Assignment 2 - Algorithms

Liverpool University
Robert Johnson
200962268

Contents

0.1 Problem

The problem given revolves around being given a list of strings and outputting a single string that contains all of them. This could be done trivially by simply concatenating them however an added requirement of the assignment states that we must find the shortest string that contains them. There are only a few different things to consider when thinking about such a problem.

- Repeated strings
- Overlapping strings

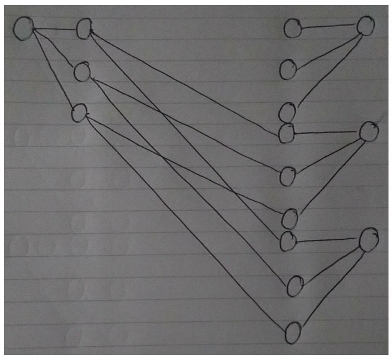
The first is easily satisfied by simply using a data structure that removes duplicates such as a set. The second point is more complex and has many possible solutions 3 of which are mentioned below.

0.2 Approaches to find overlap values

0.2.1 Naive approach (Working out maximum overlaps)

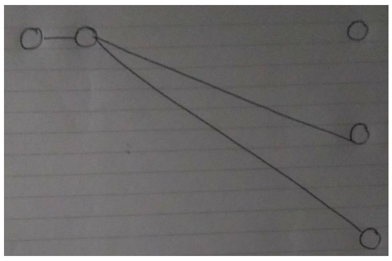
When looking at the strings given we can see they are all of the same length, because of this it makes using a sliding approach easy, where each of the words is cut at the start and end then stored separately to be compared with others. For example to compare the words abc and cab we would simply separate abc into an array with [ab,bc] and cab would be separated into [ca,ab]. Using this we can easily compare the first element in a list to the second element in all other lists.

The below example shows this process, the left most node is the word we are trying to overlap, on the right there are 3 Strings one of which is itself. Each word has m different child nodes m representing the length of each string. Using the example we can see that for each child node on the left, we need to do n comparison's, each comparison taking m time. This needs to be done for all child nodes taking nm^2 time. Finally since this is only the cost for comparing one node to all other nodes this process would need to be repeated n times, resulting in the time complexity of n^2m^2. This does not take into account the time needed to generate these child nodes which would take nm^2 time therefore the final cost of this algorithm is $n^2m^2 + nm^2$.



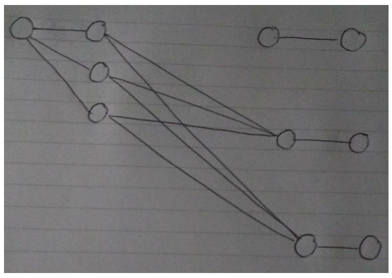
0.2.2 KMP (Knuth-morris-pratt)

Knuth-morris-pratt is an algorithm that creates a transition table which is used to allow for movement through a string to look for a specific pattern. It is used to check if a specific pattern exists within a piece of text, this algorithm can be altered to fit our purpose. Since the length the strings and pattern are the same we can state that $m = k$. Therefore creation of the transition table shown as the child of the left most node would take m time to compute, this would need to be done for all strings taking nm time. Next we do our comparisons each taking m time for n different nodes on the right this process taking nm time, this would need to be done for each string on the left n bringing our time taken to $mn + mn^2$.



0.2.3 Suffix Tree

A suffix tree is used when trying search a string for a specific pattern, it works in m time (length of pattern) and takes n time to compute. The reasoning why this may not be the optimal structure to use in this assignment is due to 2 reasons: the length of the string we are searching is the same length as the pattern therefore the time taken to make the suffix tree is the same as KMP's transition table; a suffix tree excels at finding a pattern within a piece of text rather than only matching a partial part of the pattern, therefore to use this structure optimally we would have to send the same string cut down into m different lengths to check if any of them are in the text. This is represented below where the left most node is the pattern to be searched for and the right most nodes are the raw string's, which can be turned into suffix trees in m time. Again we would need to create m different child nodes for the string on the left taking nm^2 time. Additionally we would need to make the suffix tree nodes on the right taking nm time to compute. Finally to do all comparisons we would need to compare every child node on the left with each suffix tree on the right, each comparison taking m time for nm different child nodes, resulting in an overall time complexity of n^2m^2. This puts us at a time complexity of $mn + 2n^2m^2$.



