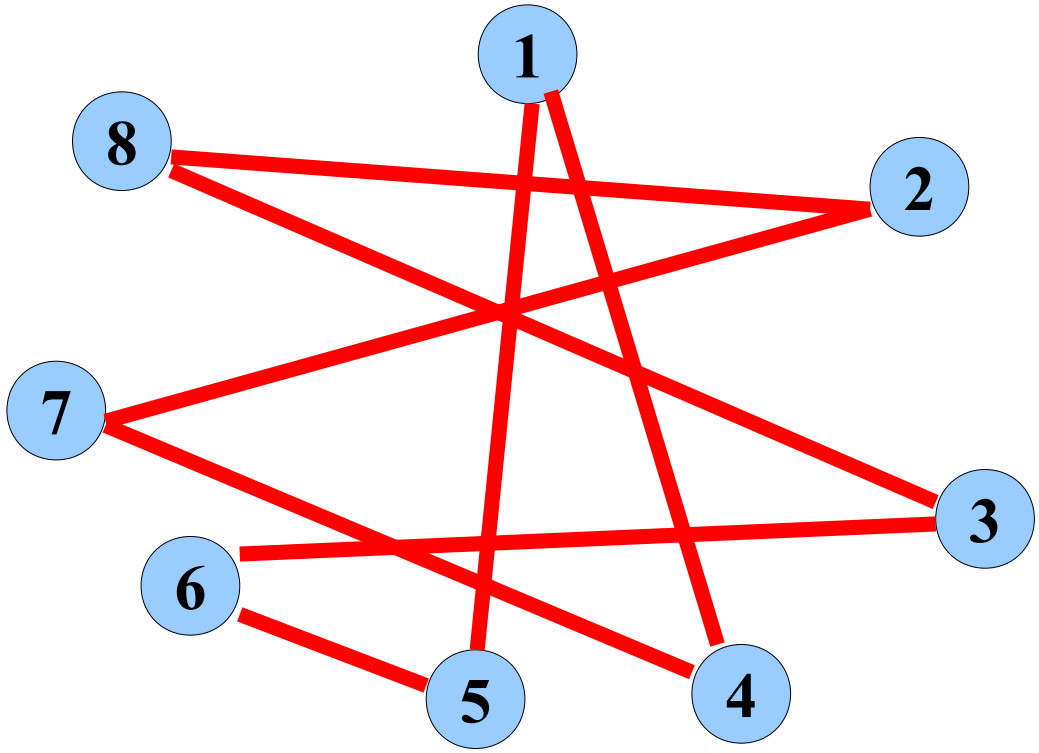
Tour and its cost

Consider a simple undirected complete graph K_n with a cost assigned to every edge. With the vertices of K_n labelled $1,2,\ldots,n$, the cost of the edge between nodes i and j, will be denoted by $c_{i,j}$. Since the graph is undirected, clearly $c_{i,j}=c_{j,i}, 1\le i, j\le n$. A **tour** in K_n is a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ in K_n with the **cost of such a tour** defined as

$C=\sum_{k=1}^{n-1} c_{i_k,i_{k+1}}+c_{i_n,i_1}$. In other words the cost of a tour is the sum of the costs of all its edges.



In every K_n, there are a total of $\frac{(n-1)!}{2}$ different tours. For the above graph, for example, there are 2520.

Travelling Salesman Problem

Given a complete graph K_n, the travelling salesman problem for this graph is to find a tour with a minimum cost.

Here are some applications of the theoretical problem, the first one giving the problem its name:

- a salesman is living in an area with n towns. Starting from his home town, a travelling salesman plans to visit all the n towns returning home. For some reason, every town must be visited exactly once;
- a predetermined pattern of n holes is to be bored in a printed circuit board while minimizing the distance travelled by the drilling machine.

Exhaustive search methods

One way of solving this problem is an **exhaustive search**. This means that an algorithm goes systematically through a list of all tours calculating their costs. A tour with the least cost is then chosen. While this method may seem quite simple and straightforward, it may only be used for small values of n. Suppose that we have a computer that can check 10^9 tours per second. Consider the following table:

Number of towns	Computing time
10	0.00018 **seconds**
15	44 **seconds**
20	1.9 **years**
25	9 837 145 **years**

Branch and bound method

The following method may overcome the difficulty encountered when solving a TSP by exhaustive search or “rude force” as it is sometimes called. For some particular problems it may deliver the exact minimum tour within a reasonable time, but on the whole, the number of nodes cannot be increased too much. Generally, the computing time may still grow exponentially.

A Hamiltonian circle H of a graph G may be defined as a subgraph of G satisfying the following two conditions:

1. H is connected and contains exactly one circle
2. The degree of each vertex of H is exactly two.

We will denote the set of all the Hamiltonian circles of G by Ham.

A subgraph S complying only with condition 1 above is actually a spanning tree of G with an extra edge. If $ST+1$ is the set of all such subgraphs, clearly, $Ham \subseteq ST+1$.

Denoting by H^* the Hamiltonian circle with the least cost and by S^* the graph in $ST+1$ with the least cost, we can observe the following:

- If H^* does not contain edge e and the cost of e is changed to infinity in G, H^* will still remain the Hamiltonian circle in G with the least cost.
- Finding the graph in $ST+1$ with the least cost is an easy task.
- $S^* \leq H^*$
- If $S^* \in Ham \Rightarrow S^* = H^*$

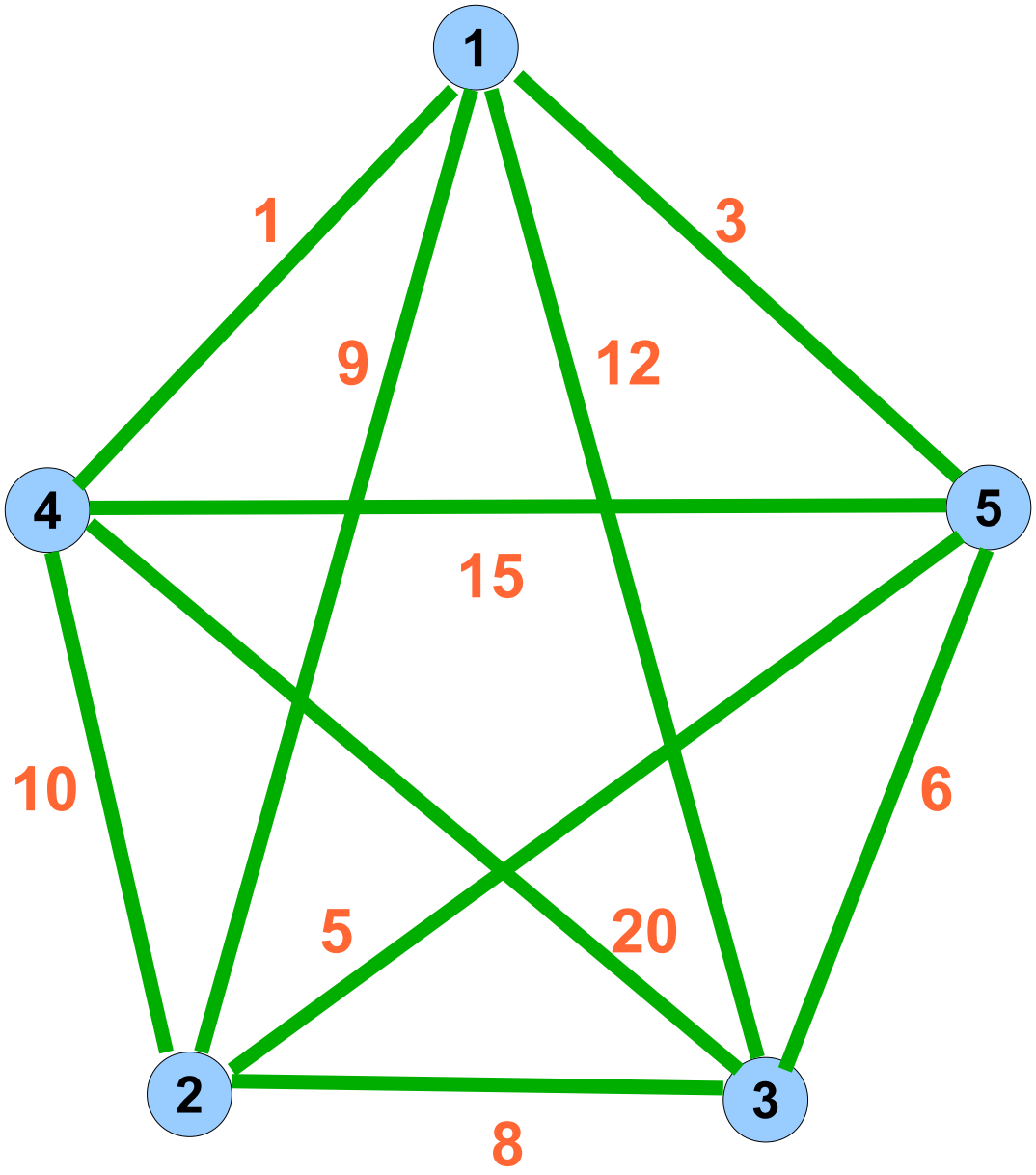
Using the above observations, we can devise the following algorithm:

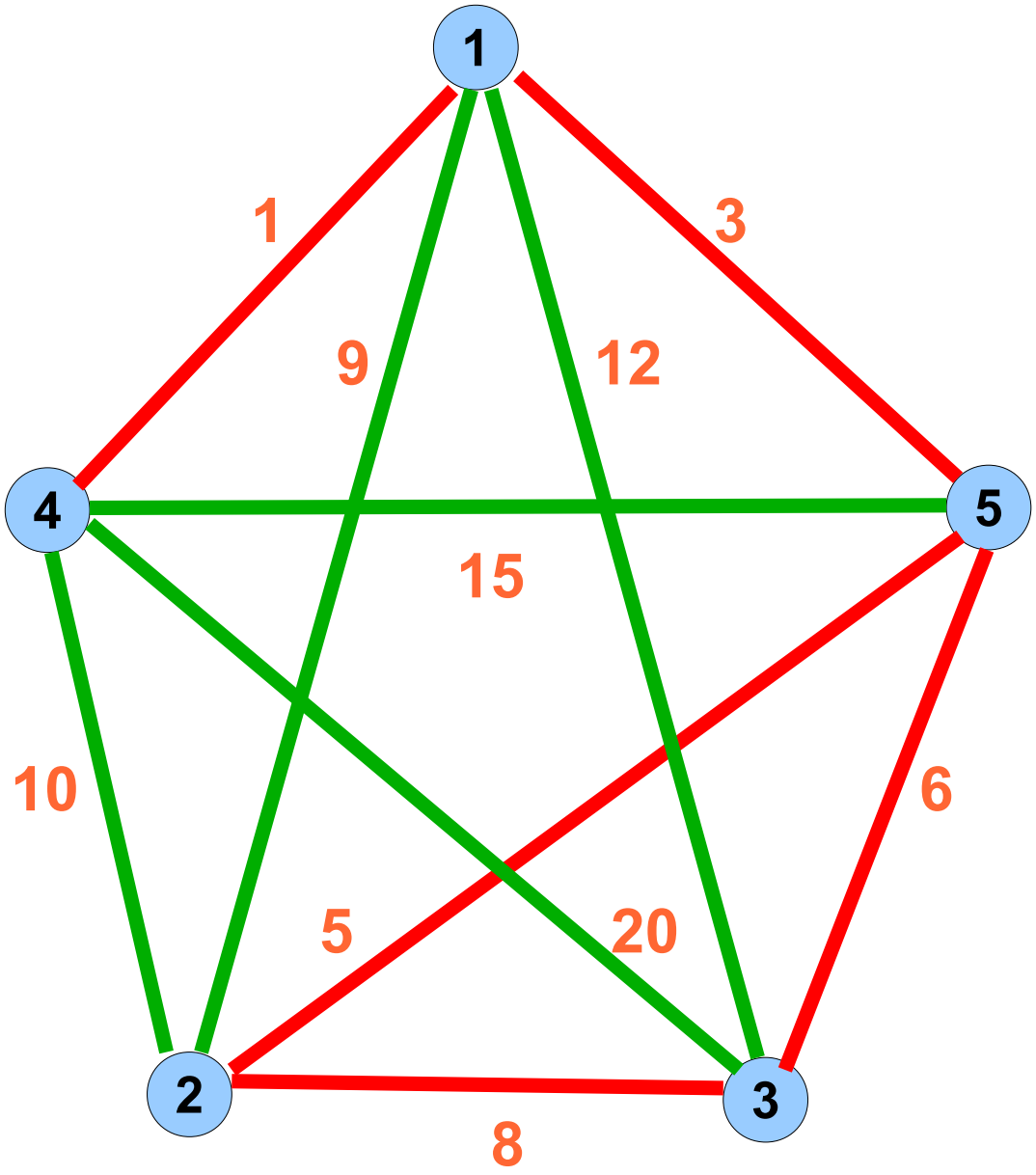
1. Push to *Stack* task T : find S^* in G. Initialise: $MinCost := \infty$
   $MinTour := \emptyset$

2. Remove the top task from *Stack,* and solve it. Store solution in *CurSol* and its price in *CurCost*.

3. (a) If $CurSol \in H$ and $CurCost < MinCost$, then $MinTour := CurSol$ and $MinCost := CurCost$.
   (b) otherwise if $CurCost < MinCost$, push to *Stack* new tasks T_1, T_2, ... T_k created by setting to ∞ the costs of all edges e_1, e_2, ... e_k respectively incident on all vertices of a degree greater than two of the task currently being solved.
   (c) otherwise do nothing

4. If *Stack* is not empty go to 2).
   Otherwise return *MinTour and MinCost* as the result.

STEP 0	STACK:	T
(1,4) = 1 (1,5) = 3 (2,5) = 5 (3,5) = 6 (2,3) = 8 (1,2) = 9 (2,4) = 10 (1,3) = 12 (4,5) = 15 (3,4) = 20 *CurCost=?* $MinTour = \emptyset$ $MinCost = \infty$	*T0 ()*	

STEP 1	**STACK:**	*T*
(1,4) = 1 **(1,5) = 3** **(2,5) = 5** **(3,5) = 6** **(2,3) = 8** **(1,2) = 9** **(2,4) = 10** **(1,3) = 12** **(4,5) = 15** **(3,4) = 20** *CurCost=23* $MinTour = \emptyset$ $MinCost = \infty$	***T1 ((2,5))*** ***T2 ((3,5))*** ***T3 ((1,5))***	

STEP 2

(1,4) = 1

(1,5) = 3

(2,5) = ∞

(3,5) = 6

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=27

MinTour = Ø

MinCost = ∞

STACK:

T2 ((3,5))

