# Abstract

In this phase, we solve the TSP given in the assignment manual with the DFS algorithm. The algorithm uses the distance table reference and the problem branches pruning to reduce the search time. The proposed algorithm is implemented in C.

# Algorithm

At first, we used the brute-force algorithm. The permutation was applied to the array of the cities, the path was determined for all cases, and the minimum distance was determined. This was quite inefficient. If the minimum distance has already been exceeded by current path, the arrangement of the city is not required to be inspected. The search was performed only when the conditions were satisfied. This is called backtracking algorithm. By applying this algorithm, the time could be further reduced.

There was also one drawback in this algorithm. Even though backtracking was done, it was necessary to calculate the corresponding length each time. The length calculation was composed of a fairly complex expression such as root and square. So at the end of the meeting, we came up with one measure. If we created a matrix that stores distances between cities before we start the search, and then calculate the path using only the constant values already stored, we do not have to repeat the complex operations. As a result of applying this algorithm, we could find the solution within the most optimized time.

The final algorithm we used is as follows. First, use the double for statement to store the distance between cities. In this case, it is stored in the form of triangular matrix to prevent unnecessary calculation. Then start DFS(Depth First Search). If the number of visited cities reaches seven and the distance is less than the current minimum value, the distance is stored as the minimum value and the order of the searched city is also stored. When searching, check the backtracking condition before going to the next city, and search the next city only when the condition is satisfied. When all the searches are completed, the final path and length are printed and the program is terminated.

# Flow Chart

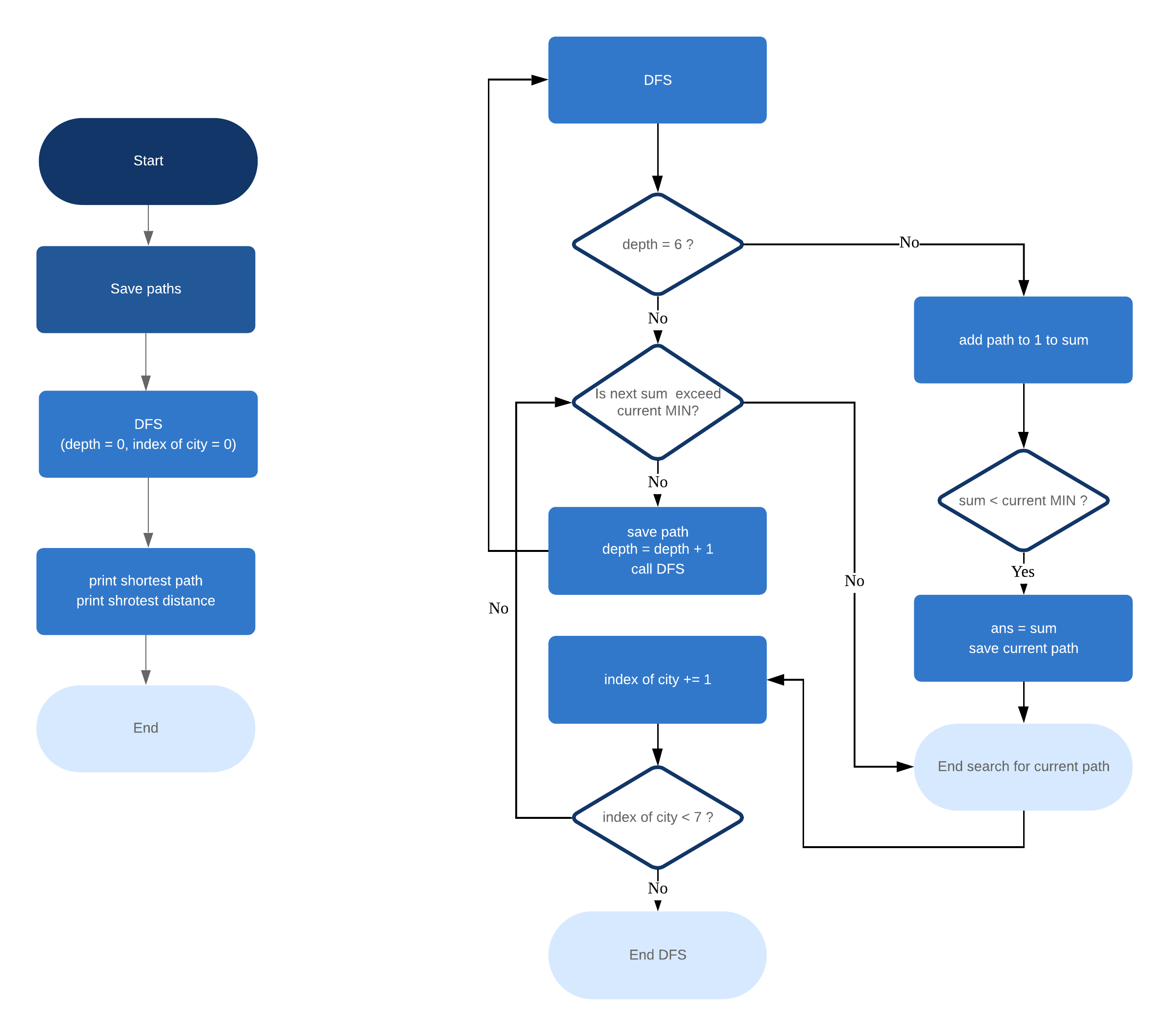


Figure 1 Flow Chart

# Source Code

[tsp.c]

