

# NAG Library Function Document

## nag\_mip\_tsp\_simann (h03bbc)

### 1 Purpose

nag\_mip\_tsp\_simann (h03bbc) calculates an approximate solution to a symmetric travelling salesman problem using simulated annealing via a configuration free interface.

### 2 Specification

```
#include <nag.h>
#include <nagh.h>

void nag_mip_tsp_simann (Integer nc, const double dm[], double bound,
    double targc, Integer path[], double *cost, Integer *tmode,
    double alg_stats[], Integer state[], NagError *fail)
```

### 3 Description

nag\_mip\_tsp\_simann (h03bbc) provides a probabilistic strategy for the calculation of a near optimal path through a symmetric and fully connected distance matrix; that is, a matrix for which element  $(i, j)$  is the pairwise distance (also called the cost, or weight) between nodes (cities)  $i$  and  $j$ . This problem is better known as the Travelling Salesman Problem (TSP), and symmetric means that the distance to travel between two cities is independent of which is the destination city.

In the classical TSP, which this function addresses, a salesman wishes to visit a given set of cities once only by starting and finishing in a home city and travelling the minimum total distance possible. It is one of the most intensively studied problems in computational mathematics and, as a result, has developed some fairly sophisticated techniques for getting near-optimal solutions for large numbers of cities. nag\_mip\_tsp\_simann (h03bbc) adopts a very simple approach to try to find a reasonable solution, for moderately large problems. The function uses simulated annealing: a stochastic mechanical process in which the heating and controlled cooling of a material is used to optimally refine its molecular structure.

The material in the TSP is the distance matrix and a given state is represented by the order in which each city is visited—the path. This system can move from one state to a neighbouring state by selecting two cities on the current path at random and switching their places; the order of the cities in the path between the switched cities is then reversed. The cost of a state is the total cost of traversing its path; the resulting difference in cost between the current state and this new proposed state is called the delta; a negative delta indicates the proposal creates a more optimal path and a positive delta a less optimal path. The random selection of cities to switch uses random number generators (RNGs) from Chapter g05; it is thus necessary to initialize a state array for the RNG of choice (by a call to nag\_rand\_init\_repeatable (g05kfc) or nag\_rand\_init\_nonrepeatable (g05kge)) prior to calling nag\_mip\_tsp\_simann (h03bbc).

The simulation itself is executed in two stages. In the first stage, a series of sample searches through the distance matrix is conducted where each proposed new state is accepted, regardless of the change in cost (delta) incurred by applying the switches, and statistics on the set of deltas are recorded. These metrics are updated after each such sample search; the number of these searches and the number of switches applied in each search is dependent on the number of cities. The final collated set of metrics for the deltas obtained by the first stage are used as control parameters for the second stage. If no single improvement in cost is found during the first stage, the algorithm is terminated.

In the second stage, as before, neighbouring states are proposed. If the resulting delta is negative or causes no change the proposal is accepted and the path updated; otherwise moves are accepted based on a probabilistic criterion, a modified version of the Metropolis–Hastings algorithm.

The acceptance of some positive deltas (increased cost) reduces the probability of a solution getting trapped at a non-optimal solution where any single switch causes an increase in cost. Initially the acceptance criteria allow for relatively large positive deltas, but as the number of proposed changes

increases, the criteria become more stringent, allowing fewer positive deltas of smaller size to be accepted; this process is, within the realm of the simulated annealing algorithm, referred to as ‘cooling’. Further exploration of the system is initially encouraged by accepting non-optimal routes, but is increasingly discouraged as the process continues.

The second stage will terminate when:

- a solution is obtained that is deemed acceptable (as defined by supplied values);
- the algorithm will accept no further positive deltas and a set of proposed changes have resulted in no improvements (has cooled);
- a number of consecutive sets of proposed changes has resulted in no improvement.