0.2.4 Comparison of all 3 methods

Time taken to run Nieve approach $= n^2m^2 + nm^2$
Time taken to run Knuth-morris-pratt approach $= mn + mn^2$
Time taken to run Suffix tree approach $= mn + 2n^2m^2$
We can see that using suffix tree's may perform better than the Nieve approach if m is large enough, however if m is small then the nieve approach may actually perform better due the suffix tree's constant. We can also see the clear winner from this comparison being Knuth-morris-pratt.

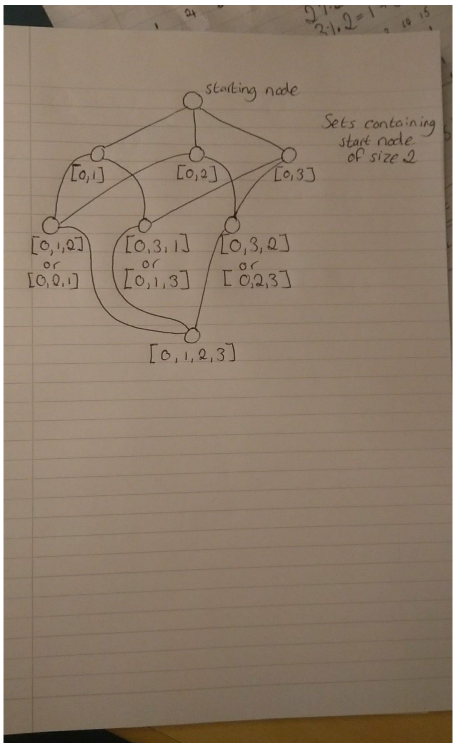
0.3 Now that we have the sizes of all overlaps now what?

0.3.1 Naive approach

Using the transition matrix it would be relatively easily to exhaustively search for the shortest path, since this graph is a clique this would take $n!$ time, but would give a optimal solution. However if we do not mind having a sub optimal solution, we could use a graph reduction algorithm that simply removes paths of maximum weights until the graphs cycling property no longer holds. This would be greedy and would not guarantee the optimal solution.

0.3.2 Held-karp algorithm

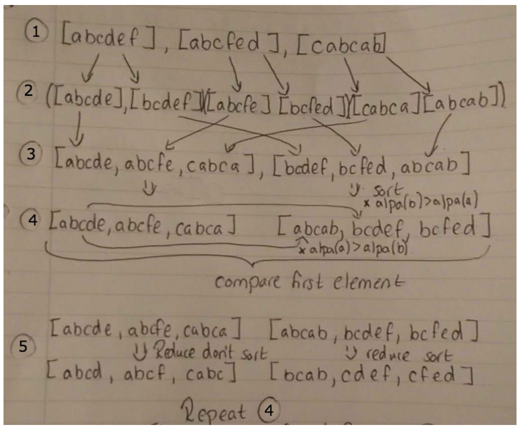
If we would like an optimal solution in a shorter time then n! we can use dynamic programming to work out optimal paths for subsets of our problem, using these to create optimal solutions for this subset plus one additional element. This is the basis around the Held-karp algorithm. As shown in the diagram below we first look at any combination of edges which include a singular "start" node as well as k-1 other nodes, initially we try to find sets of size 2. The example shows the data flow throughout the algorithm for a graph consisting of only 4 nodes, the first line representing the set only including our starting node (node 0). The second line consists of sets including node 0 as well as any other node [0,1], [0,2] and [0,3]. These sets contain the cost from moving from the added element to the current set. For example the set consisting of [0,1] would contain the cost of moving from 1 to 0 plus the cost of the set only containing 0. The next line consists of all possible sets which again must contain our starting node (node 0) and are of size 3, this would be sets [0,1,2],[0,2,1],[0,3,1],[0,1,3],[0,3,2],[0,2,3]. As you may notice the first two of these sets contain the same elements however have different ordering. The first ending in 2 the second ending in 1, in these two examples we would like to find the minimum cost path that makes up the set 0,1,2 which would either be the path [0→1](which is already computed)→2 or the path from [0→2](which is already computed)→1, because we have precomputed values we can do these calculations very quickly and store the minimum path. This repeats until we have the minimum path for the whole graph.



