VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DISCRETE MATHEMATICS FOR COMPUTER SCIENCE (CO1007)

Assignment (Semester: 232, Class: L04)

"Traveling Salesman Problem"

Advisor: Trần Hồng Tài
Student: Trương Quang Nghĩa - 2212243

Ho Chi Minh City, 06/2024

Contents

1 Introduction

1.1 What is the Traveling Salesman Problem?

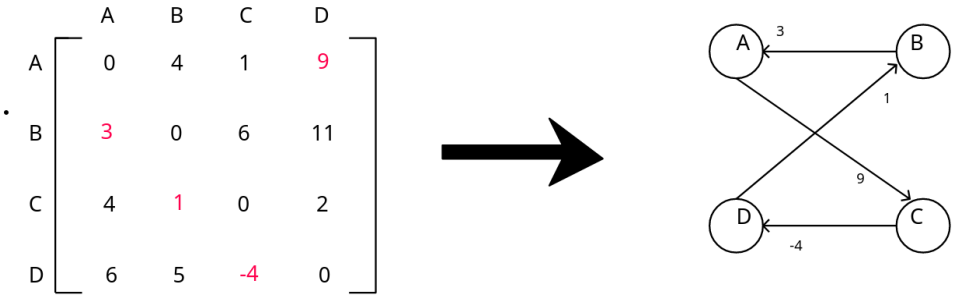
"The travelling salesman problem, also known as the travelling salesperson problem (TSP), asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research" - from Wikipedia.

1.2 Finding approach

In some other words, the problem is: given a **complete graph** with weighted edges (as an adjacency matrix) what is the the **Hamiltonian** cycle (path that visits every node once) of minimum cost?

In our case of this assignment, if an edge between two nodes does not exist, simply set its value to 0. And since all the edges are positive (provided in the note), we can disregard checking for negative cycles.

Here is an example of what we are looking for:



Full tour: A -> D -> C -> B -> A
Tour cost: 9 + -4 + 1 + 4 = 9

Note that: there may be multiple valid optimal tours, but they will all have the minimum cost.

Finding the optimal solution to the TSP problem is very challenging; it is known to be NP-Complete. However, numerous approximation algorithms exist.

Two common algorithms are the **Brute force solution** and **Dynamic programming**. We will discuss both but will only use the **Dynamic programming** approach to solve the problem.

1.3 Brute force solution

The brute force way to solve the TSP is to compute the cost of every possible tour. This means we have to try all possible permutations of node orderings which takes O(n!) time. Which is very slow.

I have listed all permutations of nodes and highlighted the ones which yield the optimal solution.

	A	B	C	D
A	0	4	1	9
B	3	0	6	11
C	4	1	0	2
D	6	5	-4	0

Tour	Cost	Tour	Cost
A B C D	18	C A B D	15
A B D C	15	C A D B	24
A C B D	19	C B A D	9
A C D B	11	C B D A	19
A D B C	24	C D A B	18
A D C B	9	C D B A	11
B A C D	11	D A B C	18
B A D C	9	D A C B	19
B C A D	24	D B A C	11
B C A D	18	D B C A	24
B D A C	19	D C A B	15
B D C A	15	D C B A	9

1.4 Dynamic programming solution

The dynamic programming solution to the TSP problem significantly improves on the time complexity, taking it from $O(n!)$ to $O(n^2 2^n)$.

At first glance, this may not seem like a substantial improvement, however, it now makes solving this problem feasible on graphs with up to roughly 23 nodes on a typical computer.

In our case, we only have to solve a graph with a maximum of 20 nodes, so this algorithm is extremely suitable.

1.5 Compare $O(n!)$ to $O(n^2 2^n)$

At first, you can notice that $n!$ is optimal for small number, but this quickly changes favor to $n^2 2^n$ which can give a significant improvement over $n!$ seeing how large the number can get when we hit n equals 15.

n	$n!$	$n^2 2^n$
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800
6	720	2304
...	...	...
15	1307674368000	7372800
16	20922789888000	16777216
17	355687428096000	37879808

So in this assignment, because of its performance, we will use dynamic programming to solve the given problem. Now let us see how it works.