## 4 References

Applegate D L, Bixby R E, Chvátal V and Cook W J (2006) *The Traveling Salesman Problem: A Computational Study* Princeton University Press

Cook W J (2012) *In Pursuit of the Traveling Salesman* Princeton University Press

Johnson D S and McGeoch L A The traveling salesman problem: A case study in local optimization *Local search in combinatorial optimization* (1997) 215–310

Press W H, Teukolsky S A, Vetterling W T and Flannery B P (2007) *Numerical Recipes The Art of Scientific Computing (3rd Edition)*

Rego C, Gamboa D, Glover F and Osterman C (2011) Traveling salesman problem heuristics: leading methods, implementations and latest advances *European Journal of Operational Research* **211 (3)** 427–441

Reinelt G (1994) *The Travelling Salesman. Computational Solutions for TSP Applications, Volume 840 of Lecture Notes in Computer Science* Springer-Verlag, Berlin Heidelberg New York

## 5 Arguments

1: **nc** – Integer *Input*

*On entry:* the number of cities. In the trivial cases **nc** = 1, 2 or 3, the function returns the optimal solution immediately with **tmode** = 0 (provided the relevant distance matrix entries are not negative).

*Constraint:* **nc** ≥ 1.

2: **dm[nc × nc]** – const double *Input*

**Note:** the (*i*, *j*)th element of the matrix is stored in **dm**[(*j* – 1) × **nc** + *i* – 1].

*On entry:* the distance matrix; each **dm**[(*j* – 1) × **nc** + *i* – 1] is the effective cost or weight between nodes *i* and *j*. Only the strictly upper half of the matrix is referenced.

*Constraint:* **dm**[(*j* – 1) × **nc** + *i* – 1] ≥ 0.0, for *j* = 2, 3, ..., **nc** and *i* = 1, 2, ..., *j* – 1.

3: **bound** – double *Input*

*On entry:* a lower bound on the solution. If the optimum is unknown set **bound** to zero or a negative value; the function will then calculate the minimum spanning tree for **dm** and use this as a lower bound (returned in **alg\_stats**[5]). If an optimal value for the cost is known then this should be used for the lower bound. A detailed discussion of relaxations for lower bounds, including the minimal spanning tree, can be found in Reinelt (1994).

4: **targc** – double *Input*

*On entry:* a measure of how close an approximation needs to be to the lower bound. The function terminates when a cost is found less than or equal to **bound** + **targc**. This argument is useful when an optimal value for the cost is known and supplied in **bound**. It may be sufficient to obtain

a path that is close enough (in terms of cost) to the optimal path; this allows the algorithm to terminate at that point and avoid further computation in attempting to find a better path.

If **targc** < 0, **targc** = 0 is assumed.

- 5: **path[nc]** – Integer *Output*  
*On exit:* the best path discovered by the simulation. That is, **path** contains the city indices in path order. If **fail.code**  $\neq$  0 on exit, **path** contains the indices 1 to **nc**.
- 6: **cost** – double \* *Output*  
*On exit:* the cost or weight of **path**. If **fail.code**  $\neq$  0 on exit, **cost** contains the largest model real number (see nag\_real\_max\_exponent (X02BLC)).
- 7: **tmode** – Integer \* *Output*  
*On exit:* the termination mode of the function (if **fail.code**  $\neq$  0 on exit, **tmode** is set to  $-1$ ):
- tmode** = 0  
 Optimal solution found, **cost** = **bound**.
- tmode** = 1  
 System temperature cooled. The algorithm returns a **path** and associated **cost** that does not attain, nor lie within **targc** of, the **bound**. This could be a sufficiently good approximation to the optimal **path**, particularly when **bound** + **targc** lies below the optimal **cost**.
- tmode** = 2  
 Halted by **cost** falling within the desired **targc** range of the **bound**.
- tmode** = 3  
 System stalled following lack of improvement.
- tmode** = 4  
 Initial search failed to find a single improvement (the solution could be optimal).
- 8: **alg\_stats[6]** – double *Output*  
*On exit:* an array of metrics collected during the initial search. These could be used as a basis for future optimization. If **fail.code**  $\neq$  0 on exit, the elements of **alg\_stats** are set to zero; the first five elements are also set to zero in the trivial cases **nc** = 1, 2 or 3.
- alg\_stats**[0]  
 Mean delta.
- alg\_stats**[1]  
 Standard deviation of deltas.
- alg\_stats**[2]  
 Cost at end of initial search phase.
- alg\_stats**[3]  
 Best cost encountered during search phase.
- alg\_stats**[4]  
 Initial system temperature. At the end of stage 1 of the algorithm, this is a function of the mean and variance of the deltas, and of the distance from best cost to the lower bound. It is a measure of the initial acceptance criteria for stage 2. The larger this value, the more iterations it will take to geometrically reduce it during stage 2 until the system is cooled (below a threshold value).
- alg\_stats**[5]  
 The lower bound used, which will be that computed internally when **bound**  $\leq$  0 on input. Subsequent calls with different random states can set **bound** to the value returned in **alg\_stats**[5] to avoid recomputation of the minimal spanning tree.