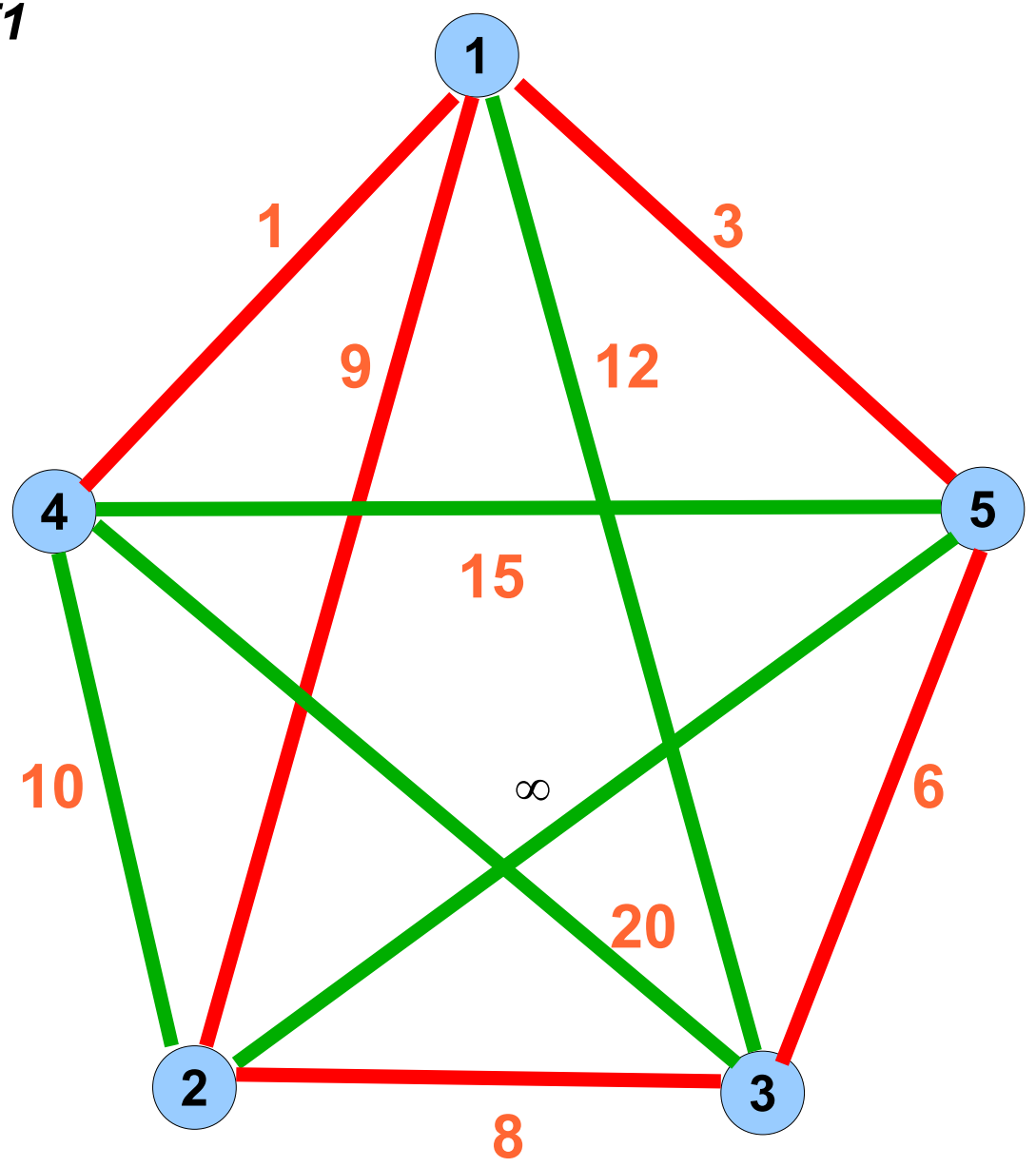
T3 ((1,5))

T4 ((2,5)(1,2))

T5 ((2,5)(1,4))

T6 ((2,5)(1,5))

T1



STEP 3

(1,4) = 1

(1,5) = 3

(2,5) = 5

(3,5) = ∞

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=26

MinTour = Ø

MinCost = ∞

STACK:

T3 ((1,5))

T4 ((2,5)(1,2))

T5 ((2,5)(1,4))

T6 ((2,5)(1,5))

T7 ((3,5)(1,4))

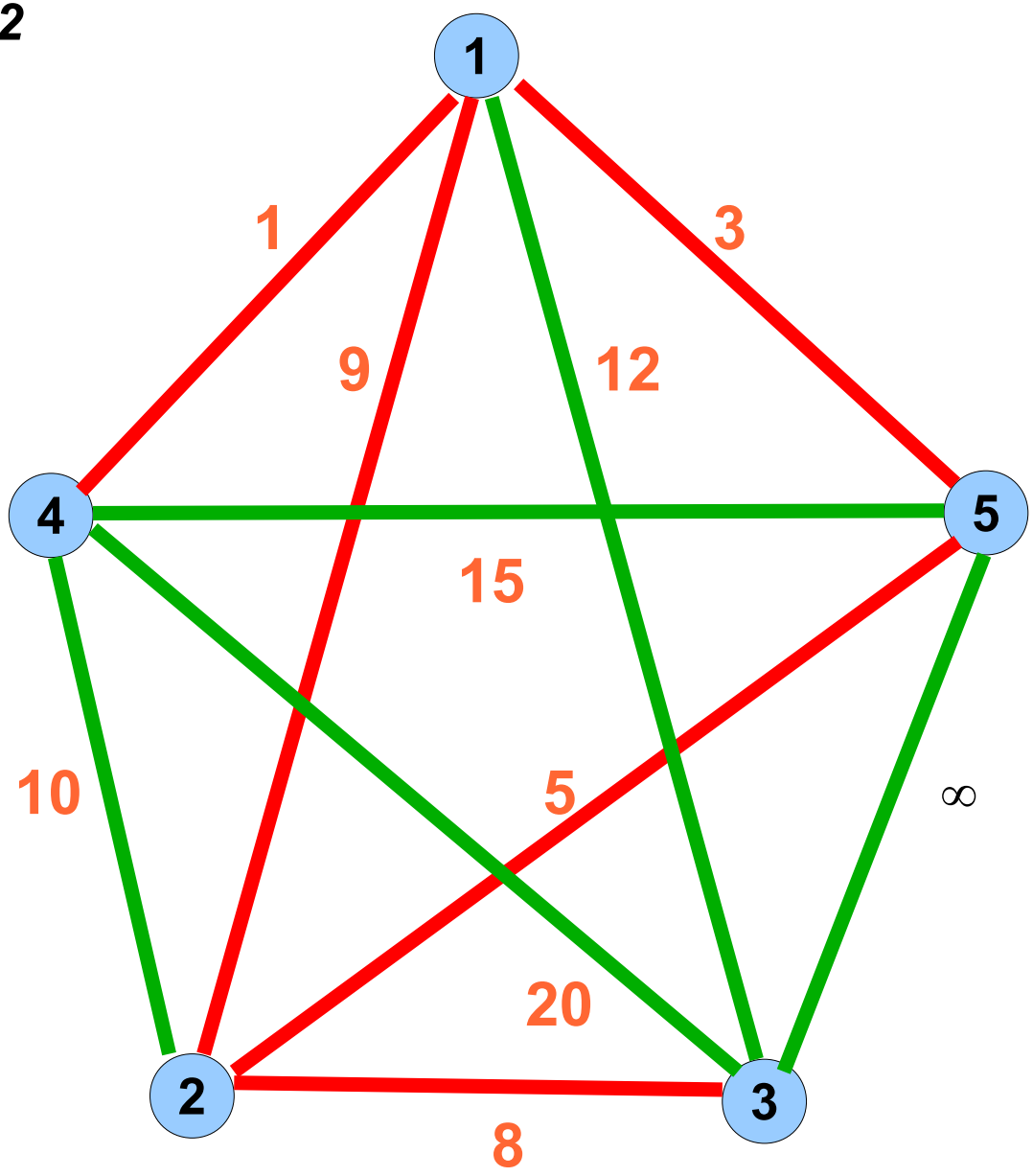
T8 ((3,5)(1,2))

T9 ((3,5)(1,5))

T10 ((3,5)(2,5))

T11 ((3,5)(2,3))

T2



STEP 4

(1,4) = 1

(1,5) = ∞

(2,5) = 5

(3,5) = 6

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=29

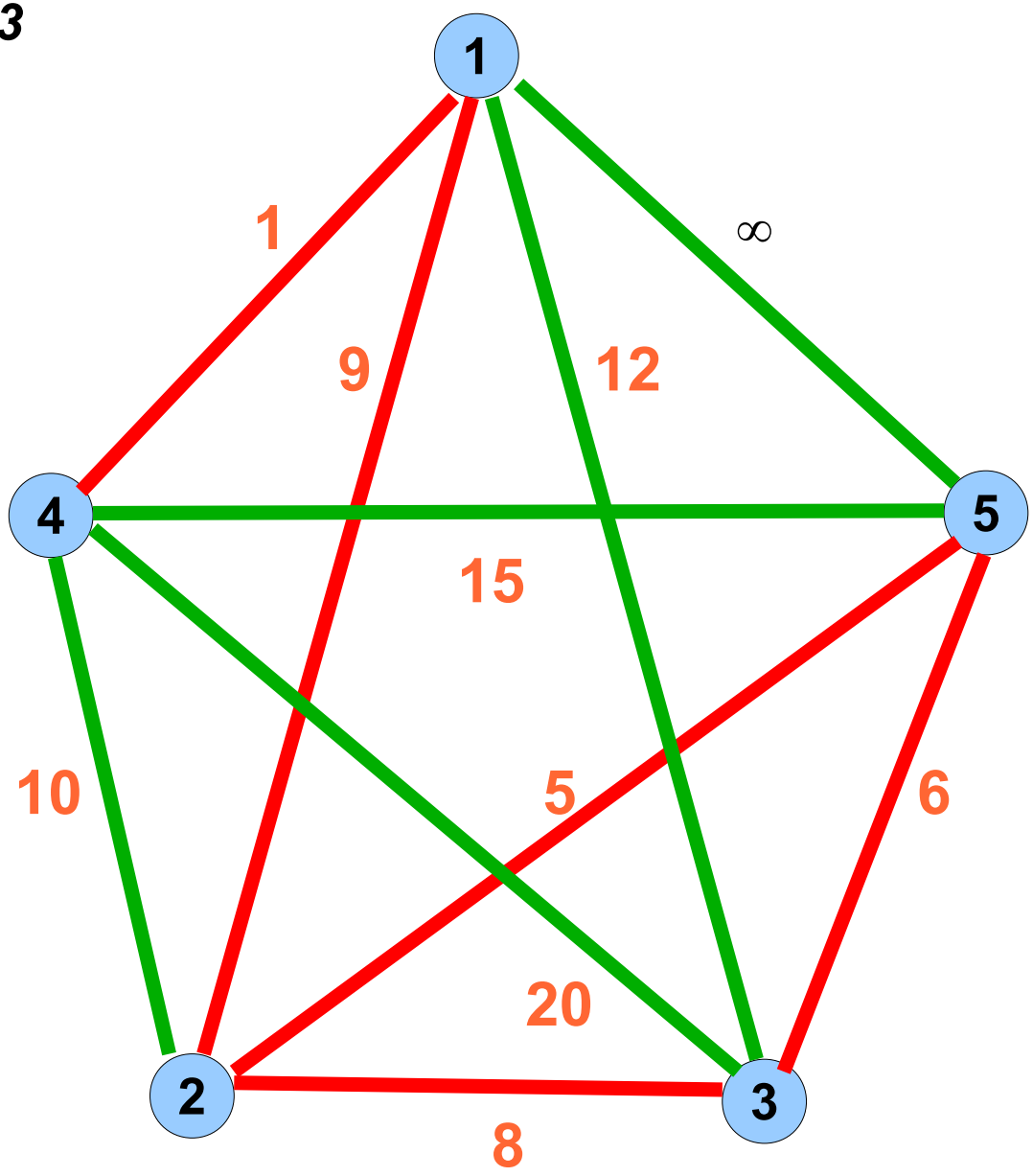
MinTour = Ø

MinCost = ∞

STACK:

T4 ((2,5)(1,2))
T5 ((2,5)(1,4))
T6 ((2,5)(1,5))
T7 ((3,5)(1,4))
T8 ((3,5)(1,2))
T9 ((3,5)(1,5))
T10 ((3,5)(2,5))
T11 ((3,5)(2,3))
T12 ((1,5)(1,2))
T13 ((1,5)(2,5))
T14 ((1,5)(2,3))

T3



STEP 5

(1,4) = 1

(1,5) =3

(2,5) = ∞

(3,5) = 6

(2,3) = 8

(1,2) = ∞

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=28

MinTour=

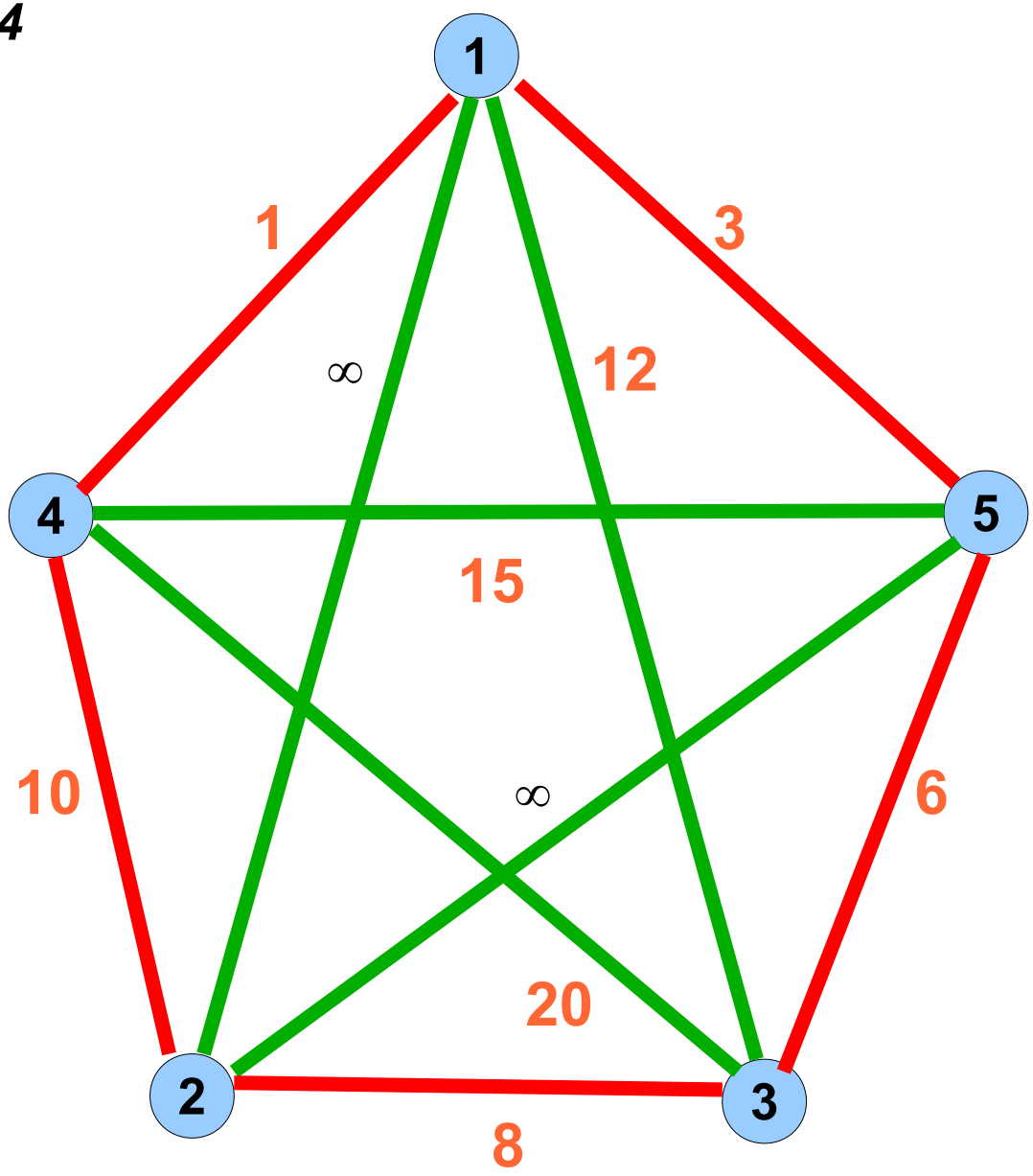
1,5,3,2,4

MinCost=28

STACK:

