```c
/* ===================================================================
 *
 * CS 566 - Assignment 04
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code solves the Travelling Salesman Problem by using a
 * Depth First Search, Branch and Bound algorithm.
 *
 * =================================================================== */

#include <stdio.h>
#include <limits.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"
#include "tsplib95.h"
#include "tsp.h"

struct tsp_state *tsp_state_alloc(struct tsp_matrix *matrix)
{
        struct tsp_state *state;
        state =
            (struct tsp_state *)malloc(offsetof(struct tsp_state, tour) +
                                       sizeof(state->tour[0]) * matrix->n);
        state->cost = 0;
        state->ub = INT_MAX;
        state->ub_rank = 0;
        state->give_depth = state->subtree_depth = 1;
        state->len = 1;
        state->tour[0] = 0;     /* we always begin the path from node 0 */
        /* state->first_given = state->last_given = NULL; */
        state->matrix = matrix;
        state->last_started = calloc(matrix->n, sizeof(*state->last_started));
        state->term_token.request = MPI_REQUEST_NULL;
        state->term_token.token = NO_TOKEN;
        state->term_token.mycolor = WHITE;
        state->work_partner = 0;
        state->best_tour = calloc(matrix->n, sizeof(*state->best_tour));
        return state;
}

int main(int argc, char *argv[])
{
        struct tsp_matrix matrix;
        struct tsp_state *state;
        int numprocs, namelen, i;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        FILE *file;

        /* setup */
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Get_processor_name(processor_name, &namelen);

        /* parse our TSP matrix */
        file = fopen(argv[1], "r");
        parse_matrix_from_file(&matrix, file);

        /* setup the state variable */
        state = tsp_state_alloc(&matrix);
        MPI_Comm_rank(MPI_COMM_WORLD, &(state->myrank));
        MPI_Comm_size(MPI_COMM_WORLD, &(state->num_procs));

        /* printf("%d: After tsp_state_alloc\n", state->myrank); */
        if (state->myrank == 0) {
                state->work_state = WORKING;
//              print_matrix(&matrix);
        } else {
                state->work_state = NEED_WORK;
        }

        tsp(state);

        if (state->myrank == 0) {
                fprintf(stdout, "best: %d\n", state->ub);
                for (i = 0; i < state->matrix->n; i++)
                        fprintf(stdout, "%2d ", state->best_tour[i]);
                fprintf(stdout, "\n");
                fprintf(stdout, "total time: %f\n", state->total_time);
                fprintf(stdout, "work time: %f (%2.1f%%)\n", state->work_time,
                        state->work_time / state->total_time * 100.0);
        }

        MPI_Finalize();
        return 0;

}

void tsp(struct tsp_state *state)
{
        MPI_Status pending_status;
        MPI_Request request;
        int i;

        double start_time = MPI_Wtime();
        state->work_time = 0;

        /* begin by giving away work */
        if (state->myrank == 0) {
                service_pending_messages(state);
        }

        while (state->work_state != QUIT) {

                if (state->work_state == NEED_WORK) {
                        request_work(state);

                        if (state->work_state != WORKING) {
                                state->work_state = IDLE;
                                if (state->myrank == 0
                                    || state->term_token.token != NO_TOKEN)
                                        send_token(state);
                        }
                }

                if (state->work_state == WORKING) {
                        double work_start_time = MPI_Wtime();
                        do_work(state);
                        state->work_time += MPI_Wtime() - work_start_time;
                }

                if (state->work_state == IDLE) {
//                      fprintf(stdout, "%d: Probing for pending messages.\n", state->myrank);
                        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                                  &pending_status);
//                      fprintf(stdout, "%d: We have messages.\n", );
                }

                service_pending_messages(state);

        }
        if (state->myrank == 0) {
                if (state->ub_rank != state->myrank) {
                        MPI_Isend(&state->myrank, 1, MPI_INT, state->ub_rank,
                                  BEST_PATH_REQ_TAG, MPI_COMM_WORLD, &request);
                        MPI_Recv(state->best_tour, state->matrix->n, MPI_INT, state->ub_rank,
BEST_PATH_TAG, MPI_COMM_WORLD, &pending_status);        // send the termination message
                }
                for (i = 1; i < state->num_procs; i++) {
                        MPI_Isend(&i, 1, MPI_INT, i, TERMINATION_TAG,
                                  MPI_COMM_WORLD, &request);
                }

        }

        state->total_time = MPI_Wtime() - start_time;
