Random Search in the Traveling Salesman Problem

László Hárs*

December 24, 1989

Abstract. A class of random search procedures for approximately solving the traveling salesman problem is investigated and a new algorithm is proposed. Experiments show that it generally gives very good tours in reasonable running time.

We also propose a set of random graphs for testing traveling salesman algorithms, which graphs easy to identically generate on any computer capable performing only 32-bit integer arithmetic.

Keywords. Traveling salesman problem, Random search, Local search.

1. Introduction

In one iteration of the so called k-opt exchange heuristics for the traveling salesman problem we remove k edges of an already constructed tour, i.e., of a Hamiltonian circuit of a given graph (of n nodes) with associated length to the edges, and try to reconnect the remaining k paths such that they form a shorter tour. If we fail to reduce the tour length we try another set of k edges of the tour, until a certain number of tries are made. (Usually all the k-subsets are tried.) The next iterations repeat these until no improvement is found. (See [3],[4].)

We are free to choose the order of the tried edge sets and to choose the number of trials. If all the k-subsets of the edges are tried, their order can be some kind of lexicographic order, since it is easy to generate. However, if we perform less trials, a random order seems to be more fair.

When an improving edge set is found, the corresponding rearrangement of the tour is quite costly. Practically, we have to copy all the nodes to another array or follow (and possibly alter) pointers all the way along the tour. The commonly used data structures to store the tour (see e.g., [2]) all require $O(n)$ operations in the average to reconstruct the tour. (In [7] a data structure is discussed which allow us to do it with $O(\log n)$ operations, but, as we see from the experiments, much more trials fail than succeed, therefore the increase of the number of operations to check if an improvement exists or not overweights the gain in the rearrangement in this type of exchange algorithms. Other algorithms, such as the one in [1] benefit from that data

* University Bonn, Institute for Operations Research and
Eötvös Loránd University, Budapest, Department of Computer Science.

structure.) Consequently, only a little time could be paid to reduce the number of tour rearrangements. Our proposal here is to check a sample collection of edge-set-trials and rearrange the tour only according to the best one (which yields the largest reduction of the tour length). This decreases the total number of tour rearrangements performed by the algorithm and adds no time to the improvement check. The tour length of the solution is not increased, in the average, either.

2. The (k,m)-opt heuristic

Let us choose k edges $C = \{e_1, \ldots, e_k\}$ from an already constructed tour, and delete m edges of C from the tour, all the possible ways. Associate to each of the resulting $p_1, \ldots, p_m$ paths a + or – sign. The path p_1 containing node n_1 always gets a +. Choose all the possible π permutations of $\{2, \ldots, m\}$, and connect the paths in the corresponding order: $p_1, p_{\pi(2)}, \ldots, p_{\pi(m)}$ to form a new tour. If the path p_i has the associated sign + it is used in its original direction, otherwise in reverse direction. (There are $\binom{k}{m} \cdot 2^{m-1} \cdot (m-1)!$ possibilities.)

We calculate all the resulting tour lengths and permanently rearrange the tour according to the best choice. Then repeat the whole process until all the possible choices for C are exhausted without improving the tour.

2.1. Computational experiences

The following diagrams show in a set of randomly generated test problems how the quality of the solution improves with the the number of "tries", i.e., with the number of checking a particular rearrangement of the tour segments. (This is an objective measure of the performance of the algorithm. The running time is proportional to that, but computer dependent.)

The y axis corresponds to the average tour lengths of 100 or 25 random graphs of 36 or 100 nodes, respectively. The x axis represents the number of improvement checks made. The different figures show different parameter settings of the algorithm.

On the figures, where two diagrams are drawn (for comparison) the one wich has steeper initial decrease corresponds to the (3,2)-opt algorithm. Only those diagrams are plotted here, which show the general behavior of the algorithms.

We want to stop the algorithm early and restart with different search order. Our experiments indicate, that the (3,2)-opt is the optimum choice.

The following tables show the results of the (k, m)-opt algorithms, if we continue them until termination. There are some interesting things to be observed.

- The increase of m from 2 to 3 yields some 4% improvement in the tour length, while increasing it from 3 to 4 yields further about 1% improvement. (But $m \geq 4$ is impractical for larger problems.)
- When k increases the total number of tries the algorithm makes first decreases then increases, having an optimum, which depends on n.
- As k increases the number of tour rearrangements the algorithms perform decreases quite fast. (But the saving in computation time is not significant, since the large number of trials dominates the total running time.)