0.4 My implementations

0.4.1 Approximate

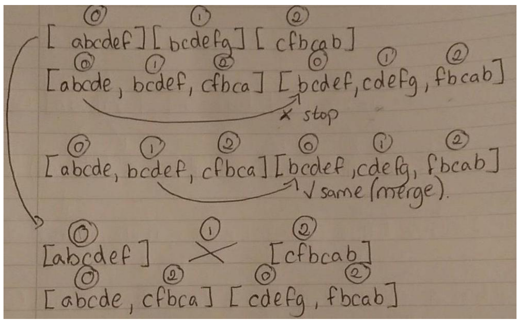
My approximate solution is based around using the naive approach of sliding windows. I first split each string into separate parts as described in 0.2.1, however I would then store the separate parts of these words in separate ordered lists these individual words would still have references to their major word and as such adding / removing from the initial strings should be easy.



① [abcdef], [abcfed], [cabcab]
② ([abcde],[bcdef])([abcfe] [bcfed])([cabca][abcab])
③ [abcde, abcfe, cabca], [bcdef, bcfed, abcab]
sort
x alpa(b)>alpa(a)
④ [abcde, abcfe, cabca] [abcab, bcdef, bcfed]
x alpa(a)>alpa(b)
compare first element
⑤ [abcde, abcfe, cabca] [abcab, bcdef, bcfed]
Reduce don't sort
reduce sort
[abcd, abcf, cabc] [bcab, cdef, cfed]
Repeat ④

- (1) Shows the initial strings within our system.
- (2) Shows the separation of strings described in 0.2.1 describing the sliding windows approach.
- (3) Shows separating these separated words into separate lists.
- (3-4) Shows the ordering of words in the right list only.
- (4-5) Using these ordered lists it is easy to make comparisons as well as end the search early, this is due to the fact that if a match is not found and the string on in the right is of greater alphanumeric value than the one on the left, then we can stop as if there was a identical word it would have already been found.
- (5) If no matches were found we simply reduce the sub words length again and repeat the process.

The above image and explanation explains the algorithms process if no matches are found however I will also explain what happens if a match is found.



[abcdef][bcdefg][cfbcab]
[abcde, bcdef, cfbca] [bcdef, cdefg, fbcab]
x stop
[abcde, bcdef, cfbca] [bcdef, cdefg, fbcab]
√ same (merge).
[abcdef] [cfbcab]
[abcde, cfbca] [cdefg, fbcab]

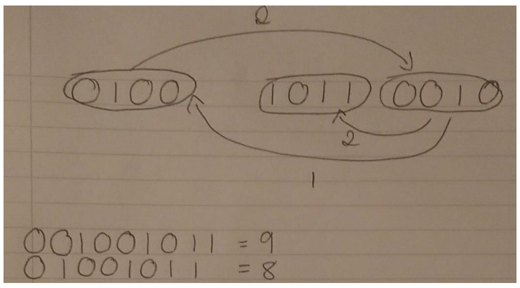
- Line 2, This shows the comparison of the first element of the left list to the right list, because the alphanumeric value is greater than the left element we are able to skip to the second element in the left list.
- Line 3, We instantly find a match so we attempt to merge, having the initial index of both sub-words we are able to merge them effortlessly, and since they were able to be merged both of

the sub-words in each list should not have changed rather their counterparts in each list should be removed, this is easily achieved since we have index values.

- Lines 4-5, Show the updated strings as well as the updated sub lists, this will continue until either both sub lists only contain 1 element, in which case all strings have been merged, or until all words are of length 0 in each sub list in which case we need to forcefully join each word left over in our set of words.

Sub-optimality

Although this solution was giving optimal super strings for the static cases provided I was able to generate certain combinations of strings which caused sub-optimal solutions, one such case is "0100", "1011" and "0010". In this case we have the following directed graph which shows overlap costs for each string where each incoming edge V(u,w) shows the amount of characters that could not be matched when attaching w to the start of u. The graph is incomplete for clarity as any edges that do not exist in the graph provided take a default weight of m. Suppose we attach words with the highest weights first "0010" and "0100" making the string "00100" however using this string it is impossible to attach the third string without adding its whole length, resulting in the string "001001011" who's length is 9. Now consider we take a different approach trying to reduce the path cost, in this case we have the path from "0100" to "0010" and finally to "1011" resulting in "01001011" which has a length of 8, because of this it is important to think of ways to compute the optimal solution, which brings be to the second method used.