}

void send_token(struct tsp_state *state)
{
        MPI_Request request;
        if (state->term_token.token == NO_TOKEN) {
                state->term_token.token = WHITE;
        }
        MPI_Isend(&state->term_token.token, 1, MPI_INT,
                  ((state->myrank + 1) % state->num_procs), TOKEN_TAG,
                  MPI_COMM_WORLD, &request);
        state->term_token.mycolor = WHITE;
}

int next_available_node(struct tsp_state *state, int depth)
{
        int next_node = state->last_started[depth] + 1, last_next_node, i;
        do {
                last_next_node = next_node;
                for (i = 0; i < depth; i++)
                        if (next_node == state->tour[i])
                                next_node++;
        } while (last_next_node != next_node);
        return next_node;
}

/*
        do a fixed amount of work
*/
void do_work(struct tsp_state *state)
{
        int work_counter = 0, i, next_node;
        int progress_counter = 0;

        for (work_counter = 0; work_counter < WORK_SLICE; work_counter++) {
                int go_up = 1;

#if 0
                if (progress_counter == 5000000) {
                        fprintf(stdout, "%d: ", state->myrank);
                        for (i = 0; i < state->len; i++)
                                fprintf(stdout, "%2d ", state->tour[i]);
                        fprintf(stdout, "\n");
                        progress_counter = 0;
                } else
                        progress_counter++;
#endif
                /* try to go down */
                if (state->len < (state->matrix->n)) {
                        /* pick the next available node */
                        next_node = next_available_node(state, state->len);
                        //fprintf(stdout, "next_node: %d\n", next_node);

                        if (next_node < state->matrix->n) {

                                /* go down */
                                state->last_started[state->len] = next_node;

                                /* append node to tour */
                                state->tour[state->len] = next_node;
                                state->cost +=
                                    CELL(state->matrix,
                                         state->tour[state->len - 1],
                                         state->tour[state->len]);
                                state->len++;

                                //fprintf(stdout, "down %d, len %d\n", next_node, state->len);

                                /* reset next level */
                                if (state->len < state->matrix->n)
                                        state->last_started[state->len] = 0;

                                if (state->cost >= state->ub) {
                                        /* prune bad branches */
/*                                      fprintf(stdout, "pruning: ");
                                        for (i = 0; i < state->len; i++) fprintf(stdout, "%2d ", state-
>tour[i]);
                                        fprintf(stdout, "\n");*/
                                        go_up = 1;
                                } else {
                                        go_up = 0;

                                        /* if we have a complete tour, update the ub */
                                        if (state->len == state->matrix->n) {
                                                //for (i = 0; i < state->len; i++) printf("%2d ", state-
>tour[i]);
                                                int cycle_cost =
                                                    state->cost +
                                                    CELL(state->matrix,
                                                         state->tour[state->
                                                                     len - 1],
                                                         state->tour[0]);
                                                //printf("cost: %d", cycle_cost);
                                                if (cycle_cost < state->ub) {
                                                        state->ub = cycle_cost;
                                                        state->ub_rank =
                                                            state->myrank;
                                                        send_ub_message(state);
                                                        //printf(" new best");
                                                        memcpy(state->best_tour,
                                                               state->tour,
                                                               sizeof(*state->
                                                                      tour) *
                                                               state->len);
                                                }
                                                //printf("\n");
                                        }
                                }