K	M	Length	Tries	Convs
2	2	152630	9098	78
3	2	149669	6777	74
4	2	149776	6841	68
5	2	150284	7060	61
6	2	149576	7183	58
7	2	150063	7414	53
8	2	148449	7864	51
9	2	149311	8072	47
10	2	150010	8629	46
11	2	149833	9392	44
12	2	148780	9986	43

Nodes = 36, Average of 100 runs

K	M	Length	Tries	Convs
3	3	144856	137889	68
4	3	144088	136602	59
5	3	144368	131206	52
6	3	143287	139211	49
7	3	143286	154470	44
8	3	142842	156500	41
9	3	143588	168403	39
10	3	144018	174614	36
11	3	143202	176088	34
12	3	143651	192544	32

Nodes = 36, Average of 100 runs

K	M	Length	Tries	Convs
4	4	142581	4879459	56
5	4	142493	4870133	51
6	4	142563	4923389	44
7	4	142500	5128368	40
8	4	142904	5382989	36
9	4	142116	5403586	33
10	4	142162	5358024	30
11	4	142119	5417280	28
12	4	142092	5831654	26

Nodes = 36, Average of 100 runs

K	M	Length	Tries	Convs
2	2	245963	102874	319
3	2	244282	69541	295
4	2	244763	67001	272
5	2	241328	70486	256
6	2	244072	69623	245
7	2	246153	69283	226
8	2	242209	68833	214
9	2	242232	79416	209
10	2	241440	80406	200
11	2	244935	78025	192
12	2	245205	75853	180
13	2	243294	81126	176
14	2	244413	83189	173
15	2	243302	84185	165
16	2	244282	90624	160
17	2	244192	88716	155
18	2	242171	97357	156
19	2	242608	96937	149
20	2	242943	104546	148
21	2	243404	108142	143
22	2	243961	110825	141
23	2	244154	110247	137
24	2	241723	115788	136
25	2	243019	122976	137
26	2	241268	126022	132
27	2	240377	130656	131
28	2	242777	133328	126
29	2	243240	139859	126
30	2	240466	146612	127

Nodes = 100, Average of 25 runs

3. Restarts

Any of the (k, m)-opt algorithm gives, of course, a local minimum. This may be far from the global one. Therefore, if we restart the procedure with different random number seed, a new local optimum can be found. By restarting many times and saving the best tour we can improve the solution.

K	M	Length	Tries	Convs
3	3	234237	3978692	280
4	3	233174	4235601	244
5	3	231363	3857418	223
6	3	233926	3592832	208
7	3	233564	3365174	197
8	3	231856	4050601	186
9	3	233034	4142934	174
10	3	232036	3999168	161
11	3	231022	4474325	160
12	3	231079	4003085	148
13	3	233210	3997502	144
14	3	231644	4533052	138
15	3	231898	4462494	131
16	3	232008	4149376	127
17	3	230303	5134054	126
18	3	231557	4797819	121
19	3	232854	4929652	117
20	3	230896	4897075	115
21	3	233254	5165933	111
22	3	231356	5453325	111
23	3	232617	5252928	107
24	3	229865	5408128	105
25	3	231502	5715776	103
26	3	230587	6155968	102
27	3	231241	6379776	100
28	3	231246	6050903	96
29	3	231095	6551476	96
30	3	232826	7003987	94

Nodes = 100, Average of 25 runs

Restarts	25	50	100	250	500	1000	2000	5000
n = 36	141370	140725	140446	140157	140065	140023	140020	140018
n = 100	233119	231891	230146	228585	227668	226336	225737	225189

Average tour lengths

Restarts	25	50	100	250	500	1000	2000
n = 36	1	0.5	0.3	0.1	0.03	0.004	0.001
n = 100	3.5	3	2.2	1.5	1.1	0.5	0.2

Percentage of tour lengths above best known lengths

Another interesting possibility is the use of simulated annealing technique (refer to, e.g., [1]). The experiments with that will be discussed in a later paper.