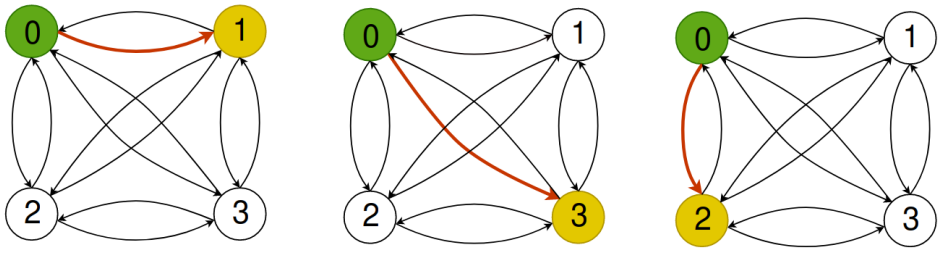
2 Solving TSP with Dynamic programming solution

2.1 The main concept

The main idea will be to compute the optimal solution for all subpaths of length N while using information from the already know optimal partial tours of length $N - 1$.

Firstly, we choose a starting node, make sure to select one $0 \leq S < N$ to be the designated starting node for the tour.

For this example let S = 'node 0'.



Next, compute and store the optimal value from S to each node X ($\neq S$). This will solve problem for all path of length $n = 2$.
To compute the optimal solution for paths of length 3, we need to remember (store) two things from each of the n = 2 cases:

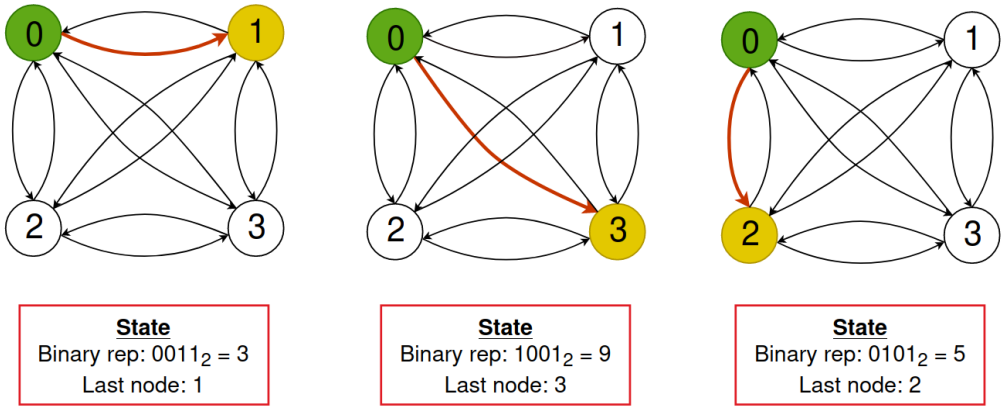
- The **set of visited nodes** in the subpaths.
- The **index of the last visited node** in the path.

Together these two things form our dynamic programming state. There are N possible nodes that we could have visited last and 2^N possible subsets of visited nodes. Therefore the space needed to store the answer to each subproblem is bounded by $O(N2^N)$.