9: **state**[*dim*] – Integer

*Communication Array*

**Note:** the dimension, *dim*, of this array is dictated by the requirements of associated functions that must have been previously called. This array **MUST** be the same array passed as argument **state** in the previous call to `nag_rand_init_repeatable` (g05kfc) or `nag_rand_init_nonrepeatable` (g05kgc).

*On entry:* a valid RNG state initialized by `nag_rand_init_repeatable` (g05kfc) or `nag_rand_init_nonrepeatable` (g05kgc). Since the algorithm used is stochastic, a random number generator is employed; if the generator is initialized to a non-repeatable sequence (`nag_rand_init_nonrepeatable` (g05kgc)) then different solution paths will be taken on successive runs, returning possibly different final approximate solutions.

*On exit:* contains updated information on the state of the generator.

10: **fail** – NagError \*

*Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

## 6 Error Indicators and Warnings

### NE\_ALLOC\_FAIL

Dynamic memory allocation failed.

See Section 3.2.1.2 in the Essential Introduction for further information.

### NE\_BAD\_PARAM

On entry, argument *<value>* had an illegal value.

### NE\_INT

On entry, **nc** = *<value>*.

Constraint: **nc**  $\geq$  1.

### NE\_INTERNAL\_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.

See Section 3.6.6 in the Essential Introduction for further information.

### NE\_INVALID\_STATE

On entry, **state** vector has been corrupted or not initialized.

### NE\_NO\_LICENCE

Your licence key may have expired or may not have been installed correctly.

See Section 3.6.5 in the Essential Introduction for further information.

### NE\_REAL\_ARRAY

On entry, the strictly upper triangle of **dm** had a negative element.

## 7 Accuracy

The function will not perform well when the average change in cost caused by switching two cities is small relative to the cost; this can happen when many of the values in the distance matrix are relatively close to each other.

The quality of results from this function can vary quite markedly when different initial random states are used. It is therefore advisable to compute a number of approximations using different initial random states. The best cost and path can then be taken from the set of approximations obtained. If no change in

results is obtained after 10 such trials then it is unlikely that any further improvement can be made by this function.

## 8 Parallelism and Performance

Running many instances of the function in parallel with independent random number generator states can yield a set of possible solutions from which a best approximate solution may be chosen.

## 9 Further Comments

Memory is internally allocated for  $3 \times \mathbf{nc} - 2$  integers and  $\mathbf{nc} - 1$  real values.

In the case of two cities that are not connected, a suitably large number should be used as the distance (cost) between them so as to deter solution paths which directly connect the two cities.

If a city is to be visited more than once (or more than twice for the home city) then the distance matrix should contain multiple entries for that city (on rows and columns  $i_1, i_2, \dots$ ) with zero entries for distances to itself and identical distances to other cities.

## 10 Example

An approximation to the best path through 21 cities in the United Kingdom and Ireland, beginning and ending in Oxford, is sought. A lower bound is calculated internally.

