```c
/* File:     tsp_iter2.c
 *
 * Purpose:  Use iterative depth-first search to solve an instance of the
 *           travelling salesman problem.  This version pushes an entire
 *           copy of a tour onto the stack and it stores ``freed''
 *           tours in an "avail" stack.
 *
 * Compile:  gcc -g -Wall -o tsp_iter2 tsp_iter2.c
 * Usage:    tsp_iterative <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. 304 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 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             */
} tour_struct;
```

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

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

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

```
201             exit(-1);
202          } else if (i != j && digraph[i*n + j] <= 0) {
203             fprintf(stderr, "Off-diagonal entries must be positive\n");
204             fprintf(stderr, "diagraph[%d,%d] = %d\n", i, j, digraph[i*n+j])
205             exit(-1);
206          }
207       }
208 }  /* Read_digraph */
209
210
211 /*--------------------------------------------------------------------
212  * Function:  Print_digraph
213  * Purpose:   Print the number of cities and the digraphrix of costs
214  * Globals in:
215  *    n:        number of cities
216  *    digraph:  digraph of costs
217  */
218 void Print_digraph(void) {
219    int i, j;
220
221    printf("Order = %d\n", n);
222    printf("Matrix = \n");
223    for (i = 0; i < n; i++) {
224       for (j = 0; j < n; j++)
225          printf("%2d ", digraph[i*n+j]);
226       printf("\n");
227    }
228    printf("\n");
229 }  /* Print_digraph */
230
231
232 /*--------------------------------------------------------------------
233  * Function:    Iterative_dfs
234  * Purpose:     Use a stack variable to implement an iterative version
235  *              of depth-first search
236  * In arg:
237  *    tour:     partial tour of cities visited so far (just city 0)
238  * Globals in:
239  *    n:        total number of cities in the problem
240  * Notes:
241  * 1  The input tour is modified during execution of search,
242  *    but returned to its original state before returning.
243  * 2. The Update_best_tour function will modify the global var
244  *    best_tour
245  */
246 void Iterative_dfs(tour_t tour) {
247    city_t nbr;
248    my_stack_t stack;
249    tour_t curr_tour;
250
```

```
251    stack = Init_stack();
252    Push(stack, tour);
253    while (!Empty(stack)) {
254       curr_tour = Pop(stack);
255 #     ifdef DEBUG
256       printf("Popped tour = %p and %p\n", curr_tour, curr_tour->cities);
257       Print_tour(curr_tour, "Popped");
258       printf("\n");
259 #     endif
260       if (City_count(curr_tour) == n) {
261          if (Best_tour(curr_tour))
262             Update_best_tour(curr_tour);
263       } else {
264          for (nbr = n-1; nbr >= 1; nbr--)
265             if (Feasible(curr_tour, nbr)) {
266                Add_city(curr_tour, nbr);
267                Push(stack, curr_tour);
268                Remove_last_city(curr_tour);
269             }
270       }
271       Free_tour(curr_tour);
272    }
273    Free_stack(stack);
274 }  /* Iterative_dfs */
275 
276 /*--------------------------------------------------------------------
277  * Function:    Best_tour
278  * Purpose:     Determine whether addition of the hometown to the
279  *              n-city input tour will lead to a best tour.
280  * In arg:
281  *    tour:     tour visiting all n cities
282  * Ret val:
283  *    TRUE if best tour, FALSE otherwise
284  */
285 int Best_tour(tour_t tour) {
286    cost_t cost_so_far = Tour_cost(tour);
287    city_t last_city = Last_city(tour);
288 
289    if (cost_so_far + Cost(last_city, home_town) < Tour_cost(best_tour))
290       return TRUE;
291    else
292       return FALSE;
293 }  /* Best_tour */
294 
295 /*--------------------------------------------------------------------
296  * Function:    Update_best_tour
297  * Purpose:     Replace the existing best tour with the input tour +
298  *              hometown
299  * In arg:
300  *    tour:     tour that's visited all n-cities
```