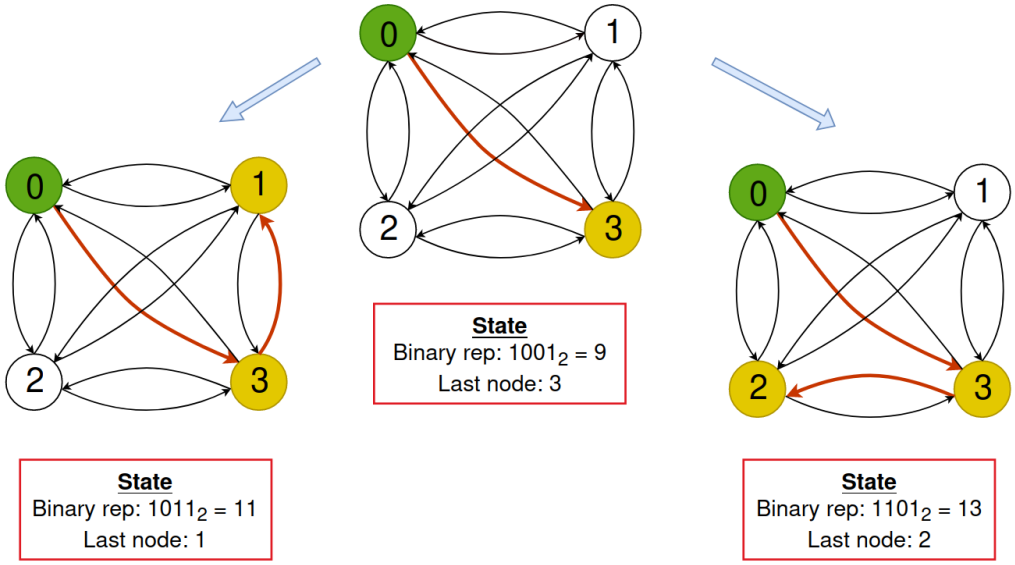
2.2 Storing

An issue we are going to face when trying to store the dynamic programming state is representing the set of visited nodes. The best way to do this is to use **a single 32-bit integer**. The main idea is that, e.g. If the third node has been visited we flip on the 3rd bit to 1 in the binary representation of the integer. The advantage of this is that a 32-bit int is compact, quick and allows for easy caching in a memo table.

For example: On the left most graph, we have visited the Oth and the first node so the binary representation is 0011 if the least significant bit is on the right. Similarly apply for the rest of the graph.



To solve $3 \leq n \leq N$, we are going to take the solved subpaths form $n-1$ and add another edge extending to a node which has not already been visited from the last visited node (which has been saved).



This process continues with gradually longer and longer paths until all paths are of length n.

2.3 Backtracking

To complete the TSP tour, we need to connect our tour back to the start node S. Loop over the *end state* in the memo table for every possible end position and minimize the lookup value plus the cost of going back to S.

Note that: the **end state** is one where the binary representation is composed fo N 1's, meaning each node has been visited.

3 Pseudo code & explanation

3.1 The Traveling function

This is the function will return the shortest way to go over all the vertices in the graph and go back to the starting vertex. It takes 3 inputs, the given matrix graph, the starting node, and the number of nodes.

```
# G - 2D adjacency matrix representing graph
# n - The number of nodes
# S - The start node (0 ≤ S ≤ N)
string Traveling(G, n, S):
    # Initialize memo table filled with null values or +∞
    memo = 2D table of size n by 2^n

    setup(G, memo, S, n)
    solve(G, memo, S, n)
    minCost = findMinCost(G, memo, S, n)
    tour = findOptimalTour(G, memo, S, n)

    return tour
```

First we initialize the memo table with size n. Then we call 4 functions, **setup(), solve(), findMinCost(), and findOptimalTour()** to do the calculation for us.

Finally, the function returns the optimal path that we are looking for. From now on, you can just think of it as a black box which gives you the answer for every input you give to it.

I will reveal the inside details later in this report.

3.2 Setup method

Initializes the memo table by caching the optimal solution from the starting node to every other node.

```
void setup(G, memo, S, n):
    for(i = 0; i < N; i++):
        if (i == S): continue

        # Store the optimal value from node S to each node i
        # (this is given as input int the adjacency matrix G).

        memo[i][i << S | 1 << i] = m[S][i]
```

We basically loop through each node, skipping over the start node and the cache the optimal value from S to i which can be found in the input distance matrix.

The DP state we store is the end node as i and the mask with bits S and i set to 1 hence the double bits shift.