4. Hybrid algorithm

If n is large ($\gg 100$) the (k, m)-opt algorithm with $m > 3$ takes too long time to complete. In this case, if $k > 3$ we get usually no better solution than with $k = 3$.

We need not complete each restart of the algorithm. Our experiments show, that after $2n^2 \dots 3n^2$ iterations of the $(3, 2)$-opt algorithm of different seeds, almost always that restart gives the best tour length which is the best at termination. We continue until all the 3-sets are tried only the best restart (which is best after the fixed number of iterations).

These give our proposed algorithm. Still we have one parameter open, the number of restarts. This can be supplied by the user. The running time is proportional to this number. Some improvement is reached with larger values.

The following part of a Pascal program does the job. The declarations:

```
CONST
      MaxN   = 500;
      Step   = 3;                           {Checks per iteration      }
TYPE
      NodeRange    = 0..MaxN;
      NodeArray    = Array [1..MaxN] of NodeRange;
      NodePointer  = ->NodeArray;
VAR
      BestNode,
      Node,
      NewNode      : NodePointer;
      Dist         : Array[1..MaxN,1..MaxN] of Integer;     {Distances}
      N            : NodeRange;             {Number of nodes           }
      Prob,                                 {Seed for random graph     }
      Starts,                               {Restarts' number          }
      Start,                                {Restart's index           }
      Samples,                              {Trials per start          }
      Tries,                                {Number of tries made      }
      Fails,                                {Number of failors improve }
      Con,                                  {Best conversion's type    }
      RndNmb,                               {Last random number        }
      Limit,                                {Failors limit to stop     }
      Improve,                              {Tour improvement          }
      MinCost,                              {Best improvement          }
      CCost,                                {Current cost of tour      }
      BestCost,                             {Cost of best tour         }
      u1,v1, u2,v2, u3,v3:   Integer;       {Random edges              }
```

We make use of procedure `Swap`, which exchanges the value of its two arguments. Two more procedures are used:

```
Function TotCost: Integer;                                  {Tour length}
Var i, t : Integer;
Begin
  t := Dist[Node->[N],Node->[1]];
  For i:= 1 to N-1 do
    t := t + Dist[Node->[i],Node->[i+1]];
  TotCost := t;
end;
```

```
Procedure Reverse( u,v : NodeRange);          {Reverses tour-arc (u,v)}
Var  i, k : Integer;
     Nd : NodePointer;
Begin
  k := u + v;
  For i := 1 to u-1 do NewNode->[i] := Node->[i];
  For i := u to v do NewNode->[k-i] := Node->[i];
  For i := v+1 to N do NewNode->[i] := Node->[i];
  Swap( NewNode, Node);
end;
```

The actual work is done in the following procedure.

```
Procedure DoImprove;
Begin
  Tries := Tries + Step;
  u1 := Rand(N); v1 := 1 + u1 mod N;                        {Random edges}
  Repeat u2 := Rand(N); v2 := 1 + u2 mod N;
  Until (u2<>u1) & (u2<>v1) & (v2<>u1);
  Repeat u3 := Rand(N); v3 := 1 + u3 mod N;
  Until (u3<>u1) & (u3<>v1) & (v3<>u1)
      & (u3<>u2) & (u3<>v2) & (v3<>u2);
  If u1 > u2 Then Swap(u1,u2);                                    {Sort}
  If u2 > u3 Then Swap(u2,u3);
  If u1 > u2 Then Swap(u1,u2);
  v1 := 1 + u1;  v2 := 1 + u2;  v3 := 1 + u3 mod N;
  MinCost := 0;
  Improve := Dist[Node->[u1],Node->[u2]] + Dist[Node->[v1],Node->[v2]]
           - Dist[Node->[u1],Node->[v1]] - Dist[Node->[u2],Node->[v2]];
  If Improve < MinCost then Begin
    Con := 1; MinCost := Improve end;
  Improve := Dist[Node->[u1],Node->[u3]] + Dist[Node->[v1],Node->[v3]]
           - Dist[Node->[u1],Node->[v1]] - Dist[Node->[u3],Node->[v3]];
  If Improve < MinCost then Begin
    Con := 2; MinCost := Improve end;
  Improve := Dist[Node->[u2],Node->[u3]] + Dist[Node->[v2],Node->[v3]]
           - Dist[Node->[u2],Node->[v2]] - Dist[Node->[u3],Node->[v3]];
  If Improve < MinCost then Begin
    Con := 3; MinCost := Improve end;
  If MinCost < 0 then Begin
    Case Con of
      1: Reverse(v1,u2);
      2: Reverse(v1,u3);
      3: Reverse(v2,u3);
    end; {Case}
    CCost := CCost + MinCost;
  End;
End;
```