T5 ((2,5)(1,4))
T6 ((2,5)(1,5))
T7 ((3,5)(1,4))
T8 ((3,5)(1,2))
T9 ((3,5)(1,5))
T10 ((3,5)(2,5))
T11 ((3,5)(2,3))
T12 ((1,5)(1,2))
T13 ((1,5)(2,5))
T14 ((1,5)(2,3))

T4



STEP 6

(1,4) = ∞

(1,5) =3

(2,5) = ∞

(3,5) = 6

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=36

MinTour=

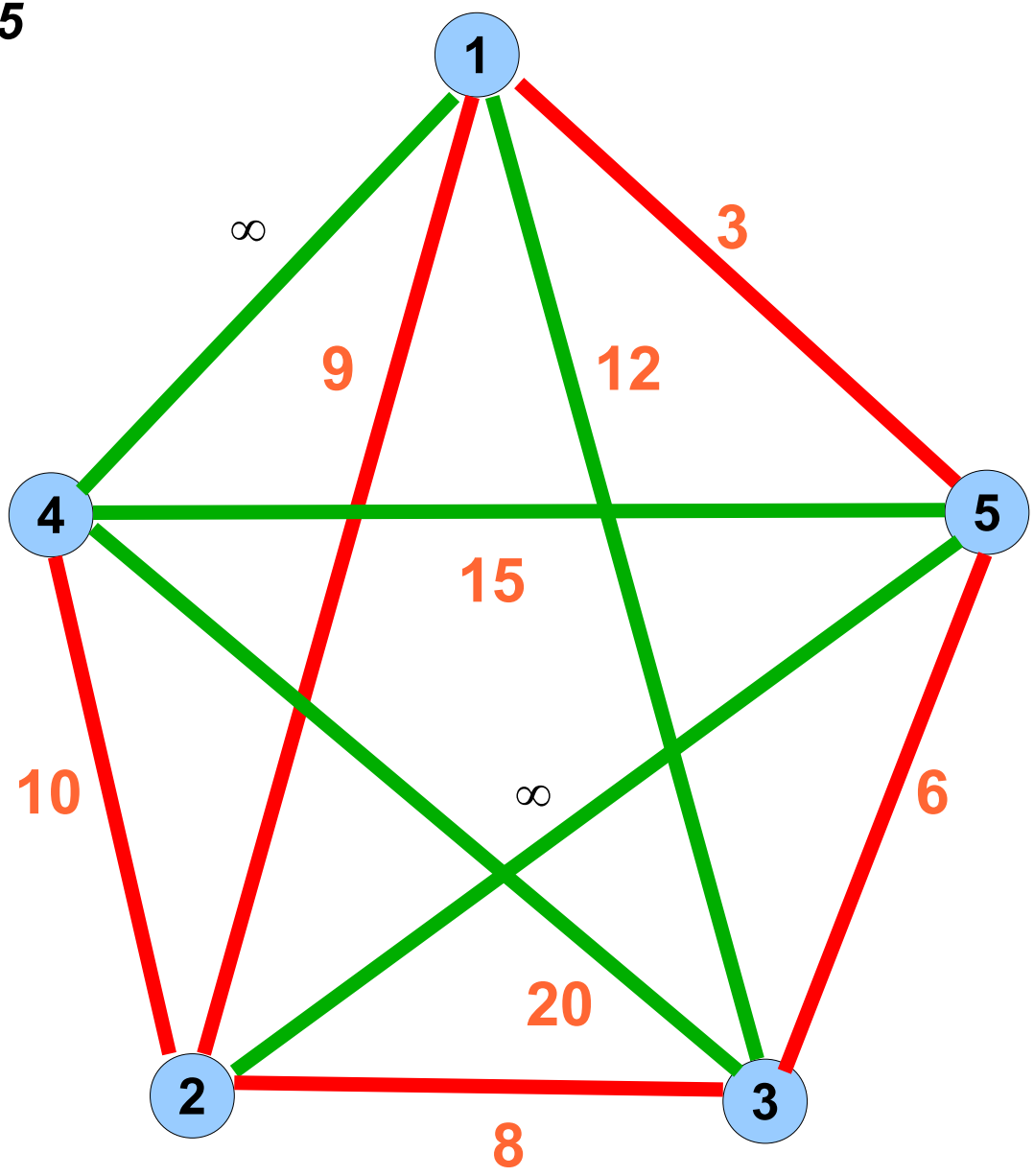
1,5,3,2,4

MinCost=28

STACK:

T6 ((2,5)(1,5))
T7 ((3,5)(1,4))
T8 ((3,5)(1,2))
T9 ((3,5)(1,5))
T10 ((3,5)(2,5))
T11 ((3,5)(2,3))
T12 ((1,5)(1,2))
T13 ((1,5)(2,5))
T14 ((1,5)(2,3))

T5



STEP 7

(1,4) =1

(1,5) = ∞

(2,5) = ∞

(3,5) = 6

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=34

MinTour=

1,5,3,2,4

MinCost=28

STACK:

T7 ((3,5)(1,4))

T8 ((3,5)(1,2))

T9 ((3,5)(1,5))

T10 ((3,5)(2,5))

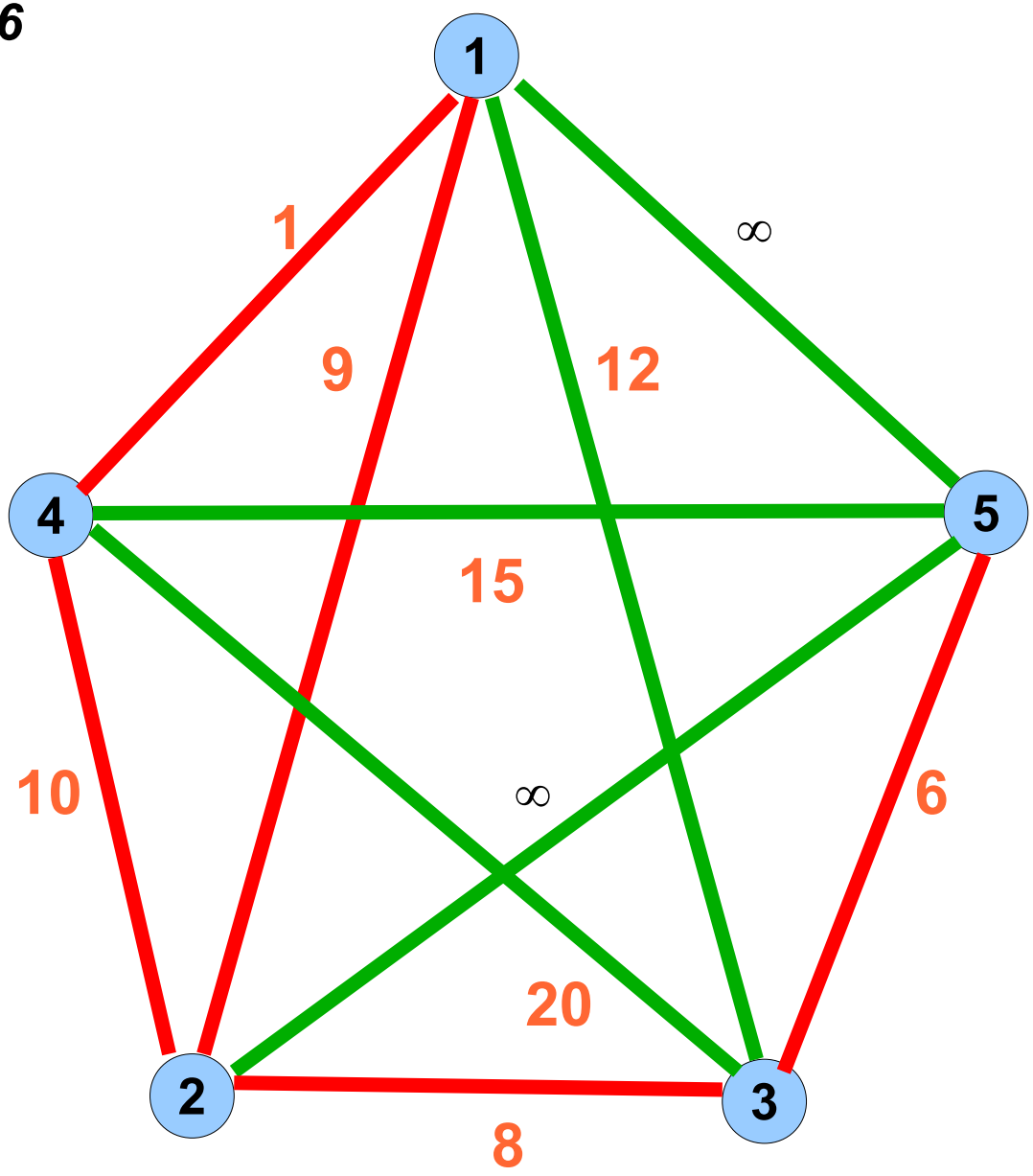
T11 ((3,5)(2,3))

T12 ((1,5)(1,2))

T13 ((1,5)(2,5))

T14 ((1,5)(2,3))

T6



STEP 8

(1,4) = ∞

(1,5) =3

(2,5) =5

(3,5) = ∞

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=35

MinTour=

1,5,3,2,4

MinCost=28

STACK:

T8 ((3,5)(1,2))

T9 ((3,5)(1,5))

T10 ((3,5)(2,5))

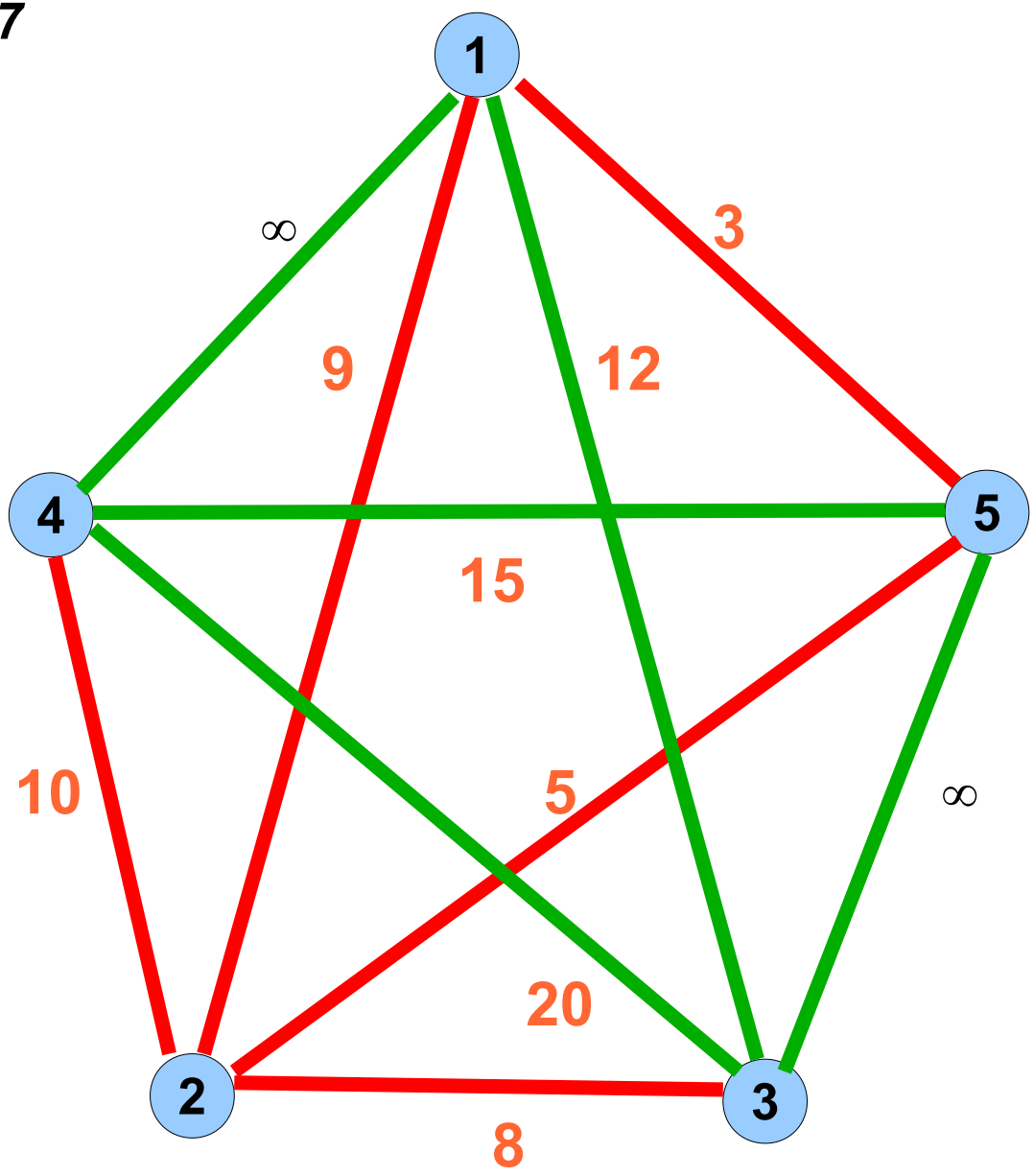
T11 ((3,5)(2,3))

T12 ((1,5)(1,2))

T13 ((1,5)(2,5))

T14 ((1,5)(2,3))

T7



STEP 9

(1,4) =1

(1,5) =3

(2,5) =5

(3,5) = ∞

(2,3) = 8

(1,2) = ∞

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=27

MinTour=

1,5,3,2,4

MinCost=28

STACK:

T9 ((3,5)(1,5))

T10 ((3,5)(2,5))

T11 ((3,5)(2,3))

T12 ((1,5)(1,2))

T13 ((1,5)(2,5))