### 10.1 Program Text

```

/* nag_mip_tsp_simann (h03bbc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 25, 2014.
 */

#include <nag.h>
#include <stdio.h>
#include <string.h>
#include <nag_stdlib.h>
#include <nagg05.h>
#include <nagh03.h>

int main(void)
{
    /* Scalars */
    Integer    exit_status = 0;
    Integer    subid = 53, lseed = 4, lstate;
    Integer    i, j, l, nc, n_i, icol, col_s, col_f, nrows, tmode;
    double     bound, targc, cost;
    /* Arrays */
    Integer    seed[] = { 304950, 889934, 209094, 23423990 };
    double     alg_stats[6];
    Integer    *state = 0, *path = 0;
    double     *dm = 0;
    char       **cities = 0;
    /* Nag Types */
    Nag_BaseRNG genid = Nag_WichmannHill_I;
    NagError   fail;

    INIT_FAIL(fail);

    printf("nag_mip_tsp_simann (h03bbc) Example Program Results\n\n");

    /* Read number of cities from data file */
#ifdef _WIN32
    scanf_s("%*[\n]"); /* Skip heading */
#else
    scanf("%*[\n]"); /* Skip heading */
#endif

```

```

#endif
#ifdef _WIN32
    scanf_s("%"NAG_IFMT" %*[\n]", &nc);
#else
    scanf("%"NAG_IFMT" %*[\n]", &nc);
#endif
/* Get the length of the state array for random number generation */
lstate = -1;
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
if (fail.code != NE_NOERROR) {
    printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
        fail.message);
    exit_status = 1;
    goto END;
}

/* Allocate arrays using nc and lstate*/
if (!(state = NAG_ALLOC(lstate, Integer)) ||
    !(path = NAG_ALLOC(nc, Integer)) ||
    !(dm = NAG_ALLOC(nc*nc, double)) ||
    !(cities = NAG_ALLOC(nc, char *)))
{
    printf("Allocation failure\n");
    exit_status = 2;
    goto END;
}

/* Read distance matrix 10 columns at a time */
/* Define DM for reading distance matrix from file */
#define DM(I, J) dm[(J-1)*nc + I - 1]
for (icol = 2; icol <= nc; icol=icol+10) {
    /* Skip a line */
#ifdef _WIN32
    scanf_s("%"NAG_IFMT" %*[\n]", &n_i);
#else
    scanf("%"NAG_IFMT" %*[\n]", &n_i);
#endif
    col_f = MIN(icol+9,nc);
    nrows = col_f - 1;
    for (i = 1; i <= nrows; i++) {
        /* Skip row number */
#ifdef _WIN32
        scanf_s("%"NAG_IFMT",&n_i);
#else
        scanf("%"NAG_IFMT",&n_i);
#endif
        col_s = MAX(i+1,icol);
        for (j = col_s; j <= col_f; j++) {
#ifdef _WIN32
            scanf_s("%lf",&DM(i,j));
#else
            scanf("%lf",&DM(i,j));
#endif
        }
    }
#ifdef _WIN32
    scanf_s("%*[\n] ");
#else
    scanf("%*[\n] ");
#endif
}

/* Read city names */
for (i = 0; i < nc; i++) {
    if (!(cities[i] = NAG_ALLOC(20, char)))
    {
        printf("Allocation failure\n");
        exit_status = 3;
        goto END;
    }
}
#ifdef _WIN32

```

```

scanf_s("%NAG_IFMT" %19s%*[^\\n] ", &n_i, cities[i], 20);
#else
scanf("%NAG_IFMT" %19s%*[^\\n] ", &n_i, cities[i]);
#endif
}

/* Initialise the random number generator to a repeatable sequence */
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
if (fail.code != NE_NOERROR) {
printf("Error from nag_rand_init_repeatable (g05kfc).\\n%s\\n",
fail.message);
exit_status = 4;
goto END;
}

/* Calculate a lower bound internally and try to find lowest cost path. */
bound = -1.0;
targc = -1.0;

/* Find low cost return path through all cities. */
nag_mip_tsp_simann(nc, dm, bound, targc, path, &cost, &tmode, alg_stats, state, &fail);
if (fail.code != NE_NOERROR) {
printf("Error from nag_mip_tsp_simann (h03bbc).\\n%s\\n",
fail.message);
exit_status = 5;
goto END;
}

printf("Initial search end cost: %12.2f\\n", alg_stats[2]);
printf("Search best cost : %12.2f\\n", alg_stats[3]);
printf("Initial temperature : %12.2f\\n", alg_stats[4]);
printf("Lower bound : %12.2f\\n", alg_stats[5]);
printf("Termination mode : %12"NAG_IFMT"\\n\\n", tmode);
printf("Final cost : %12.2f\\n\\n", cost);
printf("Final path:\\n");
printf(" %s --> %s\\n", cities[path[0]-1], cities[path[1]-1]);
l = strlen(cities[path[0]-1]);
for (i = 2; i <= nc-1; i++) {
printf(" ");
for (j = 0; j < l; j++) printf(" ");
printf(" --> %s\\n", cities[path[i]-1]);
}
printf(" ");
for (j = 0; j < l; j++) printf(" ");
printf(" --> %s\\n", cities[path[0]-1]);

END:
NAG_FREE(dm);
NAG_FREE(state);
NAG_FREE(path);
for (i = 0; i < nc; i++) {
NAG_FREE(cities[i]);
}
NAG_FREE(cities);

return exit_status;
}

