def lcs(L, R):
	B = None

	#Convert L from a string to a list, and
	#generate all subsets (i.e.: subsequences) of the list
	factory = SubsetFactory(list(L))

	while factory.has_next():
		subsequence = factory.next()
		if verify_subsequence(R, subsequence):
			if (B is None or
					length(subsequence) > length(B)):
				B = subsequence

	return to_string(B)
```

Subsequence verifier pseudocode:

```
def verify_subsequence(string_to_match, candidate_subsequence):
	next = 0

	for letter in candidate_subsequence:
		r = next
		while (r < length(string_to_match) and
				letter != string_to_match[r]):
			r = r + 1

	if r < length(string_to_match):		#Match found
		next = r + 1
	else:					#Match not found
		return False

	return True
```

B. LCS Python Source Code:

Longest Common Subsequence:

```
#Brian Mitzel       893038547
#Sorasit Wanichpan  897260477
#Abeer Salam        899423594
#CPSC 335
#Project 2

import candidate
import sys
import time

def lcs(L, R):
    B = None

    #Convert L from a string to a list, and
    #generate all subsets (i.e.: subsequences) of the list
    factory = candidate.SubsetFactory(list(L))

    #Find the best candidate, if one exists
    while factory.has_next()
        subsequence = factory.next()
        if verify_subsequence(R, subsequence):
            if (B is None or
                    len(subsequence) > len(B)):
                B = subsequence

    #Convert B to a string and return it
    return "".join(B)

def verify_subsequence(string_to_match, candidate_subsequence):
    next = 0

    #Verify that each letter in the candidate subsequence appears in order in the
    #string
    for letter in candidate_subsequence:
        r = next
        while (r < len(string_to_match) and
                letter != string_to_match[r]):
            r = r + 1

        if r < len(string_to_match):          #Match found
            next = r + 1                      #Advance to the next letter
        else:                                 #Match not found
            return False                      #The candidate is not a subsequence

    #All the letters in the candidate were matched in sequence with letters in
    #the string successfully
    #Therefore, the candidate is a subsequence of the string
    return True

def main():
    #Verify the correct number of command line arguments were used
    if len(sys.argv) != 5:
        print('error: you must supply exactly four arguments\n\n' +
              'usage: python3 lcs.py <text file L> <text file R> <n(L)> <n(R)>')
        sys.exit(1)

```

```
    #Capture the command line arguments
    fileL = sys.argv[1]
    fileR = sys.argv[2]
    nL = int(sys.argv[3])
    nR = int(sys.argv[4])

    print('LCS:')
    string_a = open(fileL).read()[:nL]                  #First input
    string_b = open(fileR).read()[:nR]                  #Second input
    assert(len(string_a) == nL)
    assert(len(string_b) == nR)
    print('  a = ' + string_a + ', length', str(len(string_a)))
    print('  b = ' + string_b + ', length', str(len(string_b)))

    start     = time.perf_counter()                     #Get start time
    result    = lcs(string_a, string_b)                 #Find the LCS
    end       = time.perf_counter()                     #Get end time

    #Display the results
    print('  Elapsed time = ' + str(end - start))
    print('  Longest Common Subsequence = "' + result + '", length', len(result))

if __name__ == "__main__":
    main()
```

C. LCS Program Output:

For $n_L = 6$, $n_R = 6$:

```
>python lcs.py instance1L.txt instance1R.txt 6 6
LCS:
  a = abazdc, length 6
  b = bacbad, length 6
  Elapsed time = 0.00042337618128369367
  Longest Common Subsequence = "abad", length 4
```

For $n_L = 11$, $n_R = 13$:

```
>python lcs.py instance2L.txt instance2R.txt 11 13
LCS:
  a = abracadabra, length 11
  b = yabbadabbadoo, length 13
  Elapsed time = 0.01466207310637519
  Longest Common Subsequence = "abadaba", length 7
```

For $n_L = 10$, $n_R = 10$:

```
>python lcs.py pg11.txt pg76.txt 10 10
LCS:
  a = Project Gu, length 10
  b = 