3.3 Solve method

This function is the actual one that solves our problem.

```
1  void solve(G, memo, S, n):
2      for(r = 3; r <= n; r++):
3          # The combinations function generates all bit sets
4          # of size n with r bits set to 1. For example,
5          # combinations (3, 4) = {0111_2, 1011_2, 1101_2, 1110_2}
6          for subset in combinations(r, n):
7              if (notIn(S, subset)): continue
8              for(next = 0; next < n; next++):
9                  if (next == S || notIn(next, subset)): continue
10                 # The subset state without the next node
11                 state = subset ^ (1 << next)
12                 minDist = +∞
13                 # 'e' is short for end node
14                 for(e = 0; e < n; e++):
15                     if (e == S || e == next || notIn(e, subset))):
16                         continue
17                     newDistance = memo[e][state] + G[e][next]
18                     if (newDistance < minDist):
19                         minDist = newDistance
20                 memo[next][subset] = minDist
21
22 # Returns true if the i^th bit in 'subset' is not set
23 bool notIn(i, subset):
24     return ((1 << i) & subset) == 0
```

The first line in the method loops over $r = 3$ up to an inclusive, think of r as the number of nodes in a partial tour so we're increasing this number one at a time.

The next line says for subset in combinations the **combinations()** function generates all bit sets of size and width exactly our bits set to 1. For example, as seen in the comments at line 5 in the snippet, when calling the combinations functions with r equals 3 and n equals 4 we get four different bit sets, each distinct and with three ones turned on. These are meant to represent a subset of visit nodes.

Moving on, notice that I enforce the node s to be part of the generated subset, otherwise the subset of nodes is not valid since it could not have started at our designated starting node.

Notice that this *if statement* calls the **notIn()** function defined at the bottom of this snippet. All it does is it checks if the ith bit in the subset is a zero, then we loop over a variable called *next* which represents the index of the next node. The next node must be part of the current subset. This may sound strange but know that the subset variable generated by the **combinations()** function has a bit which is meant for the next node, this is why variable state on the next line represents the subset excluding the next node. This is so we can loop up in our memo table to figure out what the best partial tour value is when the next node was not yet in a subset. Being able to look back parts of other partially completed tours is essential to the dynamic programming aspect of this algorithm.

The following variable to consider is e, short for end node. This variable is quite important because while the next node is temporarily fixed in the scope of the inner loop, we try all possible end nodes of the current subset and try to see which end node best optimizes this partial tour. Of course the end node cannot be any of the start node, the next node or not be part of the current subset that we are considering, so we skip all of that possibilities. So we compute the new distance and compare it to the minimum distance, if the new one is better then the minimum one then we update the best minimum distance. Afterwards, once we've considered all possible

end nodes to connect to the next node, we store the best partial tour in the memo table and this conclude the **solve()** method.

The only left unanswered question is how the **combinations()** function works. I'll explain it gets done right after this.

3.4 Combinations method

```
# Generate all bit sets of size n with r bits set to 1.
void combinations(r, n):
    subsets = []
    combinations(0, 0, r, n, subsets)
    return subsets

# Recursive method to generate bit sets.
void combinations(set, at, r, n, subsets):
    if (r == 0):
        subsets.add(set)
    else:
        for(i = at; i < n; i++):
            # Flip on i^th bit
            set = set | (1 << i)

            combinations(set, i + 1, r - 1, n, subsets)

            #Backtrack and flip off i^th bit
            set = set & ~(1 << i)
```

What the first **combinations()** method does is it fills up the subsets array using the second **combinations()** method and then returns that result.

Starting with the empty set which is zero, you want to set r out of n bits to be 1 for all possible combination, so you keep track of which index position you're currently at and then try to set the bit position to 1 and then keep moving forward hoping that at the end you have exactly r bits, but if you didn't, you backtrack flip off the bit you flipped on and the move to the next position.