We compose the proposed algorithm now in the following way:

```
BEGIN
  New(Node); New(NewNode); New(BestNode);
  Repeat
    WriteLn; WriteLn('N, Prob, Starts, Samples:');
    ReadLn( N, Prob, Starts, Samples);
    If N < 7 Then LEAVE;
```

```
    Limit := (N * (N-1) * 5) div 8;              {>N(N-1)/2 to be sure}
    BestCost := MaxInt;
    For Start := 1 to Starts do Begin
      Tries := 0;
      If Prob < 0 Then RandGrid(-Prob)            {Random test problems}
                  Else RandGraph(Prob);
      CCost := TotCost;
      If Start = 1 Then Write('Starting Length = ', CCost:1);
      RndNmb := Start;
      Repeat DoImprove;
      Until Tries >= Samples;
      If BestCost > CCost Then Begin
        Swap( BestNode, Node);                                {Save best}
        BestCost := CCost end;
    end; {Start}
    Swap( BestNode, Node);                                    {Load best}
    CCost := BestCost;
    RndNmb:= 0;
    Tries := 0;
    Fails := 0;
    Repeat
      DoImprove;
      If MinCost < 0 then  Fails := Tries;
    Until Tries-Fails >= Limit;
    Write('   Reduced Length = ', CCost:1);
  Until FALSE;
END.
```

4.1. Running time

With $3n^2$ samples and l restarts the algorithm does approximatelly $3n^2(l+2)$ trials. To get the best solution one needs generally so many restarts, which is proportional to an exponential function of n. That is impractical with large n values. For acceptable tour length we can fix l to $25 \ldots 100$, which gives an $O(n^2)$ running time algorithm.

4.2. Combination with other algorithms

We can start with another traveling salesman heuristic, such as nearest neighbor, fartest insertion, and so on (see [3]). This can be followed by our algorithm or just the second step of it. There are so many possibilities, we can not even list them here. Our algorithm has the advantage to be very simple, since it is not a combination of different heuristics. Also, we know the running time in advance.

5. Random test problems

If we want to compare algorithms developed by different people on different computers, we need to run the programs on the same set of test problems. If these test problems are randomly generated, the first thing we need is a pseudo random number generator which gives the same sequence of numbers on any computer. We only assume, that the integer operations are performed in 32-bit precision, or this precision can be simulated on that particular computer.

5.1. Random number generator

The random number generator must use only integer arithmetic, since the floating point operations may give different results depending on the precision of the floating point arithmetic unit, the rounding rules applied, etc. We have chosen the simple congruency method from [8]: $X_{n+1} = (1 + 1664525 \times X_n) \bmod 2^{32}$. It gives quite a good sequence according to many statistical tests. (The use of two's complement representation of the integers does not make a difference. See [8].)

We avoid the floating point operations, so it takes a few statements to transform this sequence to numbers of a given interval $[1, m]$. For simplicity, we confine ourselves to use only intervals with $m < 2^{15}$. The following Pascal procedure does the job. (Note that the division and multiplication with 2^{16} are used only to access the upper and lower half of the 32-bit machine word. In practice they can be replaced by machine dependent simple statements. Optimizing compilers, such as IBM's VS-PASCAL may do this automatically. We assume that the overflow of the integer multiplication is ignored at the first statement of the procedure and the least significant digits are given as the result. With some compilers one may need turn off overflow checking or simulate the 32-bit multiplication with multiplying the upper and lower half words and combine the results.)

```
Function Rand( Mx: Integer): Integer;                          {Mx < 32768}
Const    TwoTo16 = 65536;
Var      Half1, Half2: Integer;
Begin
  RndNmb := 1 + 1664525 * RndNmb;                {Global 32 bit integer}
  Half1 := RndNmb div TwoTo16;
  If Half1 < 0 Then Half1 := TwoTo16 + Half1;
  Half2 := RndNmb mod TwoTo16;
  Rand := 1 + ( Half1 * Mx + (Half2 * Mx) div TwoTo16 ) div TwoTo16;
End;
```

If it is used on computers with larger precision, the first executable statement should be replaced by

```
  RndNmb := (1 + 1664525 * RndNmb) mod TwoTo32;
```

with constant `TwoTo32` $= 2^{32}$.