The Proj, length 10
  Elapsed time = 0.007093475473689523
  Longest Common Subsequence = "Proj", length 4
```

For $n_L = 15$, $n_R = 15$:

```
>python lcs.py pg11.txt pg76.txt 15 15
LCS:
  a = Project Gutenbe, length 15
  b = 

The Project G, length 15
  Elapsed time = 0.2944247771715563
  Longest Common Subsequence = "Project G", length 9
```

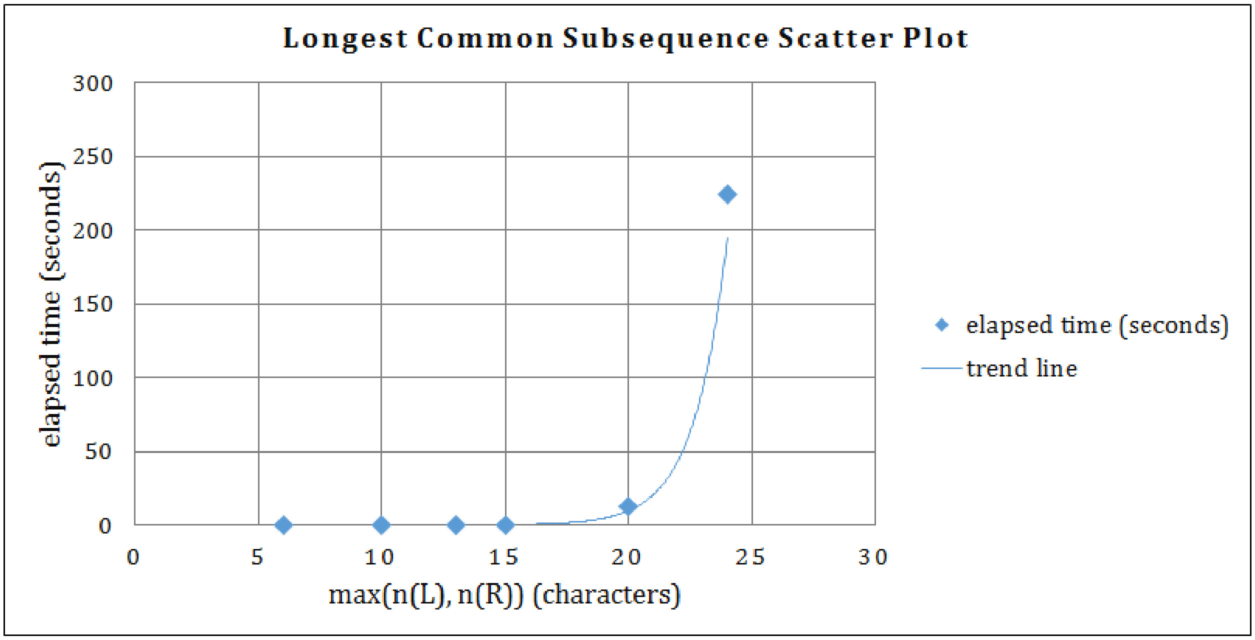
For $n_L = 20,\ n_R = 20$:

```
>python lcs.py pg11.txt pg76.txt 20 20
LCS:
  a = Project Gutenberg's , length 20
  b = 

The Project Gutenb, length 20
  Elapsed time = 12.360177664041165
  Longest Common Subsequence = "Project Gutenb", length 14
```

For $n_L = 24,\ n_R = 24$:

```
>python lcs.py pg11.txt pg76.txt 24 24
LCS:
  a = Project Gutenberg's Alic, length 24
  b = 