```

## 10.2 Program Data

nag\_mip\_tsp\_simann (h03bbc) Example Program Data

21												: number of cities
	2	3	4	5	6	7	8	9	10	11		
1	23961	7112	21331	9050	22548	20667	13227	11617	14292	9455		
2		25998	4724	27936	2014	3997	20826	30488	21891	28327		
3			23108	2871	24325	22444	15004	8664	16359	6503		
4				25203	3444	3379	18093	27755	19158	25593		
5					26434	24553	15169	10773	16033	8612		

6						2668	19496	29159	20562	26997
7							17550	27212	18615	25051
8								19516	1895	17354
9									20649	3135
10										18537
	12	13	14	15	16	17	18	19	20	21
1	19634	6394	29483	14068	28136	11052	7228	13771	4752	24111
2	5403	25281	9312	31882	4751	18651	24909	25448	20113	25289
3	21411	1263	31260	7889	29913	12829	12517	8941	7038	26178
4	3598	22547	10592	29149	8868	15918	21956	22715	17380	23484
5	23519	3372	33368	5988	32022	13917	14626	6916	9147	25852
6	4074	23951	7766	30553	6075	17322	23580	24119	18784	23960
7	2127	22005	9586	28606	8239	15375	21634	22172	16837	22013
8	16200	14308	26049	15136	24703	2447	14727	8446	9140	11714
9	25990	7981	35839	15655	34493	17409	17103	15937	11618	30467
10	17383	15491	7232	16033	25886	3630	15910	9343	10323	9866
11	23819	5810	33668	13484	32321	15237	14931	13766	9446	28296
12		21026	10985	27628	9638	14397	20655	21193	15858	20188
13			30598	8276	29252	12168	11856	9064	6377	25227
14				37538	9425	24307	30565	31103	25769	30945
15					35803	14744	19628	6869	14149	26227
16						22962	29220	29758	24423	29599
17							12712	8242	7126	13457
18								15366	6300	25639
19									9465	18936
20										20048

: dm

- 1 Oxford
- 2 Dundee
- 3 Cardiff
- 4 Edinburgh
- 5 Swansea
- 6 Perth
- 7 Stirling
- 8 Bangor
- 9 Plymouth
- 10 Holyhead
- 11 Exeter
- 12 Glasgow
- 13 Newport
- 14 Inverness
- 15 St.Davids
- 16 Aberdeen
- 17 St.Asaph
- 18 Cambridge
- 19 Aberystwyth
- 20 Birmingham
- 21 Dublin

: names of cities

**10.3 Program Results**

nag\_mip\_tsp\_simann (h03bbc) Example Program Results

```
Initial search end cost:    432459.00
Search best cost          :    237068.00
Initial temperature      :    598481.00
Lower bound               :    106350.00
Termination mode         :                3
```

```
Final cost                :    131580.00
```

```
Final path:
Oxford --> Cambridge
        --> Birmingham
        --> Glasgow
        --> Stirling
        --> Edinburgh
        --> Perth
        --> Dundee
```



- > Aberdeen
  - > Inverness
  - > Holyhead
  - > Dublin
  - > Bangor
  - > St.Asaph
  - > Aberystwyth
  - > St.Davids
  - > Swansea
  - > Cardiff
  - > Newport
  - > Exeter
  - > Plymouth
  - > Oxford
-