```c
/* File:     tsp_iter1.c
 * Purpose:  Use iterative depth-first search to solve an instance of the
 *           travelling salesman problem.  This version attempts to
 *           doesn't make copies of the tours when they're pushed onto
 *           the stack.  It also uses macros for push and pop.
 *
 * Compile:  gcc -g -Wall -o tsp_iter1 tsp_iter1.c
 *           Needs timer.h
 * Usage:    tsp_iter1 <matrix_file>
 *
 * Input:    From a user-specified file, the number of cities
 *           followed by the costs of travelling between the
 *           cities organized as a matrix:  the cost of
 *           travelling from city i to city j is the ij entry.
 *           Costs are nonnegative ints.  Diagonal entries are 0.
 * Output:   The best tour found by the program and the cost
 *           of the tour.
 *
 * Notes:
 * 1.  Costs and cities are non-negative ints.
 * 2.  Program assumes the cost of travelling from a city to
 *     itself is zero, and the cost of travelling from one
 *     city to another city is positive.
 * 3.  Note that costs may not be symmetric:  the cost of travelling
 *     from A to B, may, in general, be different from the cost
 *     of travelling from B to A.
 * 4.  Salesperson's home town is 0.
 * 5.  The digraph is stored as an adjacency matrix, which is
 *     a one-dimensional array:  digraph[i][j] is computed as
 *     digraph[i*n + j]
 *
 * IPP:  Section 6.2.2 (pp. 303 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "timer.h"

const int INFINITY = 1000000;
const int NO_CITY = -1;
const int UNUSED = -2;
const int FALSE = 0;
const int TRUE = 1;

typedef int city_t;
typedef int cost_t;
typedef struct {
   city_t* cities; /* Cities in partial tour           */
   int count;      /* Number of cities in partial tour */
   cost_t cost;    /* Cost of partial tour             */
```

```
51 } tour_struct;
52 typedef tour_struct* tour_t;
53 #define City_count(tour) (tour->count)
54 #define Tour_cost(tour) (tour->cost)
55 #define Last_city(tour) (tour->cities[(tour->count)-1])
56 #define Tour_city(tour,i) (tour->cities[(i)])
57
58 /* Each time a recursive call is made, a new city is added to the
59  * current tour.  The stack stores the cities. */
60 typedef struct {
61    city_t* list;
62    int list_sz;
63 }  stack_struct;
64 typedef stack_struct* my_stack_t;
65 #define EMPTY(stack) (stack->list_sz == 0?  TRUE: FALSE)
66 #define PUSH(stack,city) {stack->list[stack->list_sz] = city; \
67                           (stack->list_sz)++;}
68
69 /* Global Vars:  Except for best_tour, all are constant after initializatio
70 int n;  /* Number of cities in the problem */
71 city_t home_town = 0;
72 tour_t best_tour;
73 cost_t* digraph;
74 #define Cost(city1, city2) (digraph[city1*n + city2])
75
76 void Usage(char* prog_name);
77 void Read_digraph(FILE* digraph_file);
78 void Print_digraph(void);
79
80 void Iterative_dfs(void);
81 void Print_tour(tour_t tour, char* title);
82 int  Best_tour(tour_t tour);
83 void Update_best_tour(tour_t tour);
84 void Copy_tour(tour_t tour1, tour_t tour2);
85 void Add_city(tour_t tour, city_t);
86 void Remove_last_city(tour_t tour);
87 int  Feasible(tour_t tour, city_t city);
88 int  Visited(tour_t tour, city_t city);
89 void Init_tour(tour_t tour, cost_t cost);
90 tour_t Alloc_tour(void);
91 void Free_tour(tour_t tour);
92
93 my_stack_t Init_stack(void);
94 void Push(my_stack_t stack, city_t city);
95 city_t Pop(my_stack_t stack);
96 int  Empty(my_stack_t stack);
97 void Free_stack(my_stack_t stack);
98
99 /*--------------------------------------------------------------------*/
100 int main(int argc, char* argv[]) {
```

```
101     FILE* digraph_file;
102     double start, finish;
103
104     if (argc != 2) Usage(argv[0]);
105     digraph_file = fopen(argv[1], "r");
106     if (digraph_file == NULL) {
107        fprintf(stderr, "Can't open %s\n", argv[1]);
108        Usage(argv[0]);
109     }
110     Read_digraph(digraph_file);
111     fclose(digraph_file);
112 #   ifdef DEBUG
113     Print_digraph();
114 #   endif
115
116     best_tour = Alloc_tour();
117     Init_tour(best_tour, INFINITY);
118 #   ifdef DEBUG
119     Print_tour(best_tour, "Best tour");
120     printf("City count = %d\n",  City_count(best_tour));
121     printf("Cost = %d\n\n", Tour_cost(best_tour));
122 #   endif
123
124     GET_TIME(start);
125     Iterative_dfs();
126     GET_TIME(finish);
127
128     Print_tour(best_tour, "Best tour");
129     printf("Cost = %d\n", best_tour->cost);
130     printf("Elapsed time = %e seconds\n", finish-start);
131
132     Free_tour(best_tour);
133     free(digraph);
134     return 0;
135 }   /* main */
136
137 /*---------------------------------------------------------------------
138  * Function:  Init_tour
139  * Purpose:   Initialize the data member of allocated tour
140  * In args:
