CS566 Parallel Processing
Assignment 04
The Travelling Salesman Problem

Camillo Lugaresi and Cosmin Stroe

Department of Computer Science
University of Illinois at Chicago

December 5, 2011

For our assignment, we implemented a depth first search (DFS) based solution for the travelling salesman problem (TSP), using the branch and bound optimization to keep a global shortest cycle cost and to prune the search tree of paths which cannot be optimal. This ensures that our algorithm will exaust the search space and that the solution it finds is the most optimal one, while not wasting execution time by checking suboptimal paths.

1 Algorithm Details and Formulations

1.1 TSPLIB95

As inputs to our program we used the datasets part of the TSPLIB95, which is a collection of TSP problems in an easy to read format. We only used symmetric TSP problems, as those are more challenging than asymmetric TSP problems. Also, the problems in TSPLIB95 have solutions which are published on the website[1], making it easy to check the correctness of our algorithms.

As part of our work, we implemented a parser for the TSPLIB95 datafiles which reads the TSP datafile and produces a distance matrix. The matrix is then used for looking up the distances between nodes, as needed by our algorithm.

1.2 Depth First Search: Branch and Bound

The solution space of the TSP problem with n nodes can be represented as a tree of depth $d = n$. At each node in the tree, there are $n - d$ possible choices for branching, and each one must be explored or pruned. A DFS traversal of the tree starting from the root of the tree is necessary since the cycle cost can only be recorded once the algorithm reaches a leaf node. When reaching a leaf node, we record the cost of the cycle if it is lower than our current lowest cycle cost.

We then backtrack to the parent node and choose the next possible node and search that branch; but we do this only if the cumulative cost of the current path from root to the current node is less than the lowest cycle cost. Otherwise we abandon (prune) this branch, as we know we will not be able to obtain an optimal cycle from this branch.

In our algorithm, we do not maintain the tree in memory; we only keep track of the current path from root, as this is all the information we require when running our algorithm. We also keep track of the last visited node at every level. The last visited node is updated when work is given away to other processors, so that the current processor will not explore a branch which has been assigned to another computer. Using the current path from root, and the last visited node at every level, we can compute which node to visit next.

Each processor in our algorithm is given a search space which to exaust. At the beginning, only node 0 has work, and the other processors must ask node 0 for work. When work is given away, that branch is marked as visited on the

[1]`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

current computer so that it is not traversed and also that it is not given away as work to another requester. When a cycle is found that is shorter than the global shortest cycle, the new cost is broadcast to all the processors in the cluster, so that they better prune their search spaces.

1.3 Load Balancing: Asynchronous Round Robin

As processors exaust the search space assigned to them, they must ask for work to processors which are still working traversing their search space. For the purpose of load balancing, we use the asynchronous round robin method of keeping a variable on our every processor with its next work partner from which to request work. Once a work partner is asked for work, the variable is incremented to the next computer in a round robin fashion. This is done asynchronously for each processor, so the drawback to this approach is that multiple processors can be asking for work to the same exact processors, however because we need to traverse the entire space this does not affect our algorithm.

1.4 Asynchronous MPI Messages

All of our communication, except for receiving work, is done asynchronously. This is required because the processors cannot be synchronized in the servicing of their messages, and synchronous messages would result in idle time. Receiving work is synchronous because after the work request is sent, we have nothing to do but to wait for the answer of the work request. All of the message types we pass, shown in table 1, are tagged so that we me identify what type of message it is and invoke the correct message processing function.

Once termination has been detected by the root processor (described in section 1.5), a best path request is sent to the processor that broadcast the last upper bound in order to receive the actual path of the best cycle. When the root processor received the best path, it sends a termination signal message to all other processors in the cluster and exits the program. Upon receiving the termination signal, the processors service their pending messages and exit the program.