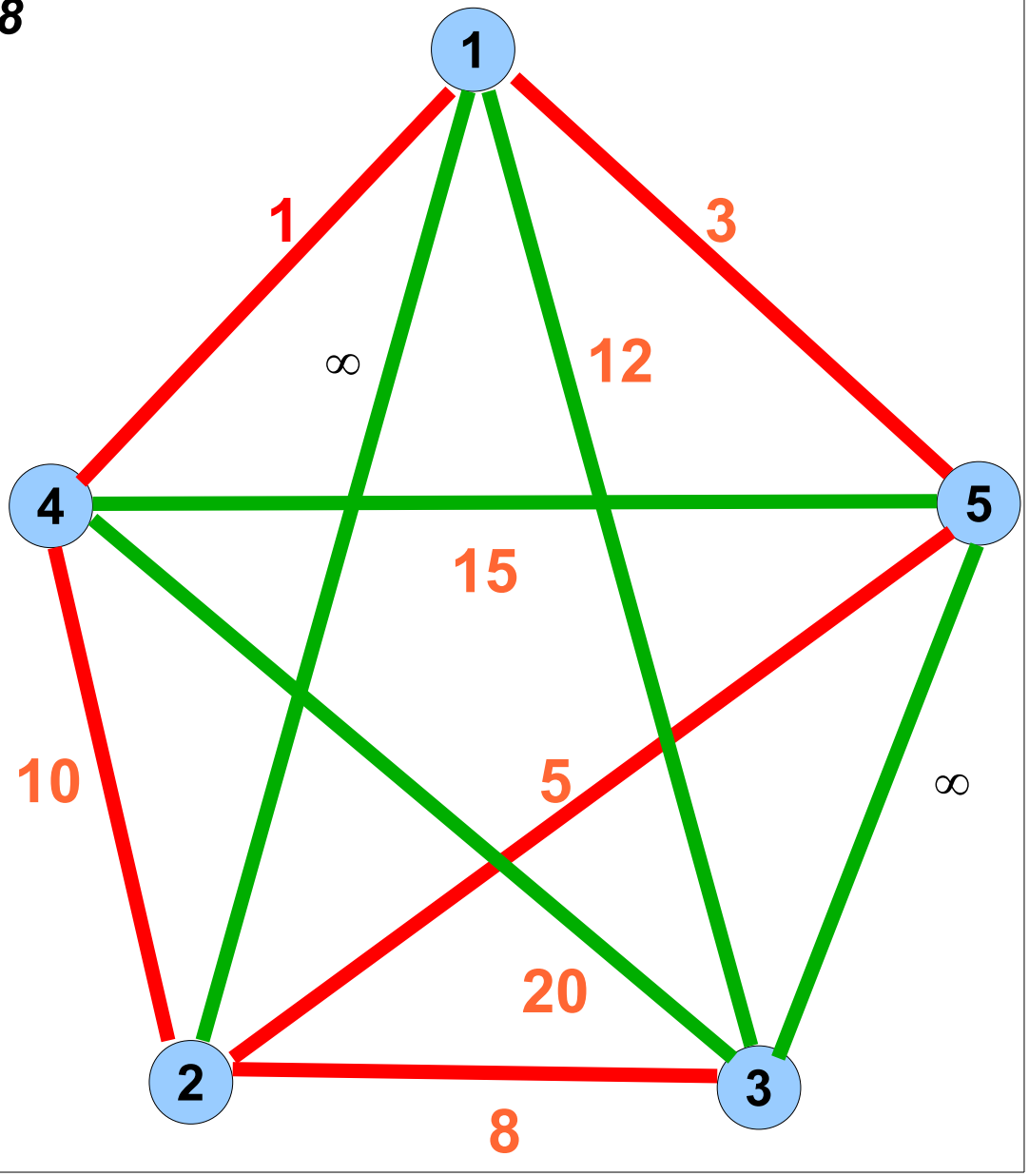
T14 ((1,5)(2,3))

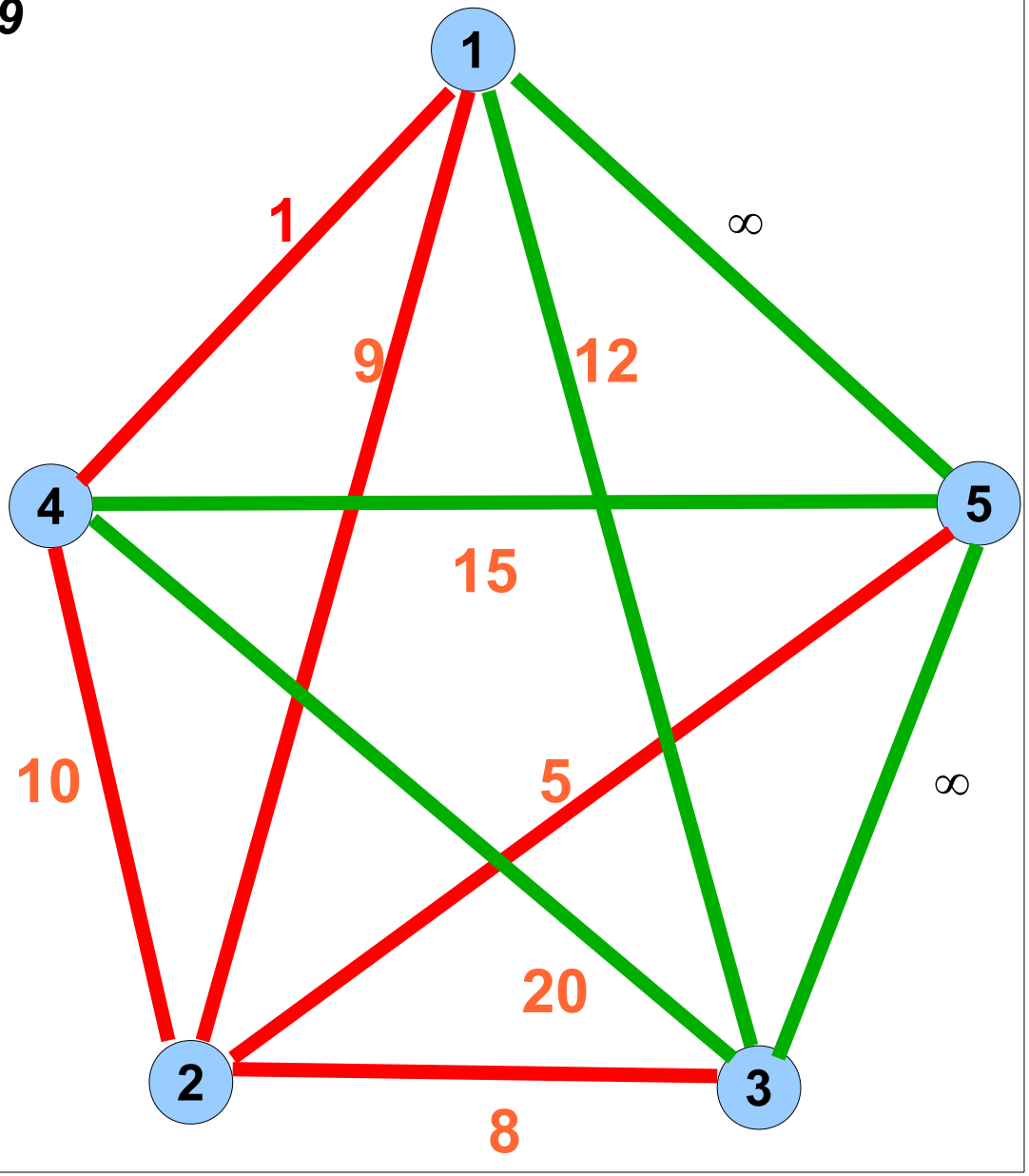
T15 ((1,2)(3,5)(2,4))

T16((1,2)(3,5)(2,5))

T17 ((1,2)(3,5)(2,3))

T8



STEP 10	STACK:	T9
(1,4) =1 (1,5) = ∞ (2,5) =5 (3,5) = ∞ (2,3) = 8 (1,2) = 9 (2,4) = 10 (1,3) = 12 (4,5) = 15 (3,4) = 20 *CurCost=33* *MinTour=* *1,5,3,2,4* *MinCost=28*	*T10 ((3,5)(2,5))* *T11 ((3,5)(2,3))* *T12 ((1,5)(1,2))* *T13 ((1,5)(2,5))* *T14 ((1,5)(2,3))* *T15 ((1,2)(3,5)(2,4))* *T16((1,2)(3,5)(2,5))* *T17 ((1,2)(3,5)(2,3))*	

STEP 11

(1,4) =1

(1,5) =3

(2,5) = ∞

(3,5) = ∞

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=31

MinTour=

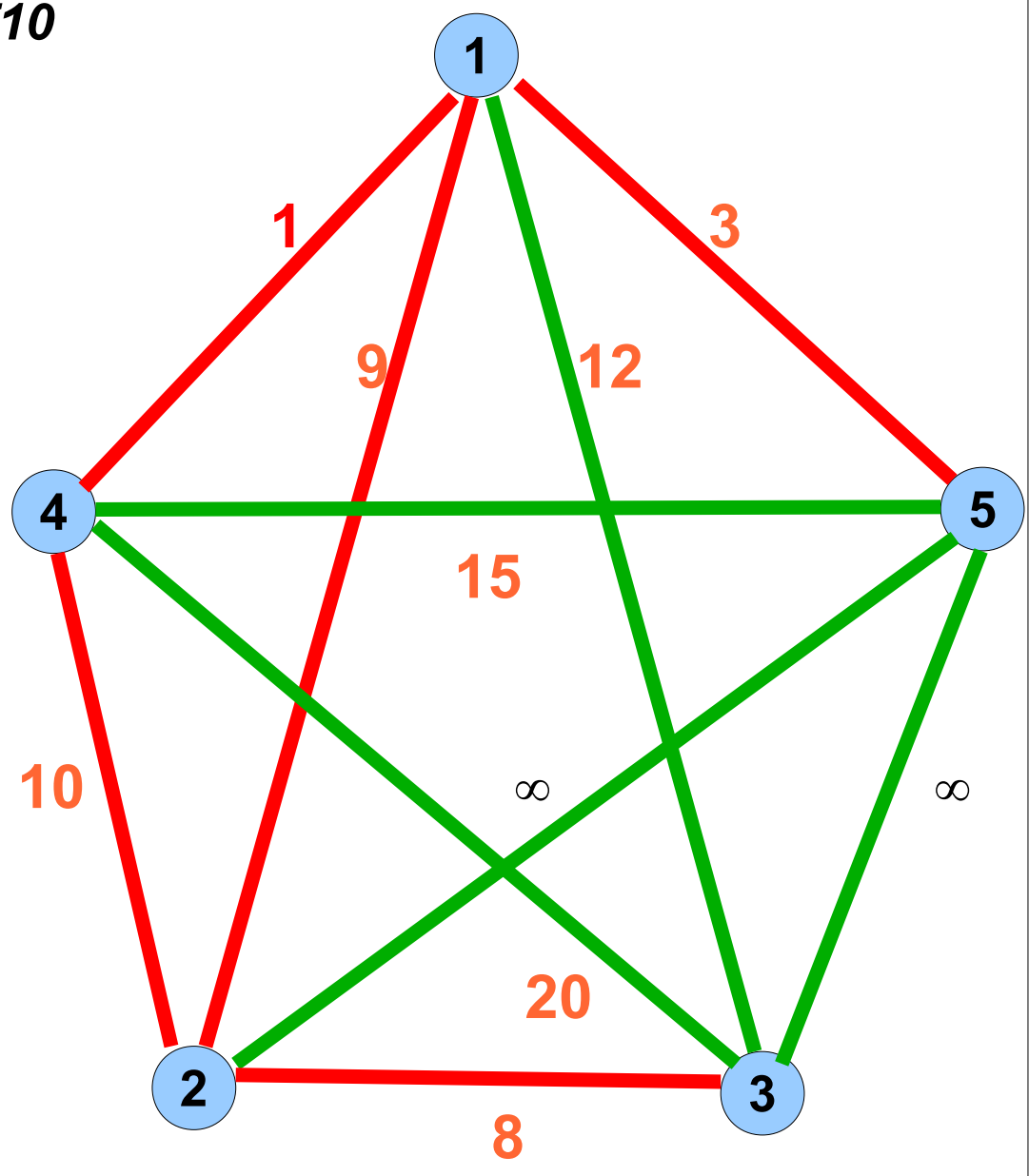
1,5,3,2,4

MinCost=28

STACK:

T11 ((3,5)(2,3))
T12 ((1,5)(1,2))
T13 ((1,5)(2,5))
T14 ((1,5)(2,3))
T15 ((1,2)(3,5)(2,4))
T16((1,2)(3,5)(2,5))
T17 ((1,2)(3,5)(2,3))

T10



STEP 12

(1,4) =1

(1,5) =3

(2,5) =5

(3,5) = ∞

(2,3) = ∞

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=30

MinTour=

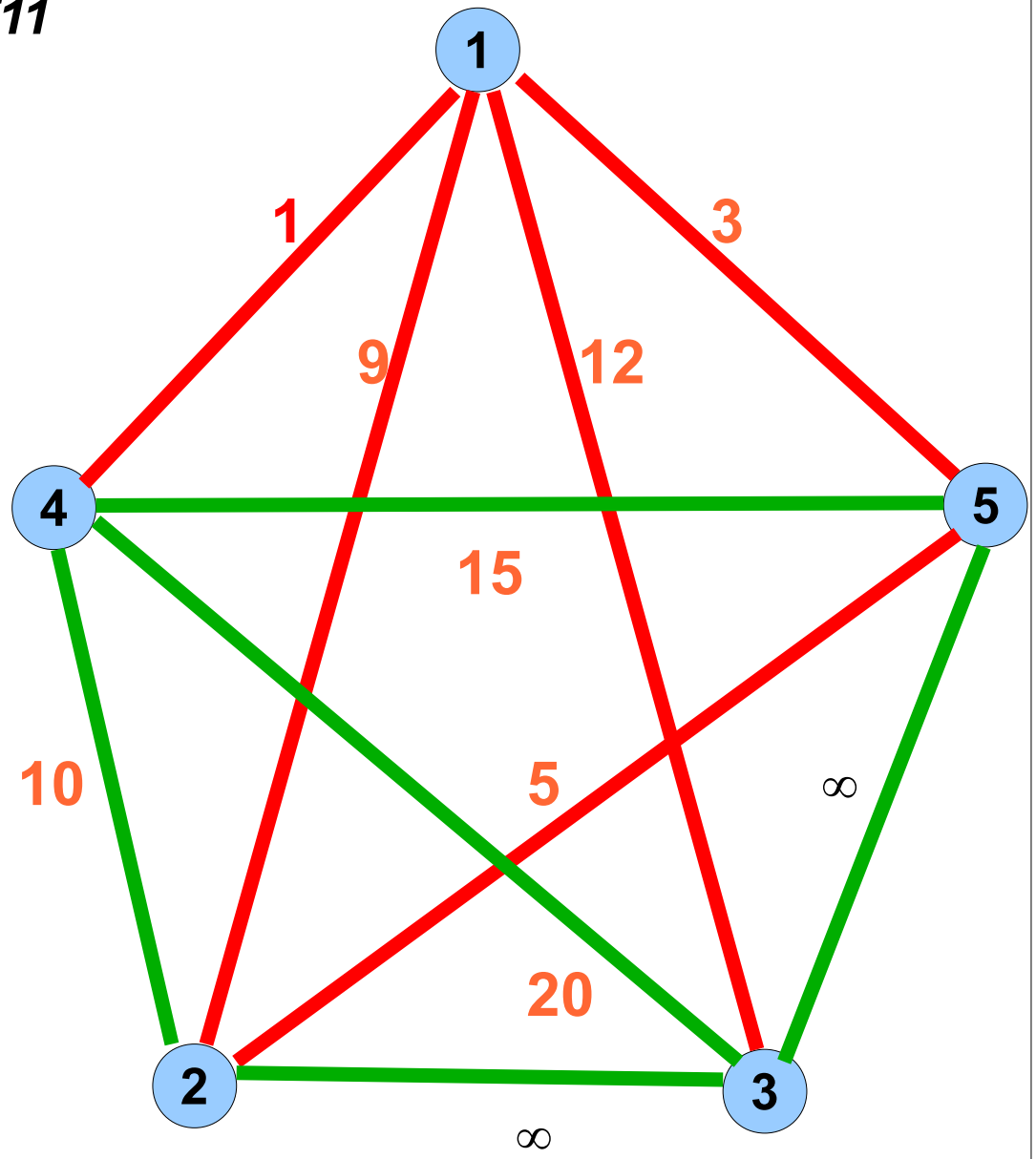
1,5,3,2,4

MinCost=28

STACK:

T12 ((1,5)(1,2))
T13 ((1,5)(2,5))
T14 ((1,5)(2,3))
T15 ((1,2)(3,5)(2,4))
T16((1,2)(3,5)(2,5))
T17 ((1,2)(3,5)(2,3))