0.4.2 Optimal solution using Held-karp algorithm and KMP (William Fisets implementation of Held-karp explained in detail, verifying my understanding of the code)

The implementation of the Held-Karp algorithm is based upon code written by William Fiset, william.alexandre.fiset@gmail.com and improved upon by myself in terms of space complexity.

Representation of sets by binary bits

In his solution, William Fiset uses bits to represent the inclusion of elements in a set something I had not previously thought of. Such an approach makes it easy to add, compare and remove elements from a set. For example in a set which contains the first and second element in a list but not the 3rd and 4th we could represent this in binary as the number "0011" = 3. Additionally using sets this way allows us to utilise built in operators to add |, remove &∼ and compare ∧ elements these operations take constant time. Using such a representation we are able to find out the bitwise value of a set containing all elements i.e the superset using (1 << N)-1, << being the bitwise operator to move a bit left N places.

Set cost generation

To explain his implementation I will follow his code explaining the function of each major line as well as the reasoning behind them.

- 55, Because we know the amount of nodes i.e strings we have in our graph we are able to specify the largest possible set representation when using bits using (1<<N) this returns the value of shifting the bit representation "0001" left N times so for a graph containing 4 elements return 16, this is 1 more then we need to represent the inclusion of all 4 of our elements in our set this being "1111" = 15.
- 56, Initializing memo to be of size [N][1<<N], as described initially to use Held-Karp algorithm we need to know the state we end at when creating our sets, the reasoning behind making the array N height is to make storing and accessing these end points easy, in essence the index at array 1 would only represent sequences that end in the state 1. The reasoning behind making it of width 1<<N is to contain all possible inclusion sets.
- 59:61, As explained in my illustration for the explanation of Held-karp the first iteration only works out the cost of sets only containing two elements one of which is stated to be our start point. In these lines we iterate through all possible nodes finding out the distance to move from this node to our pre-defined start point. The term $(1 << start)|(1 << end)$ simply calculating the value of the set containing the start and end states.
- 64, In this line we instantiate a integer "r" this represents the size of the sets we would like to find in our current iteration, for obvious reasons this need to increase to our maximum set size N as to find the overall path.
- 65, Combinations(r,N) this returns all binary combinations of size N, we then iterate through these combinations.
- 66, The method notIn(start, subset) returns false if the start state is not contained in the current set, if the start state is not in the current state then it is invalid as all states must start and therefore contain the start state.
- 67:68, In these lines we instantiate a variable "next" this iterates through all possible nodes, next being the state that we would like to end the current set in, additionally we would like to make sure that we do not try to end in the start state.
- 69, We remove this element from the current set essentially creating a set that we have previously computed, we would like to try to find the minimum cost to add this state onto the current reduced precomputed set.
- 70:72, We instantiate a variable used to store the minimum path found for the new set(70), this is initially infinity as if no path exists then the path should be allocated a value of infinity. We then iterate through all nodes in the current set (71), selecting states that are not the start point or the end point(72).
- 73, We calculate the cost to move from the current reduced state ending in "end" to the new state where we end in "next".
- 74:75, We update the minimal cost to make the movement from the next node to the current set if a better path is found.
- 78, We update the cost from moving to the current set where we end in the "next" state.

Minimal path cost retrieval

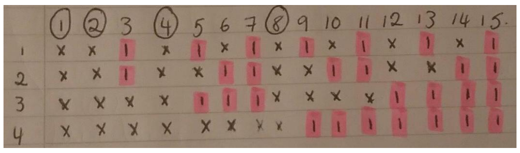
- 84:86, For all nodes as long as it isn't the node we started at, check the cost of the set ending the path at the current node plus the cost to get back to the starting node.
- 87:88, If it was smaller than any found previously, update the minimum tour cost.