3.5 findMinCost method

Right now in our memo table we have the optimal value for each partial tour with n nodes, so let's see how can we reuse the information to find the minimum tour of value.

```
int findMinCost(G, memo, S, n):
    # The end state is the bit mask with N bits set to 1
    # (equivalently 2^n - 1)
    END_STATE = (1 << n) -1

    minTourCost = +∞

    for (e = 0; e < n; e++):
        if (e == S): continue

        tourCost = memo[e][END_STATE] + G[e][S]
        if (tourCost < minTourCost):
```

```
            minTourCost = tourCost
    return minTourCost
```

The trick is going to be to construct a bit mask for the end state and use that to do a lookup in our memo. The end state is the bit mask with n bits set to 1 which we can obtain by doing a bit shift and then subtracting 1. Then look at each end node candidate and minimize over the torque cost by looking at what's in our memo table and the distance from the end node back to the start node S.

3.6 findOptimalTour method

```
vector<int> findOptimalTour(G, memo, S, n):
    lastIndex = S
    state = (1 << n) - 1 # End state
    tour = array of size n+1

    for (i = n-1; i >= 1; i--):
        index = -1
        for (j = 0; j < n; j++):
            if (j == S || notIn(j , state): continue
            if(index == -1) index = j
            prevDist = memo[index][state] + G[index][lastIndex]
            newDist  = memo[j][state] + G[j][lastIndex]
            if (newDist < prevDist): index = j

        tour[i] = index
        state = state ^ (1 << index)
        lastIndex = index

    tour[0] = tour[n] = S
    return tour
```

What we're going to do to find the actual tour is work backwards from the end state and you lookup in our memo table to find the next optimal node.

We will keep track of the last index we were at and the current state which begins with all visited nodes, then we loop over i from $n-1$ to 1 which tracks the index position for the tour. To actually find the next optimal node going backwards we're going to use a variable called index which will track the best node. The inner loop loops over j which represents all possible candidates for the next node, we must ensure that j is not the starting node that it is part of the state meaning it has not yet been visited. If this is the first valid iteration, the variable index will be set to -1, so set it to j, otherwise compare the optimal values of the best distances between nodes index and j and update index if node j is better. Once the optimal index if found, store that as part of the tour and flip off the bit in the state which represents the index node. Finally, set the first and last nodes of the tour to be S (the starting node) because the tour needs to start and end on that node. Then simply return the tour.

4 Conclusion

In this assignment, we explored the application of dynamic programming (DP) to solve the Traveling Salesman Problem (TSP), a classic optimization challenge in computer science and operations research. By leveraging the principles of dynamic programming, we were able to

significantly reduce the computational complexity compared to brute force methods.

Dynamic programming's power lies in its ability to break down the TSP into manageable subproblems, storing intermediate results to avoid redundant calculations. This approach not only improves efficiency but also demonstrates the practical utility of DP in solving complex combinatorial problems.

The results obtained from our implementation confirm the theoretical expectations, showing that dynamic programming is a viable and effective strategy for solving the TSP for small to moderately sized datasets. Despite its limitations in handling very large datasets due to exponential growth in time complexity, dynamic programming provides a foundational method that can be further optimized and combined with other techniques such as heuristics and approximation algorithms.

In summary, this assignment underscored the importance of algorithmic design in tackling NP-hard problems and highlighted dynamic programming as a robust tool in the arsenal of problem-solving strategies in computer science.

References

[1] Bellman–Ford Algorithm
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

[2] Travelling Salesman Problem using Dynamic Programming
https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/

[3] Travelling Salesman Problem implementation using BackTracking
https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/

[4] Wikepedia: Bellman–Ford algorithm
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

[5] Wikepedia: Page table
https://en.wikipedia.org/wiki/Page_table#:~:text=A%20page%20table%20is%20the,virtual%20addresses%20and%20physical%20addresses

[6] Youtube: Bellman-Ford in 5 minutes — Step by step
https://www.youtube.com/watch?v=obWXjtg0L64

[7] Youtube: Bellman Ford Algorithm | Shortest path & Negative cycles | Graph Theory
https://www.youtube.com/watch?v=lyw4FaxrwHg