The Project Gutenberg , length 24
  Elapsed time = 224.37612397823077
  Longest Common Subsequence = "Project Gutenberg ", length 18
```

D. <u>LCS Scatter Plot:</u>



III. Questions

A. Describe your TSP algorithm briefly. Did you use the eager exhaustive search algorithm we covered in class, modify that algorithm slightly, or modify it extensively? If you made changes, what were they and why did you make them? What is the time complexity of your algorithm?

a. We used the TSP algorithm covered in class but modified it to use lazy permutation generation instead of eager. So, our algorithm used the provided permutation factory module to generate Hamiltonian paths for the given graph. It then converted each path into a cycle by copying and appending the first vertex to the end. We then verified that each cycle had edges connecting the vertices in the correct order. Finally, the cycle with the least weight was returned.

b. We decided that we needed to generate our permutations lazily because our original eager generation implementation encountered memory limitations for graphs with a large number of vertices. By using lazy permutation generation instead, we were able to resolve the issue with the bottleneck of memory space. Therefore, we used the permutation factory class that was included in the `candidate.py` module.

c. Our algorithm generates $n!$ permutations. During the generation of each permutation, four function calls are made to `next()`, `verify_tsp()`, and `cycle_weight()` twice, each of which runs in $O(n)$ time.

   i. Using this information, the analysis of our TSP algorithm shows us a $O(n \cdot n!)$ time. This is why our algorithm fails to complete in under 5 minutes for larger values of n. The approximate completion time for the larger problem instances estimates to about 8 to 10 hours, depending on hardware specifications.

B. What is the largest TSP instance your implementation could solve, and how long did that take?

a. The largest TSP instance our implementation could solve was the *clique4* instance provided. The size of that instance was $n = 4$ vertices. It took less than 3 thousandths of a second to complete.

C. Describe your LCS algorithm briefly. How did you generate candidate solutions? Does your algorithm generate them eagerly or lazily? What is the time complexity of your algorithm?

a. Our LCS algorithm generated subsets of the first input string, L, as candidate subsequences. These candidates were generated using the provided lazy subset factory module in order to avoid similar memory space limitations that were encountered in our TSP algorithm. Common subsequences were found by verifying that all of the characters in each candidate also appeared in the same order in the second string, R. Finally, the common subsequence with the longest number of characters was returned.

b. The time complexity of our LCS algorithm based upon our empirical analysis uses Claim 37 from the lecture notes:

    i. Claim 37 from Section 4.14 states that the factory constructor takes $O(n)$ time. Our `factory.next()` function also takes $O(n)$ time, while `factory.has_next()` takes $O(1)$ time. The factory will produce 2^n subsets of L. Finally, our `verify_subsequence()` function evaluates to $O(n)$ time.

    ii. Using this information, our LCS algorithm evaluates to $O(2^n \bullet n^2)$. Compared to our TSP algorithm, this is a more efficient algorithm, which results in the successful completion of larger test cases.

D. What is the largest LCS instance your implementation could solve, and how long did that take?

a. The largest problem instance that our LCS algorithm was able to solve in under 5 minutes was for two strings of length $n = 24$. It took 3 minutes and 44.4 seconds.

E. Which candidate generation code did you use: the provided eager generators, provided factory classes, or the generators built into itertools? Why did you choose that implementation over the alternatives?

a. For our candidate generation code, we decided to use the provided factory classes. We chose this option over the alternatives because the factory classes were covered explicitly in the lecture notes. In addition, we ran into memory space issues in our first draft of the algorithm with the eager candidate generator. By generating our candidates lazily, we were able to overcome our memory problem.

F. Did your implementations produce correct outputs? Is this evidence consistent or inconsistent with hypothesis 1?

a. Our exhaustive optimization implementations of both the Traveling Salesperson and Longest Common Subsequence algorithms did produce correct outputs. The TSP algorithm output a Hamiltonian cycle with the least minimum weight, and the LCS algorithm correctly generated the longest common subsequences of two strings. Therefore, the evidence we found is consistent with hypothesis 1, which states that exhaustive optimization algorithms produce correct outputs.

G. How would you characterize the efficiency of your implementations? Was it possible to solve realistically-sized inputs in a reasonable amount of time? Is this evidence consistent or inconsistent with hypothesis 2?

a. The efficiencies of both algorithm implementations are simply unacceptable. As discussed in class, execution time for an algorithm that takes more than five minutes to complete is considered to have taken too long for practical use. For smaller problem instances, both algorithms were able to generate solutions in a reasonable amount of time; however, as the size of the instances grew even slightly, the execution time quickly became untenable. This evidence is consistent with hypothesis 2, which states that algorithms with exponential $(O(2^n))$ and factorial $(O(n!))$ running times are probably too slow to be of practical use.