Path retrieval

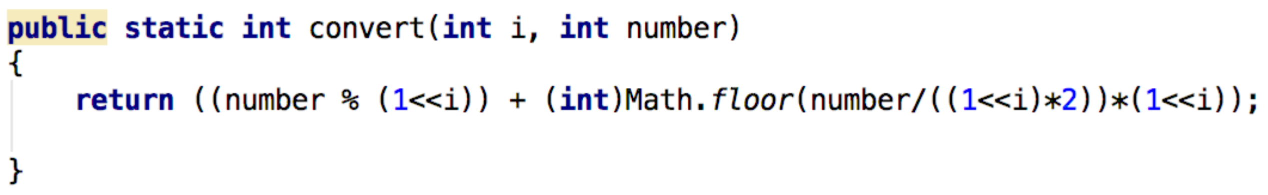
- 92:94, Firstly we add the start node to the path, we then get the maximal set containing all nodes.
- 97, We would like to find the nodes that make up the rest of the path, as we already know one of the states the amount of other nodes we need to find would be N-1 this is the reason we set i to be 1.
- 99:101, We set the index to -1 as this variable needs to be stored between runs, we then iterate through all nodes in our graph skipping over nodes which are either not in the current state or are the start node.
- 102, If we have not updated the index then we set it to the first node found that is in the state and not the start node.
- 103, We calculate the distance from the set including the previous found node in our tour to the optimal found end node in our set.
- 104, We calculate the distance from the set including the previous found node in our tour to the current found end node in our set.
- 105:106, If we found a better node that takes us from the previous node this state then update the current best index.
- 110:112, Add the best found end node for the current state to our tour and remove it from the current state. Store it as the last previous end node.
- 115, Add the start node to the path as it should create a cycle due to it solving the travelling salesman.

Improvements

When studying the algorithm it is possible to improve it in relation to storage. If we consider a graph with only 4 possible nodes,"0001" = 1, "0010" = 2, "0100" = 4 and "1000" = 8, we can see instances where memo[end][state] will be un-used. For example in paths ending in 8 we will never use the bit values 1-7, as the bit value will always contain the fourth element who's value is 8, additionally we waste space in between bit values for example any set ending in state 1 will never take odd values. Below you can see a diagram showing this wasted space, using a mapping function it may be possible to reduce the amount of space needed to store these values. The amount of space wasted between bit values seemed to be related to the state they were ending in having spacing of $1 << endstate$, using this fact I was able to make the following mapping function reducing the size needed for the overall array to $n * 2^{n-2}$ rather than $2 * 2^n$ which according to on-line sources is the theoretical minimum.