5.2. Random graphs

We use the simplest way to generate random graphs: take n uniformly distributed random points in a large square, connect all the point-pairs by an edge of length

equal to their approximate Euclidean distance. To calculate the Euclidean distance, unfortunately we need the square root function, which should be programed now using only integer arithmetic. We do not need very good approximation for our test purposes. The following procedure gives a result within about 6% relative error, if the points are not too close (at least in a distance of a hundred units). Since our points are in a big square, the probability of having larger error respective to the Euclidean distance is extremely low.

```
Function Len( x1,y1, x2,y2: Integer): Integer;
Var  dx, dy: Integer;
Begin
  dx := Abs(x1-x2);  dy := Abs(y1-y2);
  If dx > dy Then Len := dx + dy div 3
             Else Len := dy + dx div 3;
end;
```

We store the edge lengths in the two dimensional array `Dist`.

```
Procedure RandGraph(Seed: Integer);
Var i, j: Integer;
    x, y: Array[1..N] of Integer;
Begin
  RndNmb := Seed;                          {Initialize random generator}
  For i := 1 To N Do Begin
    Node->[i] := i;                                    {Starting tour}
    x[i] := Rand(MaxCoord);  y[i] := Rand(MaxCoord)  End;
  For i := 1 To N Do  For j := 1 To N Do
    Dist[i,j] := Len(x[i],y[i], x[j],y[j]);
End;
```

The following tables list the length of the starting tours and of the shortest known tours for the random number generator seed 1,2,...,100 in case of 36 nodes graphs, and seed 1,2,...,25 in case of 100 nodes graphs generated by the procedure above.

Another important graph to test the TSP programs on is the regular grid. It is a very difficult problem for the local exchange heuristic algorithms, because there are many tours with exactly the same length, and to improve the tour, sometimes a drastical rearrangement is necessary. Also, in this case the optimal tour length is known: if n (the number of the points) is even it is n times the grid-distance (the least distance between the points), otherwise one grid-distance should be replaced by the smallest diagonal distance of the points. (The proof is an easy excercise.)

If we choose the square size carefully, the grid-distance will be integer for several small n values. For example, `MaxCoord = 27720` gives integer values for $n = 2^2, \ldots, 13^2, 15^2, 16^2, 19^2, 21^2$ and several more. If the grid-distance were a fractional number, we round it down, but then the complete grid will be a little smaller.

In the case of the regular grid we choose the initial tour as a random permutation of the nodes generated from the starting value `Seed`.

```
Const MaxCoord = 27720;                    {= 2*2*2 * 3*3 * 5 * 7 * 11}
```

Seed	Start-Length	Best-Length
1	646819	138511
2	525654	150391
3	497485	139473
4	564015	124587
5	605713	153783
6	583327	147063
7	538228	139016
8	560962	132748
9	556556	142003
10	548372	140933
11	589118	148278
12	538465	156673
13	525099	132410
14	599474	135705
15	467008	149784
16	519083	146625
17	524266	141660
18	548257	138835
19	482755	138249
20	549409	131794
21	517466	149853
22	544349	151152
23	550774	138627
24	466645	145050
25	495487	133012
26	533676	143565
27	551159	133881
28	493462	136962
29	617446	152019
30	611712	144548
31	499703	132444
32	611113	151356
33	603611	139300
34	555134	144257
35	459087	135342
36	495536	138208
37	591282	154631
38	529123	147852
39	590193	121134
40	552821	128769
41	514743	137441
42	630803	144064
43	530612	142672
44	579806	137894
45	530046	123373
46	573924	139526
47	506605	132196
48	500239	135351
49	562319	148984
50	540587	142010