1.5 Termination Detection

For termination detection we use Djikstra's Token Termination Detection algorithm. In order to simplify our message handing code, the token termination detection is only initiated by the root node. All the other processors pass on the token but will not initiate a token request. Instead, they wait for a termination message which is sent from the root processor upon a successful termination detection.

Message Type	MPI Tag	Description
Upper Bound Broadcast	`UB_TAG (1)`	Broadcast the new global shortest cycle path.
Work Request	`WORK_REQ_TAG (2)`	Request work.
Work Acknowledgement	`WORK_ACQ_TAG (3)`	Respond to a work request, either deny or a path describing the branch to be explored.
Token	`TOKEN_TAG (4)`	Pass the token used in termination detection.
Termination Signal	`TERMINATION_TAG (5)`	Passed when termination is detected.
Best Path Request	`BEST_PATH_TAG (6)`	Reply to a best path request with the path of the best cycle.
Best Path Transmit	`BEST_PATH_REQ_TAG (7)`	Request the path of the best cycle stored on a processor.

Table 1: The message types and their tags handled in our communication.

Num. threads	burma14	gr17	gr21	gr24
1				
4				
8				
16				

2 Input and parameters

We ran our experiments using problems of sizes $n = \{14, 17, 21, 24\}$, namely burma14, gr17, gr21, and gr24. For our processor size we consider the total number of threads running, not differentiating between processors or cores. This is because communication is not a bottleneck in this algorithm and we can take full advantage of every core on a computer.