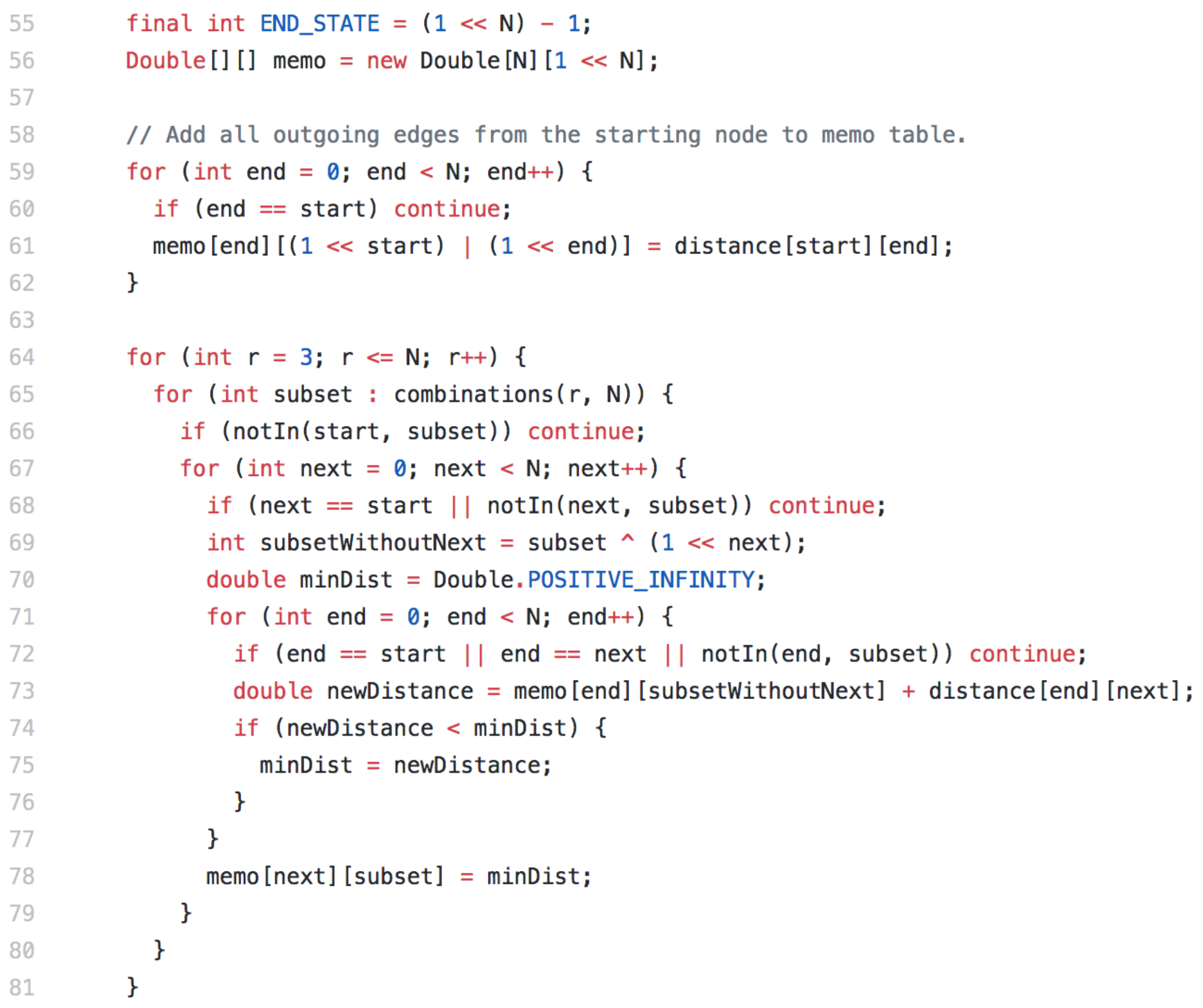
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15.
1	x	x	1	x	1	x	1	x	1	x	1	x	1	x	1
2	x	x	1	x	x	1	1	x	x	1	1	x	x	1	1
3	x	x	x	x	1	1	1	x	x	x	x	1	1	1	1
4	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1



```java
public static int convert(int i, int number)
{
    return ((number % (1<<i)) + (int)Math.floor(number/((1<<i)*2))*(1<<i));

}
```