T11



STEP 13

(1,4) =1

(1,5) = ∞

(2,5) =5

(3,5) = 6

(2,3) = 8

(1,2) = ∞

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=32

MinTour=

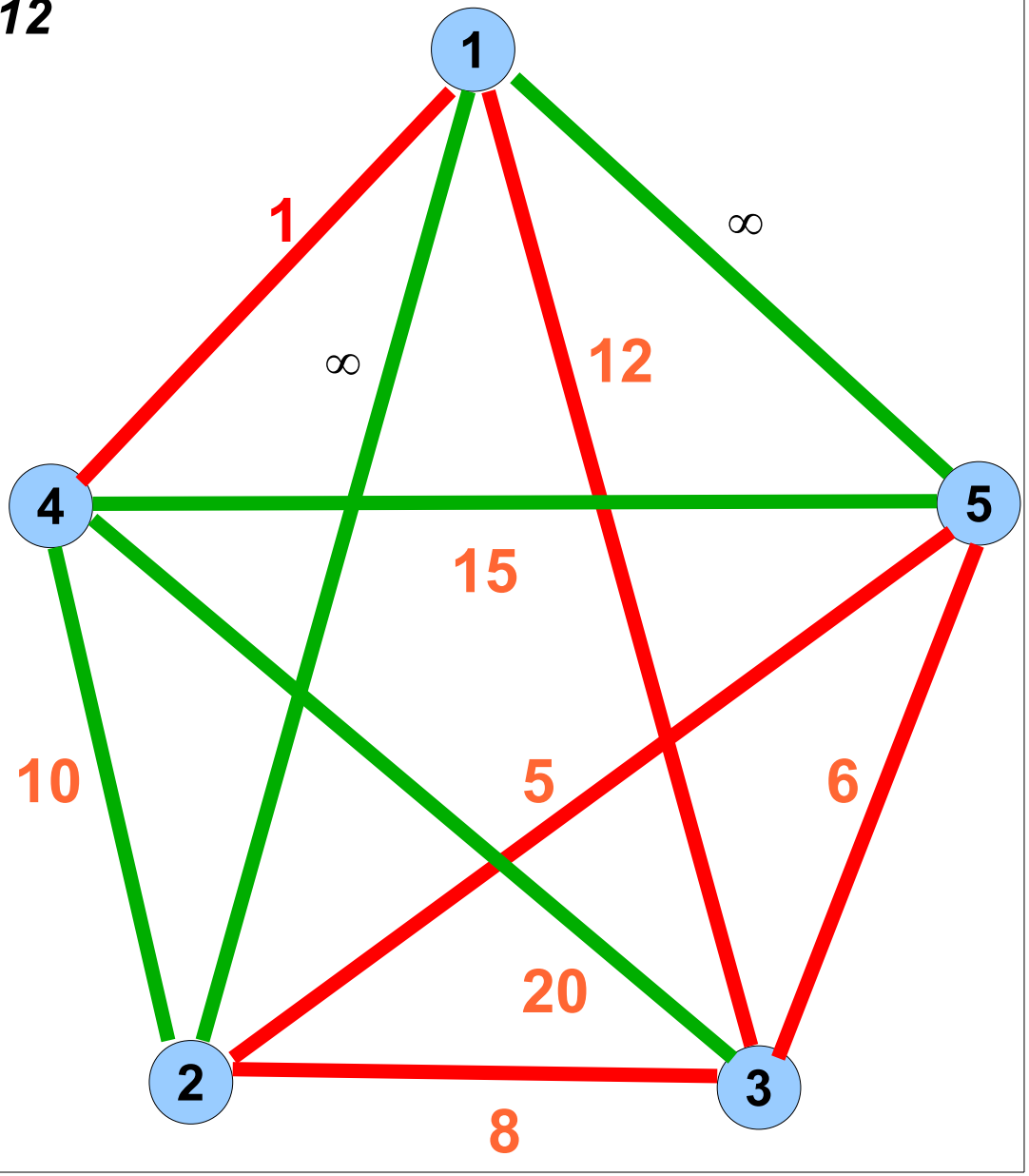
1,5,3,2,4

MinCost=28

STACK:

T13 ((1,5)(2,5))
T14 ((1,5)(2,3))
T15 ((1,2)(3,5)(2,4))
T16((1,2)(3,5)(2,5))
T17 ((1,2)(3,5)(2,3))

T12



STEP 14

(1,4) =1

(1,5) = ∞

(2,5) = ∞

(3,5) = 6

(2,3) = 8

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=34

MinTour=

1,5,3,2,4

MinCost=28

STACK:

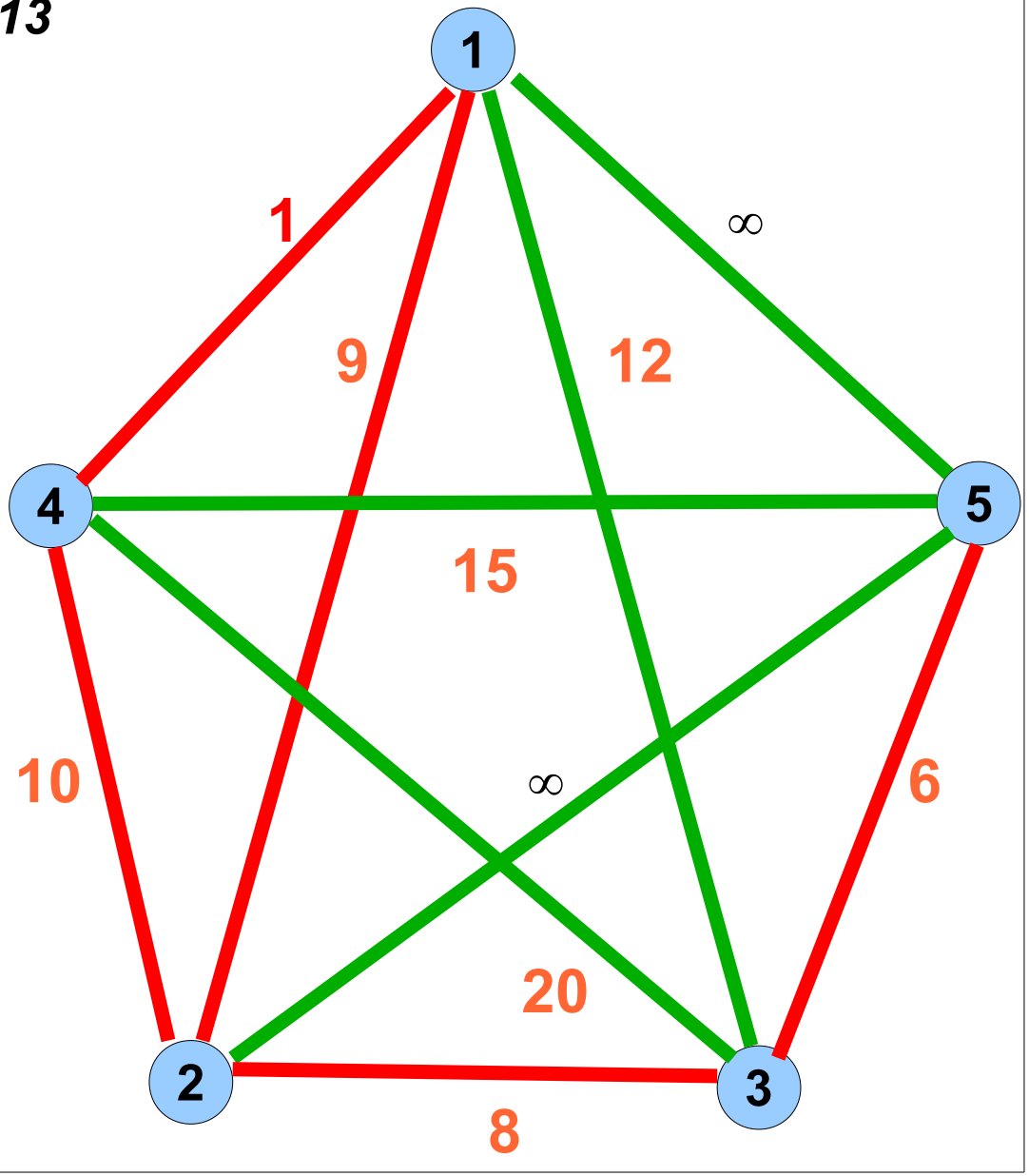
T14 ((1,5)(2,3))

T15 ((1,2)(3,5)(2,4))

T16((1,2)(3,5)(2,5))

T17 ((1,2)(3,5)(2,3))

T13



STEP 15

(1,4) =1

(1,5) = ∞

(2,5) = 5

(3,5) = 6

(2,3) = ∞

(1,2) = 9

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=31

MinTour=

1,5,3,2,4

MinCost=28

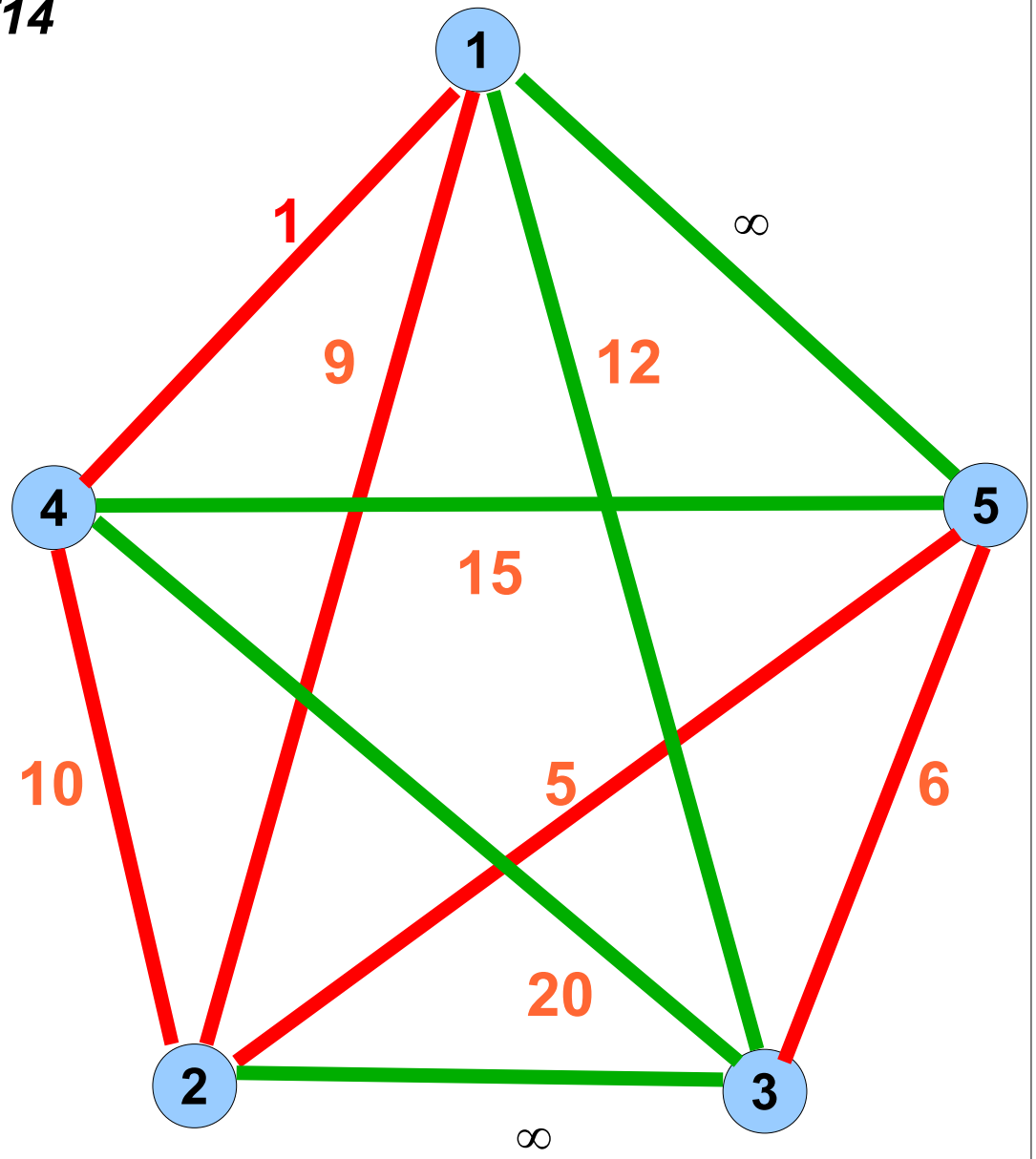
STACK:

T15 ((1,2)(3,5)(2,4))

T16((1,2)(3,5)(2,5))

T17 ((1,2)(3,5)(2,3))