3 Results

4 Analysis and Lessons

```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;
```

```c
                                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);
        }
}
```

```c
/* ==================================================================
 *
 * CS 566 - Assignment 04
 * Camillo Lugaresi, Cosmin Stroe
 *
 * ================================================================== */

#define CELL(m,r,c) (((m)->data)[((m)->n)*(r) + (c)])

#define MIN_WORK_LEVELS 5
#define WORK_SLICE 10000000

struct prefix_list_node {
        struct prefix_list_node *next;
        int prefix_len;
        int prefix[];
};

struct termination_state {
        MPI_Request request;    /* Make sure we don't modify the token while it's still being transmitted
*/
        int token;              /* whether I have the token, and what color it is */
        int mycolor;            /* what color I am */
};

struct tsp_state {
        int num_procs;
        int myrank;
        int cost;               /* the cost of this tour */
        int ub;                 /* the shortest path found so far (globally) */
        int ub_rank;
        int give_depth;         /* the minimum depth from which we can give away work */
        int subtree_depth;
        int *last_started;      /* an array of the last started node values at a certain depth */
        int len;                /* length of the tour so far */
        struct tsp_matrix *matrix;      /* the adjacency matrix of our TSP graph */
        struct termination_state term_token;
        int work_state;
        int work_partner;
        int *best_tour;
        int *send_buf;
        MPI_Request work_req;
        double work_time;
        double total_time;
        int tour[];             /* an array of node numbers, in the order which they are visited */
        /* tour must be the last field, since it's variable size! */
};

#define NO_TOKEN 0
#define WHITE 1
#define BLACK 2

#define NEED_WORK 0
#define WORKING 1
#define IDLE 2
#define QUIT 3
#define WORK_REQ_PENDING 4

struct tsp_state *tsp_state_alloc(struct tsp_matrix *matrix);

#define UB_TAG 1
#define WORK_REQ_TAG 2
#define WORK_ACK_TAG 3
#define TOKEN_TAG 4
#define TERMINATION_TAG 5
#define BEST_PATH_TAG 6
#define BEST_PATH_REQ_TAG 7
```

```c
void tsp(struct tsp_state *state);
void do_work(struct tsp_state *state);
void service_work_request(struct tsp_state *state, MPI_Status status);
void service_pending_messages(struct tsp_state *state);
void request_work(struct tsp_state *state);
void send_ub_message(struct tsp_state *state);
void send_token(struct tsp_state *state);
```

```c
/* ************************************************
   This file contains the routines for reading the
   TSPLIB95 files into an adjacency matrix.
   ************************************************ */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <strings.h>

#define _USE_MATH_DEFINES
#include <math.h>

#include "tsplib95.h"

#define NAME_FIELD "NAME"
#define DIM_FIELD "DIMENSION"
#define EW_TYPE_FIELD "EDGE_WEIGHT_TYPE"
#define EW_FORMAT_FIELD "EDGE_WEIGHT_FORMAT"
#define EW_SECTION_FIELD "EDGE_WEIGHT_SECTION"

#define EW_TYPE_EXPLICIT "EXPLICIT"
#define EW_FORMAT_LDR "LOWER_DIAG_ROW"

#define EW_EOF "EOF"

#define RRR 6378.388

/* Parse a TSPLIB95 file and return the adjacency matrix. */
void parse_matrix_from_file(struct tsp_matrix *matrix, FILE *file) {
        char line[1000];

        char *name, *ew_type, *ew_format;

        name = calloc( 100, sizeof(&name));
        ew_type = calloc( 100, sizeof(&ew_type));
        ew_format = calloc( 100, sizeof(&ew_format));

        while( fgets( line, sizeof(line), file) != NULL ) {
                if( strncmp(line, NAME_FIELD, strlen(NAME_FIELD)) == 0 ) {
                        sscanf(line, "NAME: %s", matrix->name);
                }
                else if( strncmp( line, DIM_FIELD, strlen(DIM_FIELD)) == 0 ) {
                        sscanf(line, "DIMENSION: %d", &matrix->n);
                        /* allocate the matrix */
                        matrix->data = calloc( matrix->n*matrix->n, sizeof(*matrix->data));
                }
                else if( strncmp( line, EW_TYPE_FIELD, strlen(EW_TYPE_FIELD)) == 0 ) {
                        sscanf(line, "EDGE_WEIGHT_TYPE: %s", ew_type);
                }
                else if( strncmp( line, EW_FORMAT_FIELD, strlen(EW_FORMAT_FIELD)) == 0 ) {
                        sscanf(line, "EDGE_WEIGHT_FORMAT: %s", ew_format);
                }
                else if( strncmp( line, EW_SECTION_FIELD, strlen(EW_SECTION_FIELD)) == 0 ||
                         strncmp( line, "NODE_COORD_SECTION", strlen("NODE_COORD_SECTION")) == 0 ) {
                        /* parse the data */
                        if( strncmp(ew_type, EW_TYPE_EXPLICIT, strlen(EW_TYPE_EXPLICIT)) == 0 &&
                            strncmp(ew_format, EW_FORMAT_LDR, strlen(EW_FORMAT_LDR)) == 0 ) {
                                parse_explicit_lowerdiagrow(matrix, file);
                        }
                        else if( strncmp(ew_type, "GEO", 3) == 0 &&