141  *    cost:   initial cost of tour
142  * Global in:
143  *    n:      number of cities in TSP
144  * Out arg:
145  *    tour
146  */
147 void Init_tour(tour_t tour, cost_t cost) {
148     int i;
149
150     tour->cities[0] = 0;
```

```
151    for (i = 1; i <= n; i++) {
152       tour->cities[i] = NO_CITY;
153    }
154    tour->cost = cost;
155    tour->count = 1;
156 }  /* Init_tour */
157
158
159 /*--------------------------------------------------------------------
160  * Function:  Usage
161  * Purpose:   Inform user how to start program and exit
162  * In arg:    prog_name
163  */
164 void Usage(char* prog_name) {
165    fprintf(stderr, "usage: %s <digraph file>\n", prog_name);
166    exit(0);
167 }  /* Usage */
168
169 /*--------------------------------------------------------------------
170  * Function:  Read_digraph
171  * Purpose:   Read in the number of cities and the digraph of costs
172  * In arg:    digraph_file
173  * Globals out:
174  *    n:        the number of cities
175  *    digraph:  the matrix file
176  */
177 void Read_digraph(FILE* digraph_file) {
178    int i, j;
179
180    fscanf(digraph_file, "%d", &n);
181    if (n <= 0) {
182       fprintf(stderr, "Number of vertices in digraph must be positive\n");
183       exit(-1);
184    }
185    digraph = malloc(n*n*sizeof(cost_t));
186
187    for (i = 0; i < n; i++)
188       for (j = 0; j < n; j++) {
189          fscanf(digraph_file, "%d", &digraph[i*n + j]);
190          if (i == j && digraph[i*n + j] != 0) {
191             fprintf(stderr, "Diagonal entries must be zero\n");
192             exit(-1);
193          } else if (i != j && digraph[i*n + j] <= 0) {
194             fprintf(stderr, "Off-diagonal entries must be positive\n");
195             fprintf(stderr, "diagraph[%d,%d] = %d\n", i, j, digraph[i*n+j])
196             exit(-1);
197          }
198       }
199 }  /* Read_digraph */
200
```

```
201
202 /*--------------------------------------------------------------------
203  * Function:  Print_digraph
204  * Purpose:   Print the number of cities and the digraphrix of costs
205  * Globals in:
206  *    n:        number of cities
207  *    digraph:  digraph of costs
208  */
209 void Print_digraph(void) {
210    int i, j;
211
212    printf("Order = %d\n", n);
213    printf("Matrix = \n");
214    for (i = 0; i < n; i++) {
215       for (j = 0; j < n; j++)
216          printf("%2d ", digraph[i*n+j]);
217       printf("\n");
218    }
219    printf("\n");
220 }  /* Print_digraph */
221
222
223 /*--------------------------------------------------------------------
224  * Function:    Iterative_dfs
225  * Purpose:     Use a stack variable to implement an iterative version
226  *              of depth-first search
227  * In arg:
228  *    tour:     partial tour of cities visited so far (just city 0)
229  * Globals in:
230  *    n:        total number of cities in the problem
231  * Notes:
232  * 1. The Update_best_tour function will modify the global var
233  *    best_tour
234  */
235 void Iterative_dfs(void) {
236    city_t nbr, city;
237    my_stack_t stack;
238    tour_t curr_tour;
239
240    curr_tour = Alloc_tour();
241    Init_tour(curr_tour, 0);
242 #  ifdef DEBUG
243    Print_tour(curr_tour, "Starting tour");
244    printf("City count = %d\n",  City_count(curr_tour));
245    printf("Cost = %d\n\n", Tour_cost(curr_tour));
246 #  endif
247
248    stack = Init_stack();
249    PUSH(stack, NO_CITY);
250    for (city = n-1; city >= 1; city--)
```

```
251       PUSH(stack, city);
252    while (!EMPTY(stack)) {
253       // Next two statements pop stack;
254       city = stack->list[stack->list_sz-1];
255       (stack->list_sz)--;
256       if (city == NO_CITY) // End of child list, back up
257          Remove_last_city(curr_tour);
258       else if (Feasible(curr_tour, city)) { // Only check cost (visited che
259                                             //    when city was pushed)
260          Add_city(curr_tour, city);
261 #        ifdef DEBUG
262          Print_tour(curr_tour, "New tour");
263          printf("\n");
264 #        endif
265          if (City_count(curr_tour) == n) {
266             if (Best_tour(curr_tour))
267                Update_best_tour(curr_tour);
268             Remove_last_city(curr_tour);
269          } else {
270             PUSH(stack, NO_CITY);
271             for (nbr = n-1; nbr >= 1; nbr--)
272                if (!Visited(curr_tour, nbr))
273                   PUSH(stack, nbr);
274          }
275       }/* if Feasible */
276    }  /* while !Empty */
277    Free_stack(stack);
278    Free_tour(curr_tour);
279 }  /* Iterative_dfs */
280
281 /*------------------------------------------------------------------
282  * Function:    Best_tour
283  * Purpose:     Determine whether addition of the hometown to the
284  *              n-city input tour will lead to a best tour.