T14



STEP 16

(1,4) =1

(1,5) =3

(2,5) = 5

(3,5) = ∞

(2,3) = 8

(1,2) = ∞

(2,4) = ∞

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=29

MinTour=

1,5,3,2,4

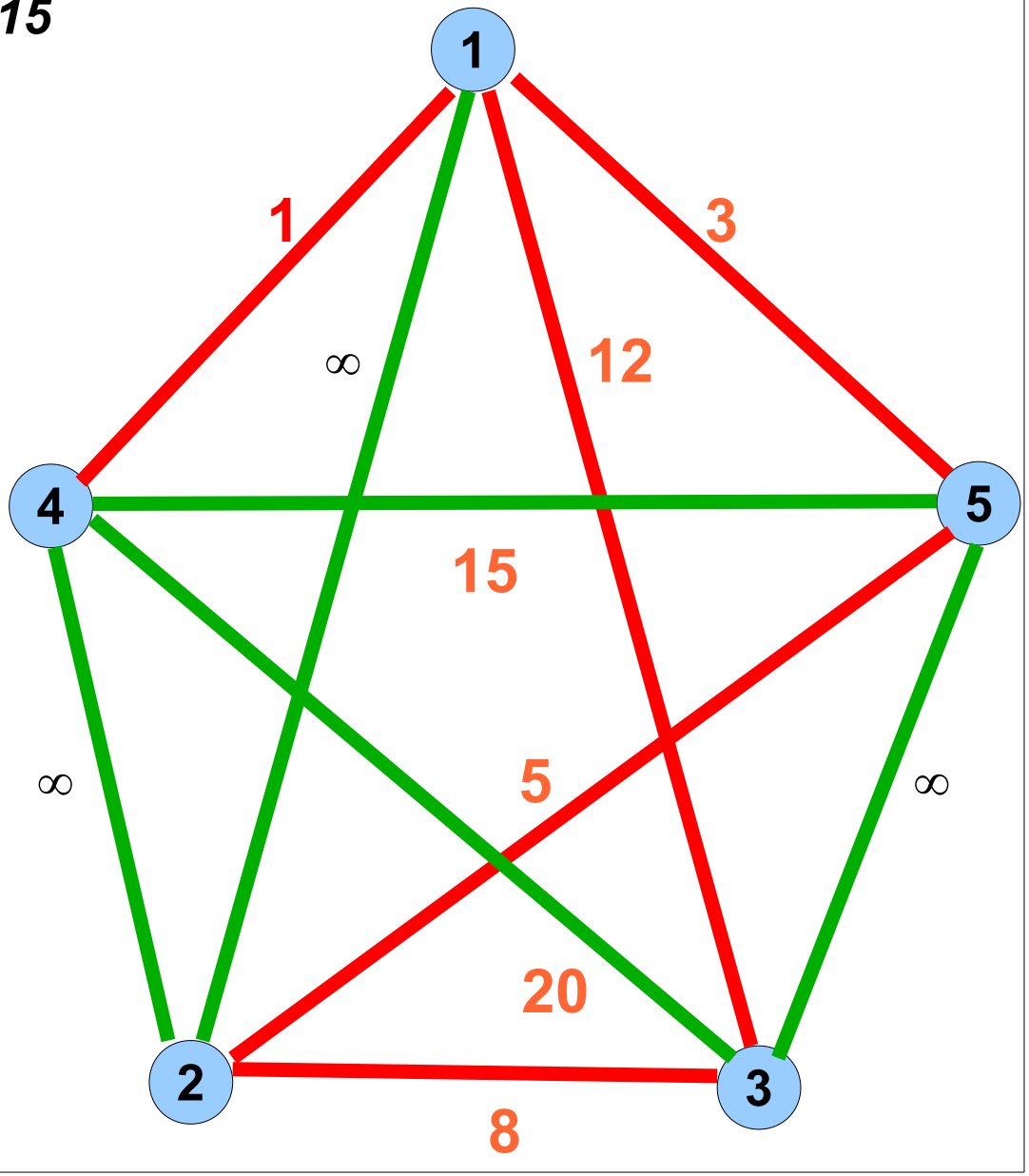
MinCost=28

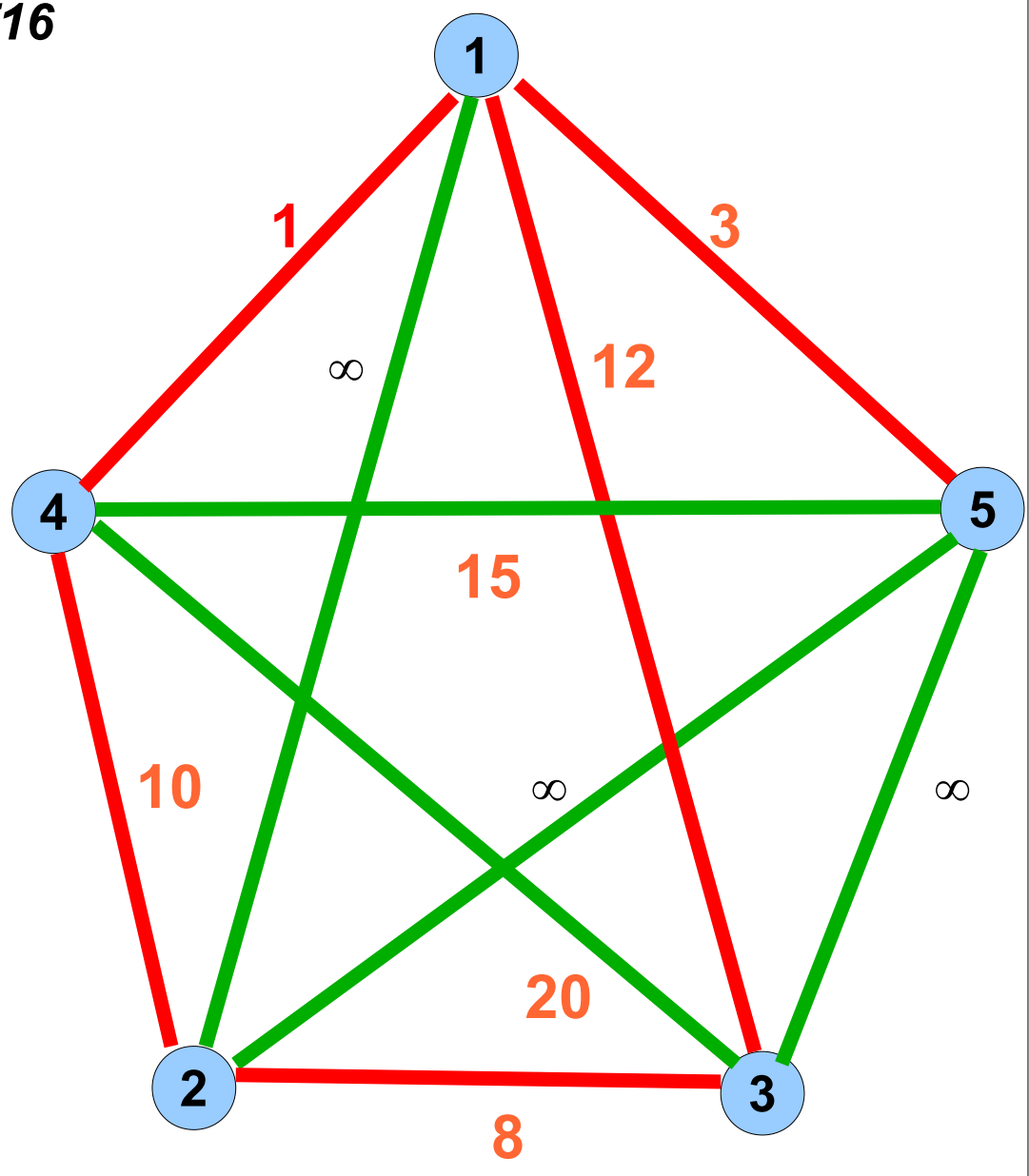
STACK:

T16((1,2)(3,5)(2,5))

T17 ((1,2)(3,5)(2,3))

T15



STEP 17	STACK:	T16
(1,4) =1 (1,5) =3 (2,5) = ∞ (3,5) = ∞ (2,3) = 8 (1,2) = ∞ (2,4) = 10 (1,3) = 12 (4,5) = 15 (3,4) = 20 *CurCost=34* *MinTour=* *1,5,3,2,4* *MinCost=28*	*T17 ((1,2)(3,5)(2,3))*	

STEP 18

(1,4) =1

(1,5) =3

(2,5) = 5

(3,5) = ∞

(2,3) = ∞

(1,2) = ∞

(2,4) = 10

(1,3) = 12

(4,5) = 15

(3,4) = 20

CurCost=31

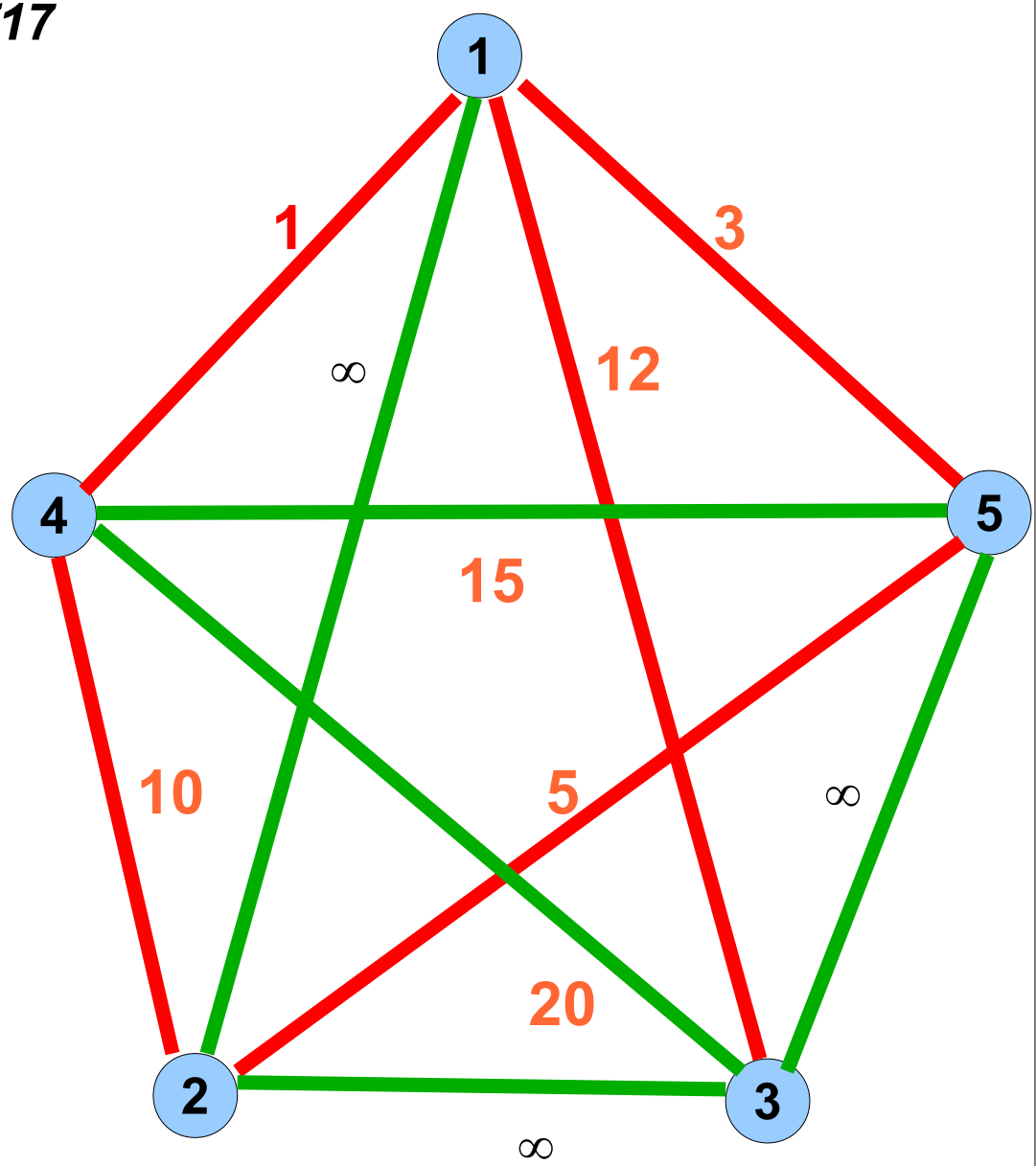
Minimal Tour

1,5,3,2,4

Cost = 28

STACK:

T17



INTEGER-PROGRAMMING APPROACH

Let us consider K_{n+1} with the set of nodes $\{0,1,2,\ldots,n\}$ and let

$$C=\begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0n} \\ c_{10} & c_{11} & \cdots & c_{1n} \\ \vdots & \vdots & \ddots & \ddots \\ c_{n0} & c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

be the the cost matrix, that is, c_{ij} is the cost of travelling from node i to node j.

The problem of finding a tour $t=(0,i_1,i_2,\ldots,i_n)$ for a travelling salesman starting from node 0 such that $\sum_{k=1}^{n-1} c_{i_k i_{k+1}}+c_{0 i_1}+c_{i_n 0}\leq\sum_{k=1}^{n-1} c_{j_k j_{k+1}}+c_{0 j_1}+c_{j_n 0}$ for any tour $t=(j_0,j_1,j_2,\ldots,j_n)$ can be formulated in terms of integer-programming as follows:

Introducing variables $x_{ij}, 0 \le i \le n, 0 \le j \le n$ with

$$x_{ij} = \begin{cases} 1 & \text{if the travelling salesman travels from node } i \text{ to node } j \\ 0 & \text{otherwise} \end{cases}$$

we solve the minimization problem

$$\sum_{i=0}^{n} \sum_{j=0}^{n} c_{ij} x_{ij} \qquad (1)$$

under the restrictions

$$\sum_{i=0}^{n} x_{ij} = 1 \qquad j = 1, 2, \ldots, n \qquad (2)$$

$$\sum_{j=0}^{n} x_{ij} = 1 \qquad i = 1, 2, \ldots, n \qquad (3)$$

$$u_i - u_j + n x_{ij} \le n - 1 \qquad i, j = 1, 2, \ldots, n \quad i \ne j \qquad (4)$$

$$\sum_{i=0}^{n}\sum_{j=0}^{n} c_{ij}x_{ij} \qquad (1)$$

$$\sum_{i=0}^{n} x_{ij}=1 \qquad j=1,\,2,\,\ldots,\,n \qquad (2)$$

$$\sum_{j=0}^{n} x_{ij}=1 \qquad i=1,\,2,\,\ldots,\,n \qquad (3)$$

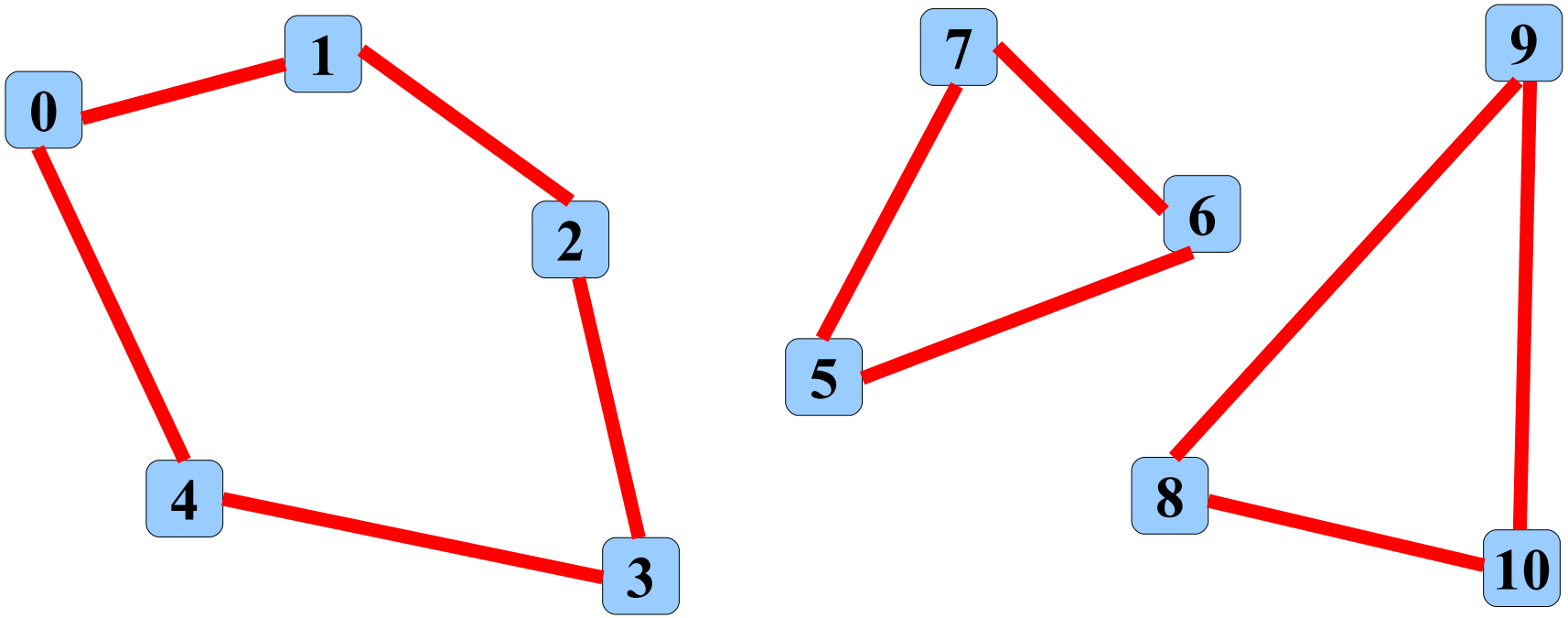
$$u_i-u_j+n\,x_{ij}\le n-1 \qquad i,\,j=1,\,2,\,\ldots,\,n \quad i\neq j \qquad (4)$$

Primarily, there are no integer restrictions on the variables u_i in (4). However, it can be shown that, without loss of generality, they may be thought of as integer variables as well.

It is not difficult to see that such an integer programming problem is equivalent to the original TSP problem.

Indeed, the objective function $\sum_{i=0}^{n}\sum_{j=0}^{n} c_{ij}x_{ij}$ clearly defines the the optimum tour cost. while the restrictions $\sum_{i=0}^{n} x_{ij}=1$ $j=1, 2, \ldots, n$ mean that the travelling salesman leaves every node exactly once (except for node 0) and the restrictions $\sum_{j=0}^{n} x_{ij}=1$ $i=1, 2, \ldots, n$ guarantee that he enters every node exactly once (except for node 0).