                        }
                }

                if (go_up) {
//                      printf("up\n");
                        /* go up */
                        state->len--;
                        if (state->len < state->subtree_depth) {
                                state->work_state = NEED_WORK;
                                break;
                        }
                        state->cost -=
                            CELL(state->matrix, state->tour[state->len - 1],
                                 state->tour[state->len]);
                }
        }

}

/*
 * service all the pending messages in our message queue
 */
void service_pending_messages(struct tsp_state *state)
{

        MPI_Status pending_status, status;
        MPI_Request request;
        int msg_pending;
        /* is there a message pending in the message QUEUE ? */
        int temp, possible_ub;

        /* service UB broadcasts */
        MPI_Iprobe(MPI_ANY_SOURCE, UB_TAG, MPI_COMM_WORLD, &msg_pending,
                   &pending_status);
        while (msg_pending) {
                MPI_Recv(&possible_ub, 1, MPI_INT, pending_status.MPI_SOURCE,
                         pending_status.MPI_TAG, MPI_COMM_WORLD, &status);
                if (possible_ub < state->ub) {
                        state->ub = possible_ub;
                        state->ub_rank = pending_status.MPI_SOURCE;
                }
                MPI_Iprobe(MPI_ANY_SOURCE, UB_TAG, MPI_COMM_WORLD, &msg_pending,
                           &pending_status);
        }

        /* service BEST PATH REQ message */
        MPI_Iprobe(MPI_ANY_SOURCE, BEST_PATH_REQ_TAG, MPI_COMM_WORLD,
                   &msg_pending, &pending_status);
        while (msg_pending) {
                MPI_Recv(&temp, 1, MPI_INT, pending_status.MPI_SOURCE,
                         pending_status.MPI_TAG, MPI_COMM_WORLD, &status);

                MPI_Isend(state->best_tour, state->matrix->n, MPI_INT,
                          pending_status.MPI_SOURCE, BEST_PATH_TAG,
                          MPI_COMM_WORLD, &request);

                MPI_Iprobe(MPI_ANY_SOURCE, BEST_PATH_REQ_TAG, MPI_COMM_WORLD,
                           &msg_pending, &pending_status);
        }

        /* service work requests */
        MPI_Iprobe(MPI_ANY_SOURCE, WORK_REQ_TAG, MPI_COMM_WORLD, &msg_pending,
                   &pending_status);
        while (msg_pending) {
                MPI_Recv(&temp, 1, MPI_INT, pending_status.MPI_SOURCE,
                         pending_status.MPI_TAG, MPI_COMM_WORLD, &status);
                service_work_request(state, pending_status);
                MPI_Iprobe(MPI_ANY_SOURCE, WORK_REQ_TAG, MPI_COMM_WORLD,
                           &msg_pending, &pending_status);
        }

        /* service token sends */
        MPI_Iprobe(MPI_ANY_SOURCE, TOKEN_TAG, MPI_COMM_WORLD, &msg_pending,
                   &pending_status);
        while (msg_pending) {
                /* fprintf(stdout, "%d: Receiving token from %d.\n", state->myrank,
pending_status.MPI_SOURCE); */
```

```
                MPI_Recv(&(state->term_token.token), 1, MPI_INT,
                         pending_status.MPI_SOURCE, pending_status.MPI_TAG,
                         MPI_COMM_WORLD, &status);
                /* fprintf(stdout, "%d: Token color: %d.\n", state->myrank, state->term_token.token); */
                if (state->term_token.mycolor == BLACK)
                        state->term_token.token = BLACK;
                if (state->myrank == 0) {
                        if (state->term_token.token == WHITE)
                                state->work_state = QUIT;
                        else
                                state->term_token.token = WHITE;
                }
                if (state->work_state == IDLE) {
                        send_token(state);
                }
                MPI_Iprobe(MPI_ANY_SOURCE, TOKEN_TAG, MPI_COMM_WORLD,
                           &msg_pending, &pending_status);
        }

        /* service TERMINATION broadcast */
        MPI_Iprobe(MPI_ANY_SOURCE, TERMINATION_TAG, MPI_COMM_WORLD,
                   &msg_pending, &pending_status);
        while (msg_pending) {
                state->work_state = QUIT;
                return;
        }

}

void service_work_request(struct tsp_state *state, MPI_Status status)
{
        int work_deny = 1;
        int i;
        int *outbuf;
        int next_node;

        if (state->work_state == WORKING) {
                while (state->give_depth < (state->matrix->n - MIN_WORK_LEVELS)
                       && (next_node =
                           next_available_node(state,
                                               state->give_depth)) >=
                       state->matrix->n) {
                        state->give_depth++;
                }

                /* do we have work to give away ? */
                if (state->give_depth < (state->matrix->n - MIN_WORK_LEVELS)
                    && next_node < state->matrix->n) {
                        /* ok, give out the work */