285  * In arg:
286  *    tour:     tour visiting all n cities
287  * Ret val:
288  *    TRUE if best tour, FALSE otherwise
289  */
290 int Best_tour(tour_t tour) {
291    cost_t cost_so_far = Tour_cost(tour);
292    city_t last_city = Last_city(tour);
293
294    if (cost_so_far + Cost(last_city, home_town) < Tour_cost(best_tour))
295       return TRUE;
296    else
297       return FALSE;
298 }  /* Best_tour */
299
300 /*------------------------------------------------------------------
```

```
301  * Function:    Update_best_tour
302  * Purpose:     Replace the existing best tour with the input tour +
303  *              hometown
304  * In arg:
305  *    tour:     tour that's visited all n-cities
306  * Global out:
307  *    best_tour:  the current best tour
308  * Note:
309  *    The input tour hasn't had the home_town added as the last
310  *    city before the call to Update_best_tour.  So we call
311  *    Add_city(best_tour, hometown) before returning.
312  */
313 void Update_best_tour(tour_t tour) {
314    Copy_tour(tour, best_tour);
315    Add_city(best_tour, home_town);
316 }  /* Update_best_tour */
317
318
319 /*--------------------------------------------------------------------
320  * Function:   Copy_tour
321  * Purpose:    Copy tour1 into tour2
322  * In arg:
323  *    tour1
324  * Out arg:
325  *    tour2
326  */
327 void Copy_tour(tour_t tour1, tour_t tour2) {
328 // int i;
329
330 // for (i = 0; i <= n; i++)
331 //   tour2->cities[i] =  tour1->cities[i];
332    memcpy(tour2->cities, tour1->cities, (n+1)*sizeof(city_t));
333    tour2->count = tour1->count;
334    tour2->cost = tour1->cost;
335 }  /* Copy_tour */
336
337 /*--------------------------------------------------------------------
338  * Function:  Add_city
339  * Purpose:   Add city to the end of tour
340  * In arg:
341  *    new_city
342  * In/out arg:
343  *    tour
344  */
345 void Add_city(tour_t tour, city_t new_city) {
346    if (City_count(tour) != 0) {
347       city_t old_last_city = Last_city(tour);
348       tour->cost += Cost(old_last_city,new_city);
349    }
350    tour->cities[tour->count] = new_city;
```

```
351     (tour->count)++;
352  }  /* Add_city */
353
354  /*-------------------------------------------------------------------
355   * Function:  Remove_last_city
356   * Purpose:   Remove last city from end of tour
357   * In/out arg:
358   *    tour
359   * Note:      Function assumes at least one city in tour
360   */
361  void Remove_last_city(tour_t tour) {
362     city_t old_last_city = Last_city(tour);
363     city_t new_last_city;
364
365     tour->cities[tour->count-1] = NO_CITY;
366     (tour->count)--;
367     if (City_count(tour) > 0) {
368        new_last_city = Last_city(tour);
369        tour->cost -= Cost(new_last_city,old_last_city);
370     } else {
371        tour->cost = 0;
372     }
373  }  /* Remove_last_city */
374
375  /*-------------------------------------------------------------------
376   * Function:  Feasible
377   * Purpose:   Check whether nbr could possibly lead to a better
378   *            solution if it is added to the current tour.  The
379   *            function checks whether nbr has already been visited
380   *            in the current tour, and, if not, whether adding the
381   *            edge from the current city to nbr will result in
382   *            a cost less than the current best cost.
383   * In args:   All
384   * Global in:
385   *    best_tour
386   * Return:    TRUE if the nbr can be added to the current tour.