Seed	Start-Length	Best-Length
51	459612	125305
52	574358	142435
53	591088	150043
54	507748	136058
55	537463	150661
56	578526	140560
57	641276	141588
58	571845	128662
59	535851	144026
60	599286	147273
61	413602	133006
62	616645	143202
63	539702	124177
64	497800	137901
65	509451	141280
66	668629	148102
67	593702	148353
68	559086	142094
69	588735	136561
70	514797	131274
71	559455	139917
72	523061	120194
73	468939	127237
74	543102	131464
75	535144	139536
76	480170	145118
77	566054	150389
78	571079	142767
79	520766	147309
80	499800	142628
81	571822	141316
82	490074	133112
83	580066	143754
84	543757	139730
85	505304	146589
86	519586	134006
87	529076	145137
88	528446	138452
89	520592	138342
90	541622	137398
91	510546	137938
92	551627	145763
93	615092	129437
94	488734	133393
95	549505	130454
96	644764	141154
97	630512	149976
98	560264	136944
99	551505	144656
100	522560	141176

Nodes = 36, Mean Start/Best-Length = 546318/140018

```
Procedure RandGrid(Seed: Integer);
Var i, j, k, sq, y0, d, dx, dy : Integer;
    x, y: Array[1..N] of Integer;
Begin
```

Seed	Start-Length	Best-Length
1	1586606	219576
2	1593808	223671
3	1496771	230093
4	1460396	226366
5	1653560	233465
6	1584497	214936
7	1453726	213977
8	1537787	220427
9	1605332	231438
10	1488760	225185
11	1486442	224516
12	1563251	231339
13	1432072	233698
14	1610838	222881
15	1476469	222863
16	1489154	230586
17	1480513	230275
18	1496084	233132
19	1370268	227081
20	1467749	215781
21	1469668	238423
22	1501976	221782
23	1500483	214051
24	1456006	226735
25	1488017	217457

Nodes = 100, Mean Start/Best-Length = 1510009/225189

```
  RndNmb := Seed;                              {Initialize random generator}
  sq := 1; While sqr(sq+2) <= N Do sq := sq + 1;           {Square-root}
  d := MaxCoord div sq;                                   {Grid-distance}
  k := 0;
  For i := 0 To sq Do Begin
    k := k + 1;
    x[k] := 0;  y[k] := i * d;
    For j := 1 To sq Do Begin k := k + 1;
      x[k] := j * d;  y[k] := i * d  end;
  end; {i}
  For i := k+1 To N Do Begin                              {If N <> square}
    x[i] := x[k];  y[i] := y[k] end;
  For i := 1 To N Do  For j := 1 To N Do
    Dist[i,j] := Len(x[i],y[i], x[j],y[j]);
  For i := 1 To N Do  Node->[i] := i;
  For i := 1 To N-1 Do                                 {Random permutation}
    Swap( Node->[i], Node->[i-1+Rand(N+1-i)]);
End;
```

The following formule give the optimum tour lengths, with $d = \lfloor\sqrt{n}\rfloor$. If d is even

$$\left\lfloor \frac{\texttt{MaxCoord}}{d-1} \right\rfloor d^2$$

and if d is odd

$$\left\lfloor \left\lfloor \frac{\texttt{MaxCoord}}{d-1} \right\rfloor (d^2 + 1/3) \right\rfloor .$$

References

[1] Y. Rossier, M. Troyon, Th. M. Liebling "Probabilistic exchange algorithms and Euclidean traveling salesman problems," *OR Spectrum* **8**(1986), pp. 151–164.

[2] R. E. Tarjan "Data structures and network algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* **44**(1983).

[3] B. L. Golden and W. R. Stewart "Empirical analysis of heuristics," in The Travelling Salesman Problem *edited by* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan and D. B. Shmoys *John Wiley & Sons, Chichester*, **1986**, pp. 207–249.

[4] C. H. Papadimitriou, K. Steiglitz "Combinatorial Optimization: Algorithms and Complexity," *Prentice-Hall* **1982**.

[5] S. Lin and B. Kernighan "An effective heuristic algorithm for the traveling salesman problem," *Operations Research* **21**(1973), pp. 498–516.

[6] A. V. Aho, J. E. Hopcroft and J. D. Ulman "The design and analysis of computer algorithms," *Addison-Wesley, Reading, MA* **1974**.

[7] — "Reversible segment list," *(To appear)* **1989**.

[8] D. E. Knuth "The art of computer programing. Volume 2 (Seminumerical algorithms," *Addison-Wesley, Reading, MA* **1981**.