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

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

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

```
451 /*------------------------------------------------------------------
452  * Function: Init_stack
453  * Purpose:  Allocate storage for a new stack and initialize members
454  * Out arg:  stack_p
455  */
456 my_stack_t Init_stack(void) {
457    int i;
458 
459    my_stack_t stack = malloc(sizeof(stack_struct));
460    stack->list = malloc(n*n*sizeof(tour_t));
461    for (i = 0; i < n*n; i++)
462       stack->list[i] = NULL;
463    stack->list_sz = 0;
464 
465    return stack;
466 }  /* Init_stack */
467 
468 
469 /*------------------------------------------------------------------
470  * Function:    Push_avail
471  * Purpose:     Store a tour in the available list
472  * In arg:      tour
473  * In/out Global:
474  */
475 void Push_avail(tour_t tour) {
476    if (avail->list_sz == n*n) {
477       fprintf(stderr, "Available stack overflow!\n");
478       free(tour->cities);
479       free(tour);
480    } else {
481 #     ifdef DEBUG
482       printf("In Push_avail, loc = %d, pushing %p and %p\n",
483             avail->list_sz, tour, tour->cities);
484       Print_tour(tour, "About to be pushed onto avail");
485       printf("\n");
486 #     endif
487       avail->list[avail->list_sz] = tour;
488       (avail->list_sz)++;
489    }
490 }  /* Push_avail */
491 
492 /*------------------------------------------------------------------
493  * Function:    Push
494  * Purpose:     Add a new tour to the top of the stack
495  * In arg:      tour
496  * In/out arg:  stack
497  * Error:       If the stack is full, print an error and exit
498  */
499 void Push(my_stack_t stack, tour_t tour) {
500    tour_t tmp;
```

```
501    if (stack->list_sz == n*n) {
502       fprintf(stderr, "Stack overflow!\n");
503       exit(-1);
504    }
505    tmp = Alloc_tour();
506    Copy_tour(tour, tmp);
507    stack->list[stack->list_sz] = tmp;
508    (stack->list_sz)++;
509 }  /* Push */
510
511
512 /*--------------------------------------------------------------------
513  * Function:  Pop
514  * Purpose:   Reduce the size of the stack by returning the top
515  * In arg:    stack
516  * Ret val:   The tour on the top of the stack
517  * Error:     If the stack is empty, print a message and exit
518  */
519 tour_t Pop(my_stack_t stack) {
520    tour_t tmp;
521
522    if (stack->list_sz == 0) {
523       fprintf(stderr, "Trying to pop empty stack!\n");
524       exit(-1);
525    }
526    tmp = stack->list[stack->list_sz-1];
527    stack->list[stack->list_sz-1] = NULL;
528    (stack->list_sz)--;
529    return tmp;
530 }  /* Pop */
531
532
533 /*--------------------------------------------------------------------
534  * Function:  Empty
535  * Purpose:   Determine whether the stack is empty
536  * In arg:    stack
537  * Ret val:   TRUE if empty, FALSE otherwise
538  */
539 int  Empty(my_stack_t stack) {
540    if (stack->list_sz == 0)
541       return TRUE;
542    else
543       return FALSE;
544 }  /* Empty */
545
546
547 /*--------------------------------------------------------------------
548  * Function:  Free_stack
549  * Purpose:   Free stack and its members
550  * Out arg:   stack
```

```
551  * Note:      Assumes stack is empty
552  */
553 void Free_stack(my_stack_t stack) {
554    free(stack->list);
555    free(stack);
556 }  /* Free_stack */
557 
558 /*---------------------------------------------------------------------
559  * Function:  Free_avail
560  * Purpose:   Free the stack of available tours
561  * Out arg:   avail
562  */
563 void Free_avail(void) {
564    int i;
565    tour_t tmp;
566 #  ifdef DEBUG
567    printf("In Free_avail, list_sz = %d\n", avail->list_sz);
568 #  endif
569    for (i = 0; i < avail->list_sz; i++) {
570       tmp = avail->list[i];
571       if (tmp != NULL) {
572 #        ifdef DEBUG
573          printf("In Free_avail, i = %d, attempting to free %p and %p\n",
574                i, tmp->cities, tmp);
575 #        endif
576          free(tmp->cities);
577          free(tmp);
578       }
579    }
580    free(avail->list);
581    free(avail);
582 }  /* Free_avail */
583 
```