/\*

\* SWE3005 Computer Architectures

\* Prof. Hee Yong Youn

\* 2019 Spring Course Project

\* TSP on MIPS

\* Phase 1: High level language program

\* Team. 20: Hyungjun Kim, Ju-eun Park

\*/

#include <stdio.h>

#include <math.h>

#include <time.h>

typedef struct {

int num;

int x, y;

} node;

node cities[7] = {{1, 0, 0}, {2, 8, 6}, {3, 2, 4},

{4, 6, 7}, {5, 1, 3}, {6, 9, 4}, {7, 2, 3}};

int visit[7];

double ans = 100000000;

double arr[7][7];

int current\_path[7];

int shortest\_path[7];

void save\_path() {

for(int i = 0; i < 7; i++){

shortest\_path[i] = current\_path[i];

}

}

double distance(node\* a, node\* b) {

return sqrt(pow((a->x - b->x), 2) + pow((a->y - b->y), 2));

}

void print\_path(int\* arr) {

for(int i = 0; i < 7; i++){

printf("%d ", arr[i]);

}

printf("\n");

}

void dfs(int n, int depth, double sum) {

if(depth == 6){

// add path to 1

sum += arr[n][0];

if(sum < ans){

ans = sum;

save\_path();

}

return;

}

for(int i = 1; i < 7; i++) {

if(visit[i] == 1) continue;

if(sum + arr[n][i] < ans) {

visit[i] = 1;

current\_path[depth+1] = cities[i].num;

dfs(i, depth + 1, sum + arr[n][i]);

visit[i] = 0;

}

}

}

int main() {

clock\_t begin, end;

begin = clock();

for(int i = 0; i < 7; i++) {

for(int j = 0; j < i; j++) {

arr[i][j] = distance(&cities[i], &cities[j]);

arr[j][i] = arr[i][j];

}

}

shortest\_path[0] = 1;

current\_path[0] = 1;

dfs(0, 0, 0);

printf("SHORTEST PATH : ");

print\_path(shortest\_path);

printf("DISTANCE : %f\n", ans);

end = clock();

printf("\nTIME : %lf", (double)(end - begin) / CLOCKS\_PER\_SEC);

}

# Output

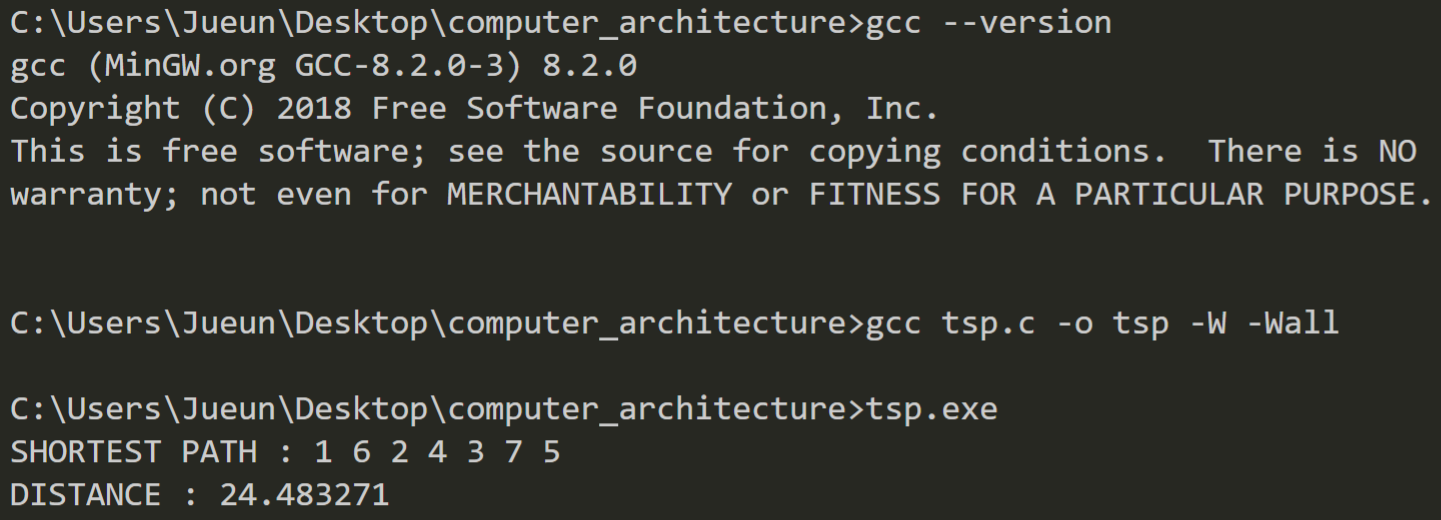


Figure Compiler Version and Program Output

# Conclusion

The algorithm that finds the shortest path using table referencing and depth-first search is proposed in this phase. The algorithm searches each most of branch of state space tree before going down each leaf state node. However, the algorithm prunes some unnecessary branches don’t have to be searched because the next sum of the distance of the branch already exceeds the current known minimum distance. To find the sum of distance, the algorithm refers to the distance table already made before the depth-first search execute. Using these methods, the implemented code can find the shortest path efficiently.