387   *            FALSE otherwise
388   */
389  int Feasible(tour_t tour, city_t city) {
390     city_t last_city = Last_city(tour);
391
392     if (Tour_cost(tour) + Cost(last_city,city) < Tour_cost(best_tour))
393        return TRUE;
394     else
395        return FALSE;
396  }  /* Feasible */
397
398
399  /*-------------------------------------------------------------------
400   * Function:   Visited
```

```
401  * Purpose:    Use linear search to determine whether city has already
402  *             been visited on the current tour.
403  * In args:    All
404  * Return val: TRUE if city has already been visited.
405  *             FALSE otherwise
406  */
407 int Visited(tour_t tour, city_t city) {
408    int i;
409 
410    for (i = 1; i < City_count(tour); i++)
411       if ( Tour_city(tour,i) == city ) return TRUE;
412    return FALSE;
413 }  /* Visited */
414 
415 
416 /*------------------------------------------------------------------
417  * Function:  Print_tour
418  * Purpose:   Print a tour
419  * In args:   All
420  */
421 void Print_tour(tour_t tour, char* title) {
422    int i;
423 
424    printf("%s:\n", title);
425    for (i = 0; i < City_count(tour); i++)
426       printf("%d ", Tour_city(tour,i));
427    printf("\n");
428 }  /* Print_tour */
429 
430 /*------------------------------------------------------------------
431  * Function:  Alloc_tour
432  * Purpose:   Allocate memory for a tour and its members
433  * Global in: n, number of cities
434  * Ret val:   Pointer to a tour_struct with storage allocated for its
435  *            members
436  */
437 tour_t Alloc_tour(void) {
438    tour_t tmp = malloc(sizeof(tour_struct));
439    tmp->cities = malloc((n+1)*sizeof(city_t));
440    return tmp;
441 }  /* Alloc_tour */
442 
443 /*------------------------------------------------------------------
444  * Function:  Free_tour
445  * Purpose:   Free tour and its member variables
446  * Out arg:   tour
447  */
448 void Free_tour(tour_t tour) {
449    free(tour->cities);
450    free(tour);
```

```
451 }  /* Free_tour */
452
453 /*--------------------------------------------------------------------
454  * Function: Init_stack
455  * Purpose:  Allocate storage for a new stack and initialize members
456  * Out arg:  stack_p
457  */
458 my_stack_t Init_stack(void) {
459    int i;
460
461    my_stack_t stack = malloc(sizeof(stack_struct));
462    stack->list = malloc(n*n*sizeof(city_t));
463    for (i = 0; i < n*n; i++)
464       stack->list[i] = UNUSED;
465    stack->list_sz = 0;
466
467    return stack;
468 }  /* Init_stack */
469
470
471 /*--------------------------------------------------------------------
472  * Function:    Push
473  * Purpose:     Add a new city to the top of the stack
474  * In arg:      city
475  * In/out arg:  stack
476  * Error:       If the stack is full, print an error and exit
477  */
478 void Push(my_stack_t stack, city_t city) {
479    if (stack->list_sz == n*n) {
480       fprintf(stderr, "Stack overflow!\n");
481       exit(-1);
482    }
483    stack->list[stack->list_sz] = city;
484    (stack->list_sz)++;
485 }  /* Push */
486
487
488 /*--------------------------------------------------------------------
489  * Function:  Pop
490  * Purpose:   Reduce the size of the stack by returning the top
491  * In arg:    stack
492  * Ret val:   The tour on the top of the stack
493  * Error:     If the stack is empty, print a message and exit
494  */
495 city_t Pop(my_stack_t stack) {
496    city_t tmp;
497
498    if (stack->list_sz == 0) {
499       fprintf(stderr, "Trying to pop empty stack!\n");
500       exit(-1);
```

```
501    }
502    tmp = stack->list[stack->list_sz-1];
503    stack->list[stack->list_sz-1] = UNUSED;
504    (stack->list_sz)--;
505    return tmp;
506 }  /* Pop */
507
508
509 /*--------------------------------------------------------------------
510  * Function:  Empty
511  * Purpose:   Determine whether the stack is empty
512  * In arg:    stack
513  * Ret val:   TRUE if empty, FALSE otherwise
514  */
515 int  Empty(my_stack_t stack) {
516    if (stack->list_sz == 0)
517       return TRUE;
518    else
519       return FALSE;
520 }  /* Empty */
521
522
523 /*--------------------------------------------------------------------
524  * Function:  Free_stack
525  * Purpose:   Free stack and its members
526  * Out arg:   stack
527  * Note:      Assumes stack is empty
528  */
529 void Free_stack(my_stack_t stack) {
530    free(stack->list);
531    free(stack);
532 }  /* Free_stack */
533
534
```