                        work_deny = 0;
                        state->last_started[state->give_depth] = next_node;
                        /* give away the next node at this depth */

                        /* prepare the message buffer(we 're sending the current path to the given depth, with the new last node) */
                        outbuf =
                            calloc(state->give_depth + 1,
                                   sizeof(state->tour[0]));
                        for (i = 0; i < state->give_depth; i++) {
                                outbuf[i] = state->tour[i];
                        }
                        outbuf[state->give_depth] =
                            state->last_started[state->give_depth];

                        /* synchronous send, because the receiver should be already waiting for the reply (and we want to free the memory) */
                        fprintf(stdout, "%d: Giving work to %d:", state->myrank,
                                status.MPI_SOURCE);
                        for (i = 0; i < state->give_depth + 1; i++) {
                                fprintf(stdout, " %2d", outbuf[i]);
                        }
                        fprintf(stdout, "\n");

                        MPI_Send(outbuf, state->give_depth + 1, MPI_INT,
                                 status.MPI_SOURCE, WORK_ACK_TAG,
                                 MPI_COMM_WORLD);

                        free(outbuf);

                        /* update our token */
                        if (state->myrank > status.MPI_SOURCE)
                                state->term_token.mycolor = BLACK;
                }
        }

        if (work_deny) {
                /* no, we don 't have any work to give away */
                fprintf(stdout,
                        "%d: Denying work request from %d (ws:%d, gd:%d, nn:%d)\n",
                        state->myrank, status.MPI_SOURCE, state->work_state,
                        state->give_depth, next_node);
                MPI_Send(&work_deny, 1, MPI_INT, status.MPI_SOURCE,
                         WORK_ACK_TAG, MPI_COMM_WORLD);
        }
}

void request_work(struct tsp_state *state)
{
        int got_work = 0;
        int msg_size, i;
        MPI_Status status;
        MPI_Request req;
        int initial_partner = state->work_partner;
        int reply_available;

        do {
                if (state->work_partner != state->myrank) {
//                      printf("%d: Sending work request to %d\n", state->myrank, state->work_partner);
                        MPI_Isend(&msg_size, 1, MPI_INT, state->work_partner,
                                  WORK_REQ_TAG, MPI_COMM_WORLD, &req);

                        reply_available = 0;
                        while (!reply_available) {
                                MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
                                          MPI_COMM_WORLD, &status);
                                service_pending_messages(state);

                                /* check for request reply */
                                MPI_Iprobe(state->work_partner, WORK_ACK_TAG,
                                           MPI_COMM_WORLD, &reply_available,
                                           &status);
                        }

                        MPI_Get_count(&status, MPI_INT, &msg_size);

                        MPI_Recv(&state->tour, msg_size, MPI_INT,
                                 state->work_partner, WORK_ACK_TAG,
                                 MPI_COMM_WORLD, &status);
                        if (msg_size > 1) {
//                              printf("%d: Got work from %d\n", state->myrank, state->work_partner);
                                state->work_state = WORKING;
                                state->give_depth = state->subtree_depth =
                                    msg_size;
                                state->len = msg_size;
                                state->last_started[state->give_depth] = 0;

                                state->cost = 0;
                                for (i = 1; i < state->len; i++) {
                                        state->cost +=
                                            CELL(state->matrix,
                                                 state->tour[i - 1],
                                                 state->tour[i]);
                                }

                                got_work = 1;
                        }
                }

                state->work_partner =
                    (state->work_partner + 1) % state->num_procs;

        }
        while (state->work_partner != initial_partner && !got_work);
}

void send_ub_message(struct tsp_state *state)
{
        int i;
        MPI_Request request;

        for (i = 0; i < state->num_procs; i++) {
                if (i == state->myrank)
                        continue;
                MPI_Isend(&state->ub, 1, MPI_INT, i, UB_TAG, MPI_COMM_WORLD,
                          &request);
        }
}
```