The role of the restrictions $u_i - u_j + n\,x_{ij} \leq n-1$, $i, j = 1, 2, \ldots, n \quad i \neq j$ is to eliminate routes consisting of two or more disjunct cycles:



which also satisfy restrictions (2) and (3). Let us use the above example to show how they work.

For the cycle 5, 6, 7 we have

$$u_5 - u_6 + 10\,x_{56} \le 9$$

$$u_6 - u_7 + 10\,x_{67} \le 9$$

$$u_7 - u_5 + 10\,x_{75} \le 9$$

Since $x_{56} = x_{67} = x_{75} = 1$, by adding the above inequalities, we obtain

$30 \le 27$ so that such a tour would be eliminated.

On the other hand, for a feasible tour, say $(0, 5, 4, 8, 7, 2, 3, 6, 10, 1, 9)$ we may put u_1=9, u_2=5, u_3=6, u_4=2, u_5=1, u_6=7, u_7=4, u_8=3, u_9=10, u_{10}=8. For example u_6=7 because node 6 occurs as seventh in the tour. In this way, for example, the inequality $u_7 - u_2 + 10\,x_{54} \le 9$ turns into $4 - 5 + 10 \cdot 1 \le 9$ while, say, the inequality $u_7 - u_2 + 10\,x_{54} \le 9$ turns into $10 - 1 + 10 \cdot 0 \le 9$.

HEURISTIC METHODS

For larger numbers of nodes, the above algorithms are of little help. In practical problems, however, we might settle for near-optimum solutions computed in a reasonable time. Algorithms enabling this are called heuristics.

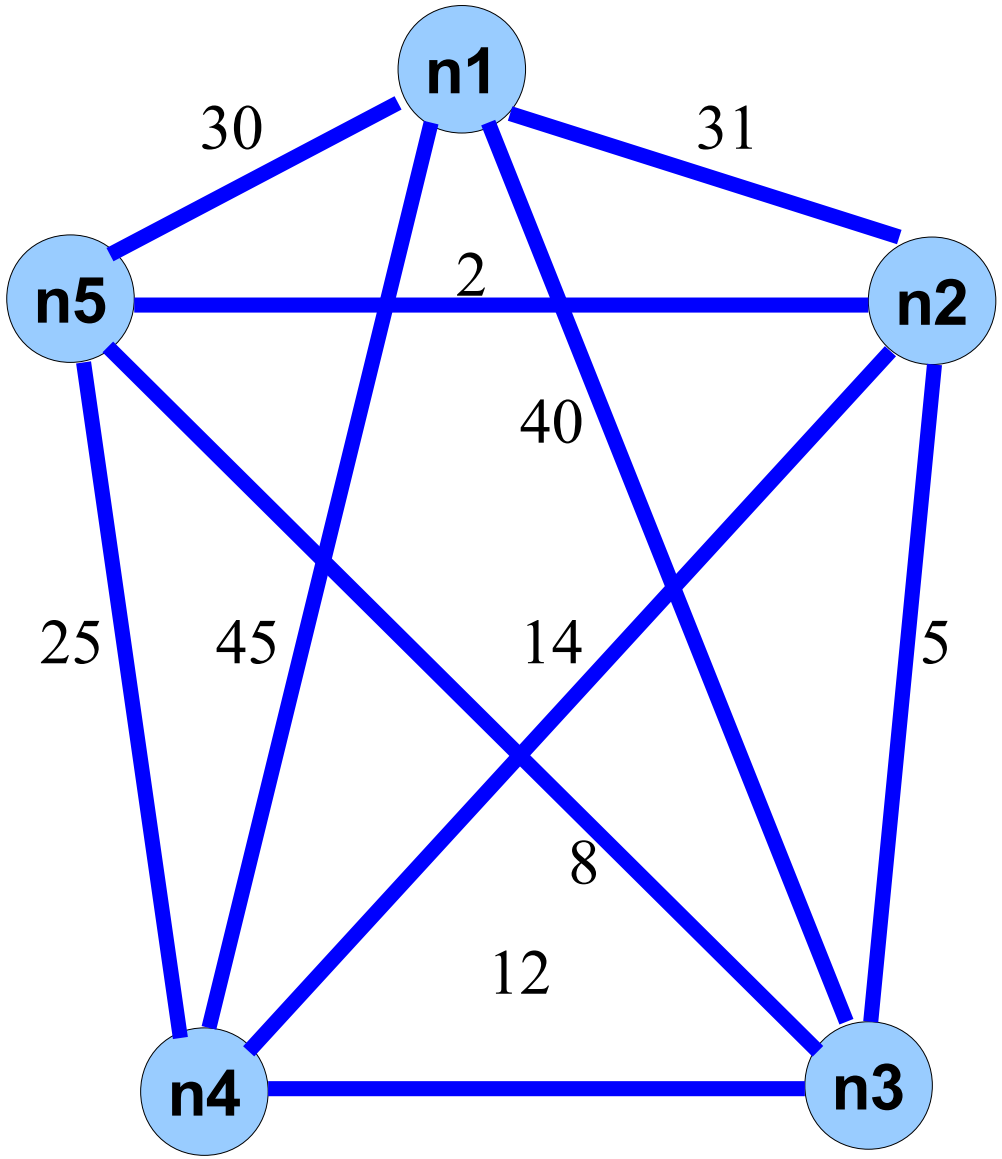
A heuristic is an algorithm that solves a particular problem without guaranteeing an absolute optimum, which, among others, means that it need not be based on an exact theoretical background. A problem may be considered solved by a heuristic, for example, if a solution is found that is known to lie within a pre-set "distance" from an optimum one or at least this may be assumed with a reasonable probability. Another stopping rule for such an algorithm may be the solution time exceeding a pre-set limit.

Local changes

When performing the search, instead of just going through a list of objects, calculating the cost of each one separately to find the shortest one, we might adopt the following approach: we try to perform a small "local" change of the object to pass to a "neighbouring" one to see whether this "local" change has reduced the price. This improvement procedure may be then iterated until no local change can bring about any reduction in price. The question is then whether the resulting object is the optimum one that we set out to search.

For some type of graphs this question may be answered in the positive. This is, for example, the case with spanning trees.

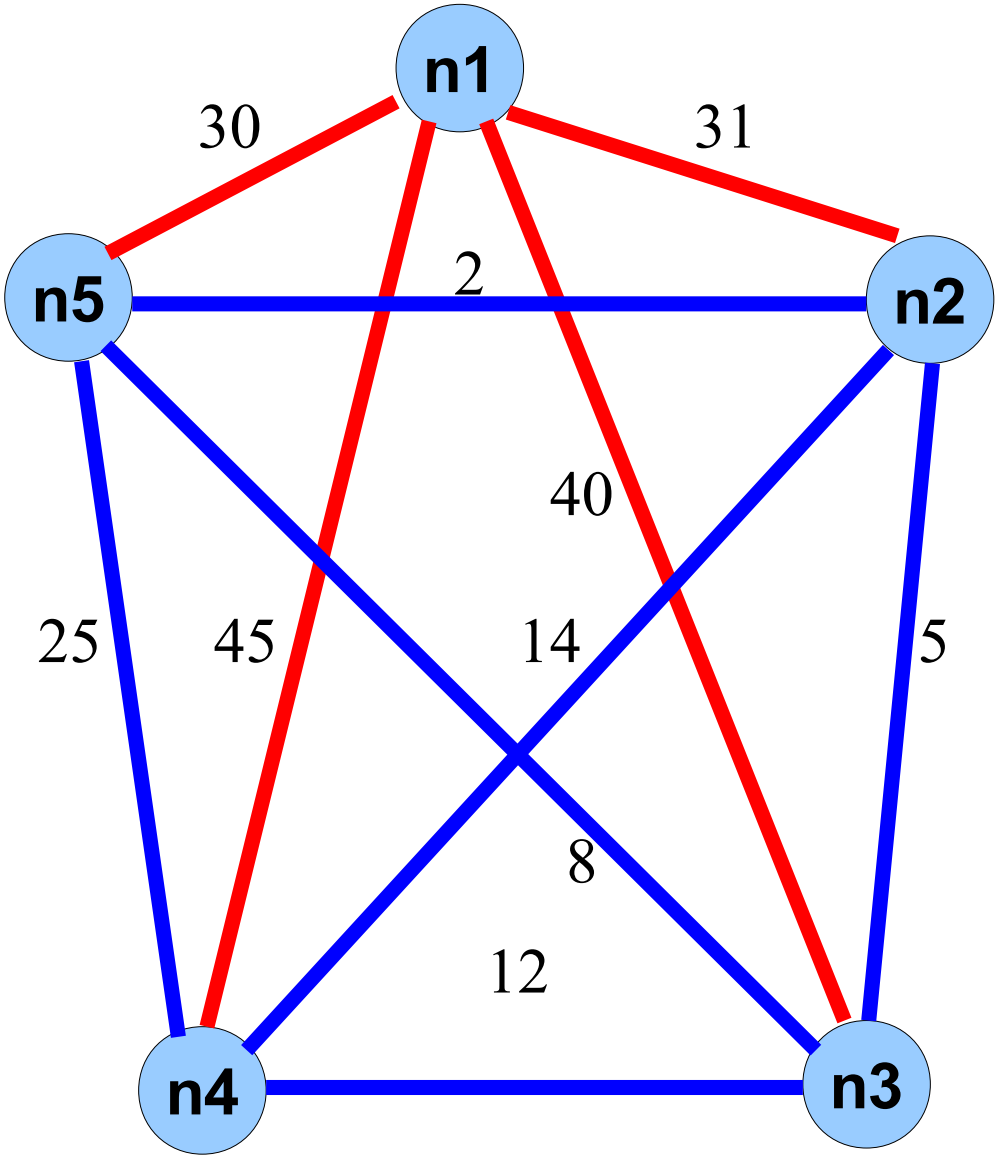
Find a minimum spanning tree of the following complete graph K_5:



Edge costs:

(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

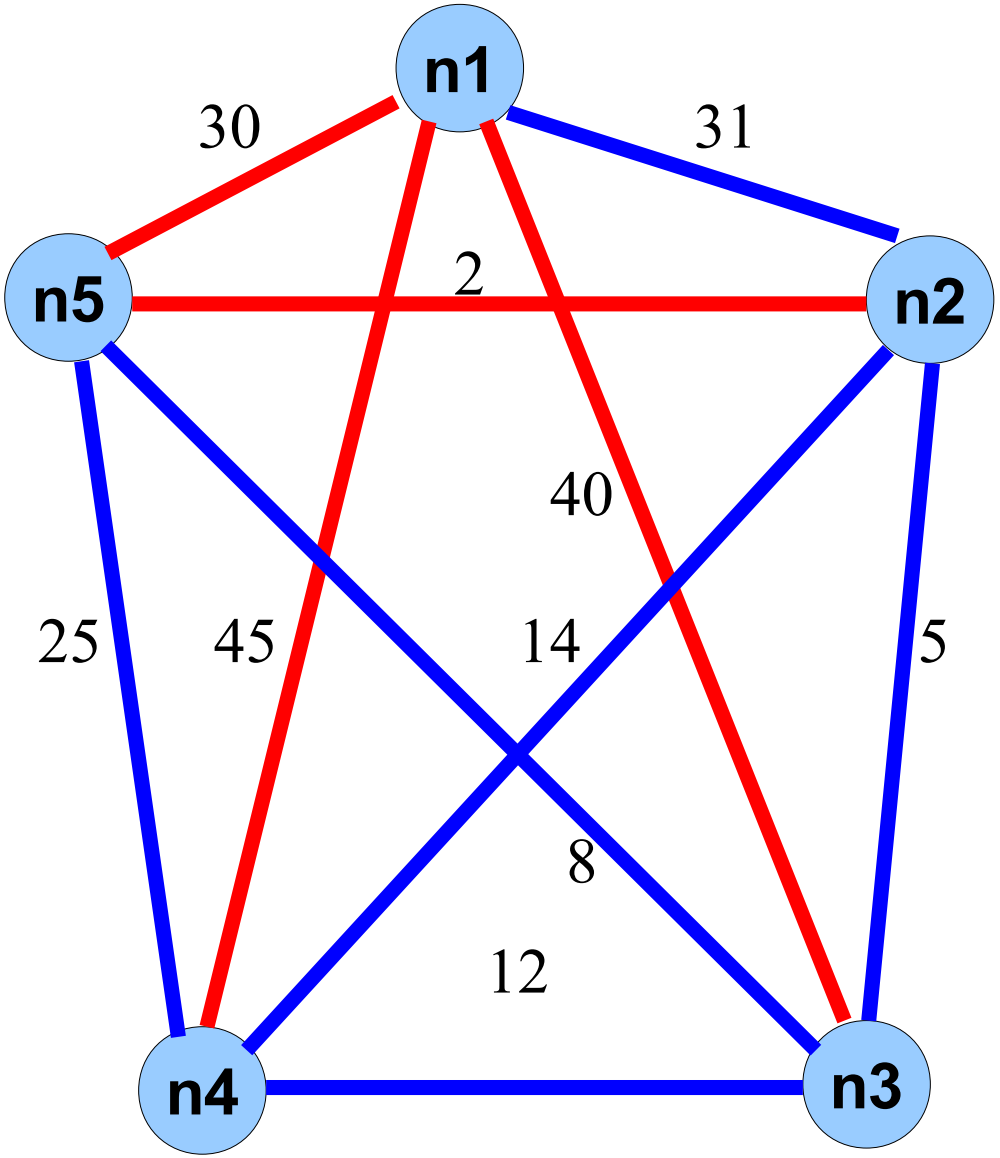
Here, the local change is represented by adding one edge and removing another in the circle thus created.