                                 strncmp(ew_format, "FUNCTION", 8) == 0 ) {
                                parse_geo_function(matrix, file);
                        }
                        else {
                                printf("PARSE ERROR: Don't know how to read %s, %s.\n", ew_type,
ew_format);
                                exit(1);
                        }
                }
        }

        /*
        printf("The file name is: %s\n", matrix->name);
        printf("The dimension is: %d\n", matrix->n);
        printf("The edge weight type is: %s\n", ew_type);
        printf("The edge weight format is: %s\n", ew_format);
        */

        return;
}

/* PARSE: GEO, FUNCTION */
void parse_geo_function(struct tsp_matrix *matrix, FILE *file) {

        char line[1000];
        int token_num = 0;
        int row_num = 0;
        int i, j;

        double *x, *y, *latitude, *longitude;
        double deg, min, q1, q2, q3;

        x = calloc( matrix->n, sizeof(*x));
        y = calloc( matrix->n, sizeof(*y));
        latitude = calloc( matrix->n, sizeof(*latitude));
        longitude = calloc( matrix->n, sizeof(*longitude));

        i = 0;
        while( fgets( line, sizeof(line), file) != NULL ) {

                if( strncmp(line, EW_EOF, strlen(EW_EOF)) == 0 ) break;

                char *token = strtok(line, " ");

                j = 0;
                while( token != NULL && strcmp(token,"\n") != 0 ) {
                        if( strlen(token) == 0 ) {
                                token = strtok(NULL, " ");
                                continue;
                        }


                        if( j == 1 ) {
                                x[i] = atof(token); // x coordinate
                        }

                        if( j == 2 ) {
                                y[i] = atof(token); // y coordinate
                        }

                        token = strtok( NULL, " ");
                        j++;
                }
                i++;
        }

        for( i = 0; i < matrix->n; i++ ) {
                deg = (int) ( x[i] ); // degrees (integer part)
                min = x[i] - deg;   // minutes (decimal part)
                latitude[i] = 3.141592 * ( deg + 5.0 * min / 3.0 ) / 180.0;

                deg = (int) ( y[i] ); // degrees (integer part)
                min = y[i] - deg;   // minutes (decimal part)
                longitude[i] = 3.141592 * ( deg + 5.0 * min / 3.0 ) / 180.0;
        }

        for( i = 0; i < matrix->n; i++ ) {
                for( j = 0; j < matrix->n; j++ ) {
                        if( i == j ) continue; // diagonals are 0

                        q1 = cos( longitude[i] - longitude[j] );
                        q2 = cos( latitude[i] - latitude[j] );
                        q3 = cos( latitude[i] + latitude[j] );

                        matrix->data[i*matrix->n + j] = (int) ( RRR * acos( 0.5*( (1.0+q1)*q2 - (1.0-
q1)*q3 ) ) + 1.0 );
                }
        }


        free(x);
        free(y);
        free(latitude);
        free(longitude);

}

void parse_explicit_lowerdiagrow(struct tsp_matrix *matrix, FILE *file) {

        char line[1000];
        int token_num = 0;
        int row_num = 0;
        int i, j;

        while( fgets( line, sizeof(line), file) != NULL ) {

                if( strncmp(line, EW_EOF, strlen(EW_EOF)) == 0 ) break;

                char *token = strtok(line, " ");
                while( token != NULL && strlen(token) != 0 && strcmp(token,"\n") != 0 ) {

                        /* printf("token: %s, %ld\n", token, strlen(token)); */

                        matrix->data[row_num*matrix->n + token_num] = atoi(token);

                        if( token_num == row_num ) {
                                token_num = 0;
                                row_num++;
                        } else {
                                token_num++;
                        }

                        token = strtok( NULL, " ");
                }
        }

        /* ok, now copy the lower diagonal to the upper diagonal */
        for( i = 0; i < matrix->n; i++ ) {
                for( j = i+1; j < matrix->n; j++ ) {
                        matrix->data[i*matrix->n + j] = matrix->data[j*matrix->n + i];
                }
        }
}


void print_matrix(struct tsp_matrix *m)
{
        int i;
        int count = m->n*m->n;
        int *p = m->data;

        for (i = 1; i <= count; i++) {
                printf("%4d", *p);
                if (i % m->n == 0) printf("\n");
                else printf(" ");
                p++;
        }
}
```

```c
struct tsp_matrix {
        int n;
        int *data;
        char name[100];
};

void parse_matrix_from_file(struct tsp_matrix *matrix, FILE *file);
void parse_geo_function(struct tsp_matrix *matrix, FILE *file);
void parse_explicit_lowerdiagrow(struct tsp_matrix *matrix, FILE *file);
void print_matrix(struct tsp_matrix *m);
```

```
.PHONY : all

all : tsplib95 tsp

tsp: tsplib95 tsp.c
        mpicc -o tsp tsp.c tsplib95.o -lm

tsplib95: tsplib95.c tsplib95.h
        mpicc -c -o tsplib95.o tsplib95.c

clean:
        rm -f tsplib95.o tsp
```