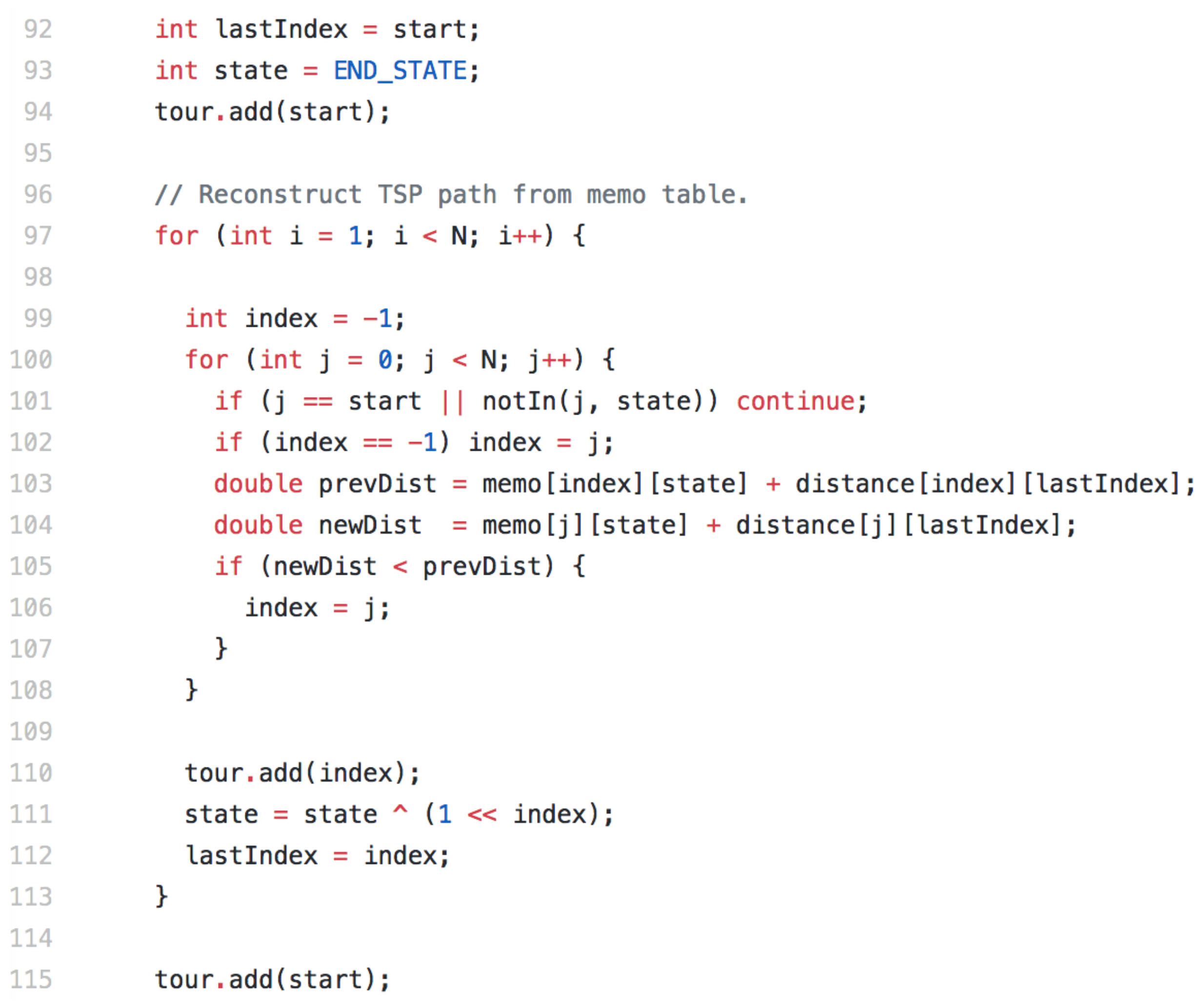
0.5 Code

0.5.1 William Fiset generating cost values



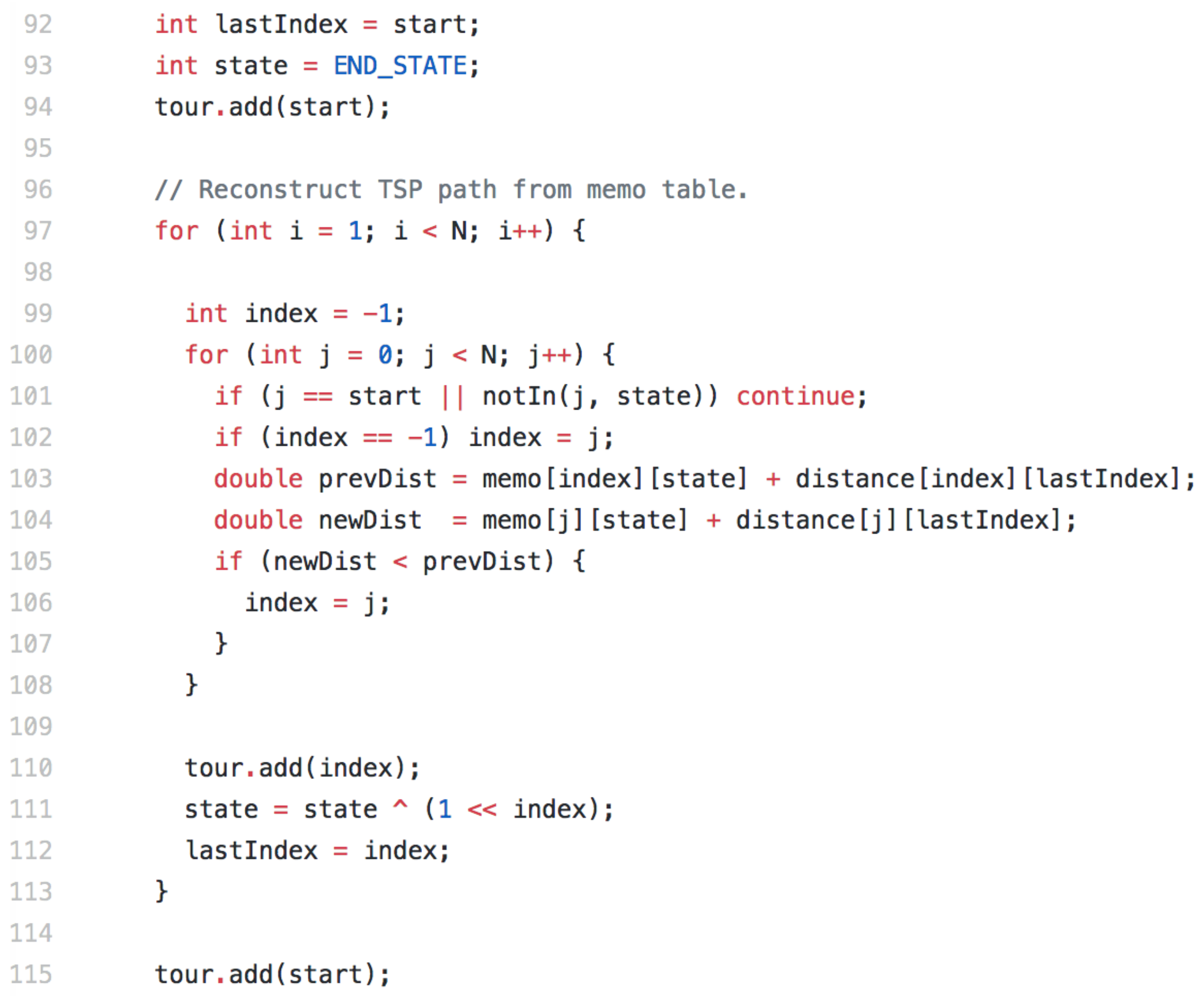
```
    final int END_STATE = (1 << N) - 1;
    Double[][] memo = new Double[N][1 << N];

    // Add all outgoing edges from the starting node to memo table.
    for (int end = 0; end < N; end++) {
      if (end == start) continue;
      memo[end][(1 << start) | (1 << end)] = distance[start][end];
    }

    for (int r = 3; r <= N; r++) {
      for (int subset : combinations(r, N)) {
        if (notIn(start, subset)) continue;
        for (int next = 0; next < N; next++) {
          if (next == start || notIn(next, subset)) continue;
          int subsetWithoutNext = subset ^ (1 << next);
          double minDist = Double.POSITIVE_INFINITY;
          for (int end = 0; end < N; end++) {
            if (end == start || end == next || notIn(end, subset)) continue;
            double newDistance = memo[end][subsetWithoutNext] + distance[end][next];
            if (newDistance < minDist) {
              minDist = newDistance;
            }
          }
          memo[next][subset] = minDist;
        }
      }
    }
```

0.5.2 William Fiset returning cost of path



```java
    int lastIndex = start;
    int state = END_STATE;
    tour.add(start);

    // Reconstruct TSP path from memo table.
    for (int i = 1; i < N; i++) {

      int index = -1;
      for (int j = 0; j < N; j++) {
        if (j == start || notIn(j, state)) continue;
        if (index == -1) index = j;
        double prevDist = memo[index][state] + distance[index][lastIndex];
        double newDist  = memo[j][state] + distance[j][lastIndex];
        if (newDist < prevDist) {
          index = j;
        }
      }

      tour.add(index);
      state = state ^ (1 << index);
      lastIndex = index;
    }

    tour.add(start);
```

0.5.3 William Fiset returning path



```java
    int lastIndex = start;
    int state = END_STATE;
    tour.add(start);

    // Reconstruct TSP path from memo table.
    for (int i = 1; i < N; i++) {

      int index = -1;
      for (int j = 0; j < N; j++) {
        if (j == start || notIn(j, state)) continue;
        if (index == -1) index = j;
        double prevDist = memo[index][state] + distance[index][lastIndex];
        double newDist  = memo[j][state] + distance[j][lastIndex];
        if (newDist < prevDist) {
          index = j;
        }
      }

      tour.add(index);
      state = state ^ (1 << index);
      lastIndex = index;
    }

    tour.add(start);
```