Edge costs:

(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

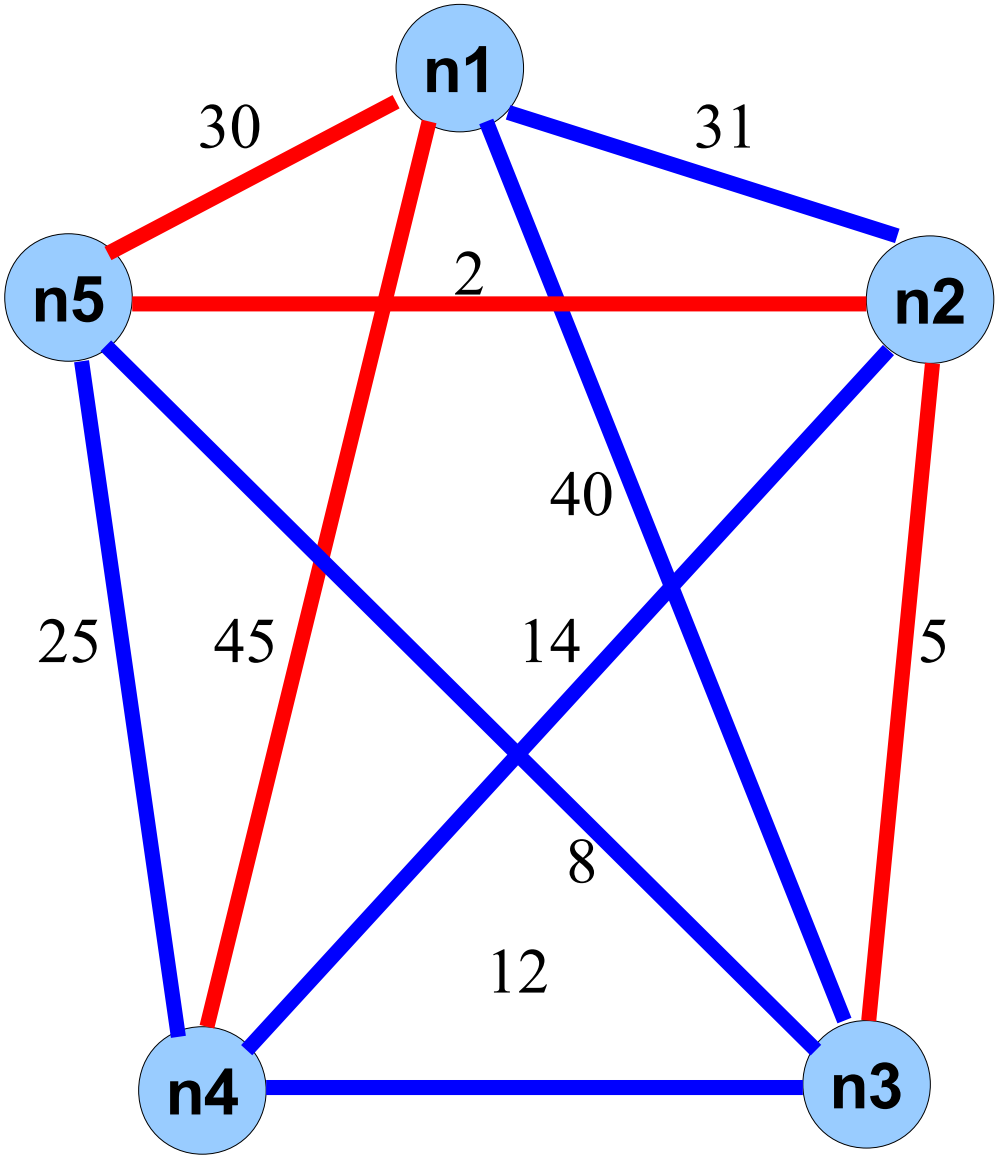
Initial cost = 146



Edge costs:

(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

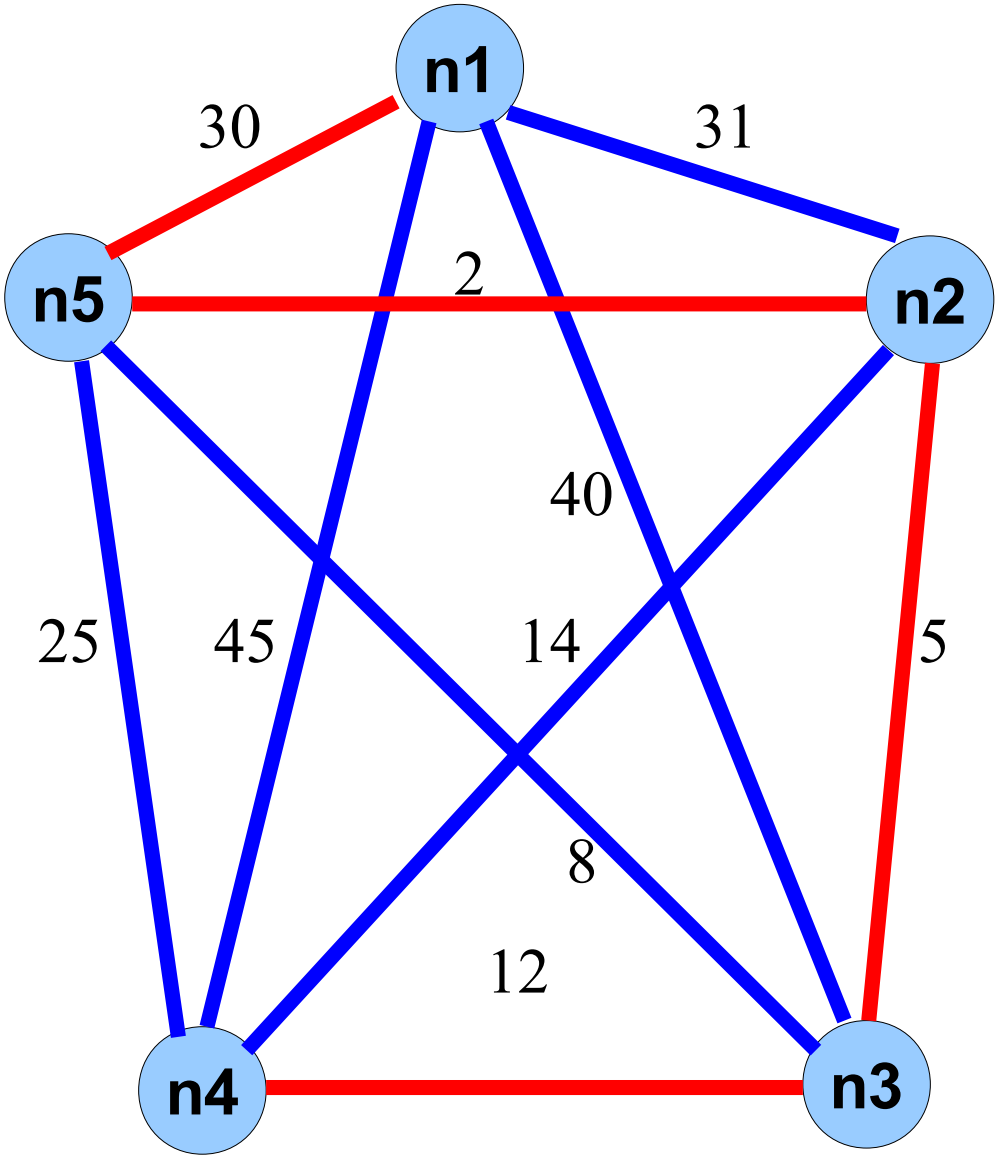
Next lower cost = 117



Edge costs:

(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

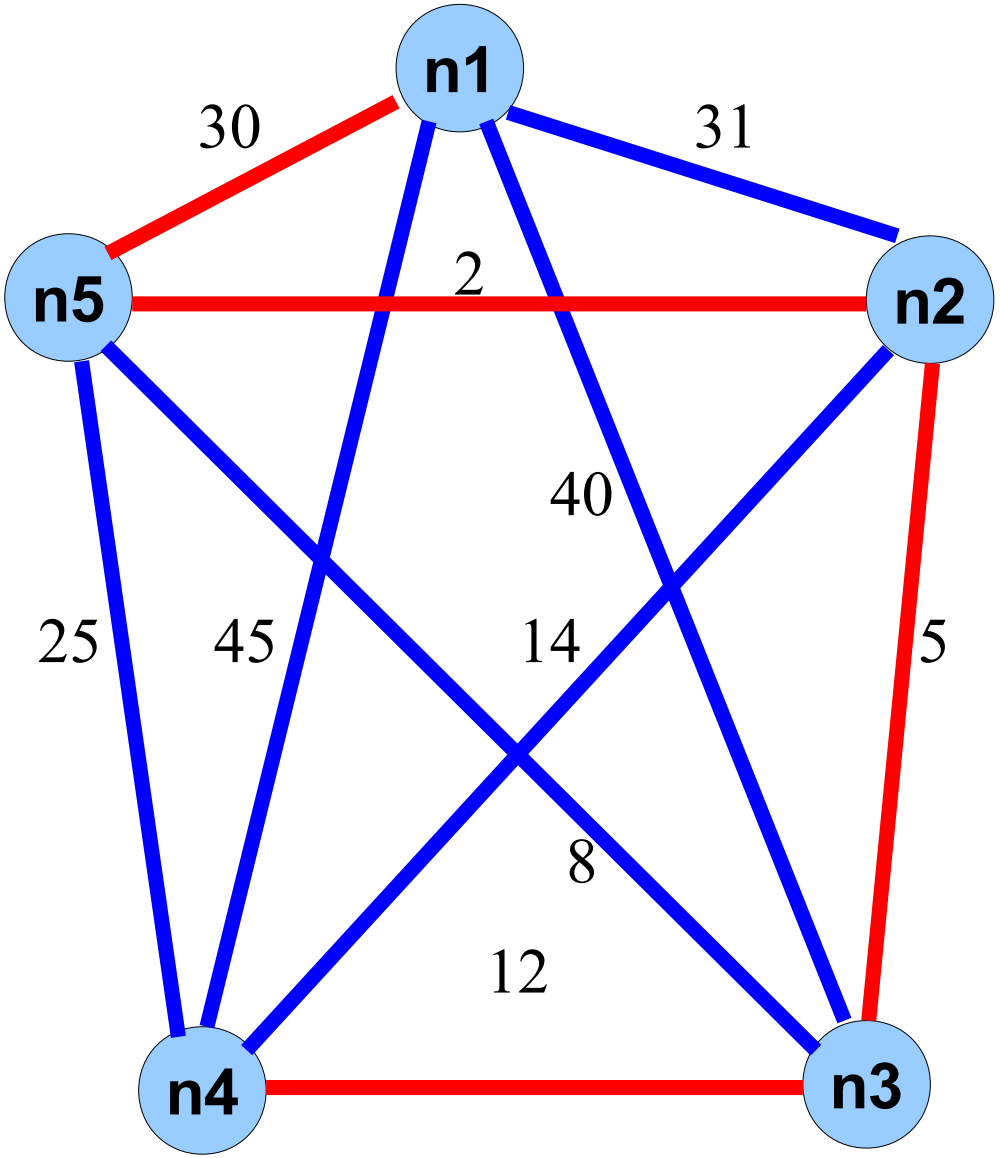
Next lower cost = 82



Edge costs:

(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

Next lower cost = 49

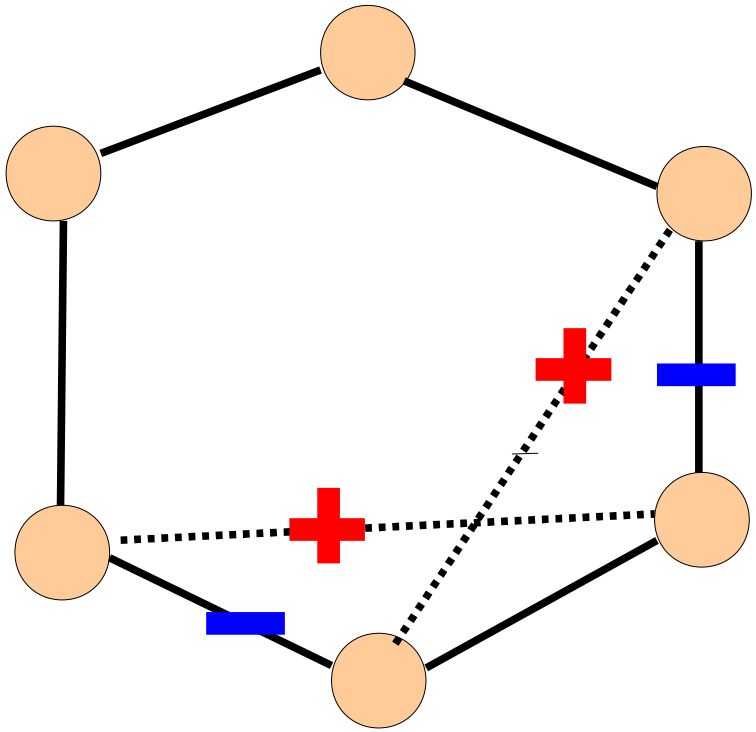


Edge costs:

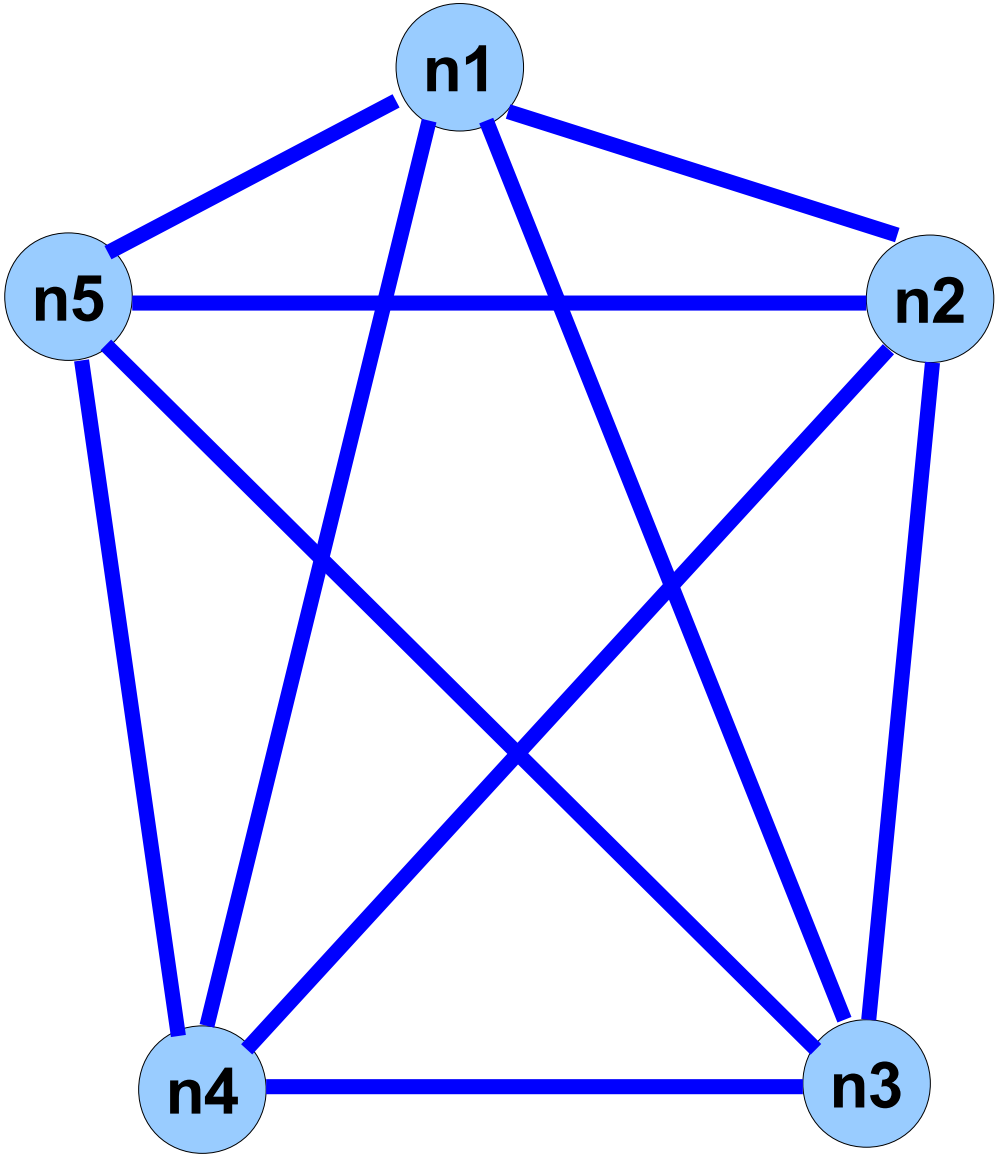
(n5,n2) = 2
(n2,n3) = 5
(n5,n3) = 8
(n4,n3) = 12
(n4,n2) = 14
(n5,n4) = 25
(n5,n1) = 30
(n1,n2) = 31
(n1,n3) = 40
(n1,n4) = 45

Minimum cost = 49
No further local changes will reduce the cost

The resulting graph is really the minimum spanning tree. This is a consequence of Theorem 7 of the minimum-spanning-tree lecture. However, for a TSP, no such guarantee exists as the following example shows. Here the local change is represented by two neighbouring nodes in a tour being swapped.

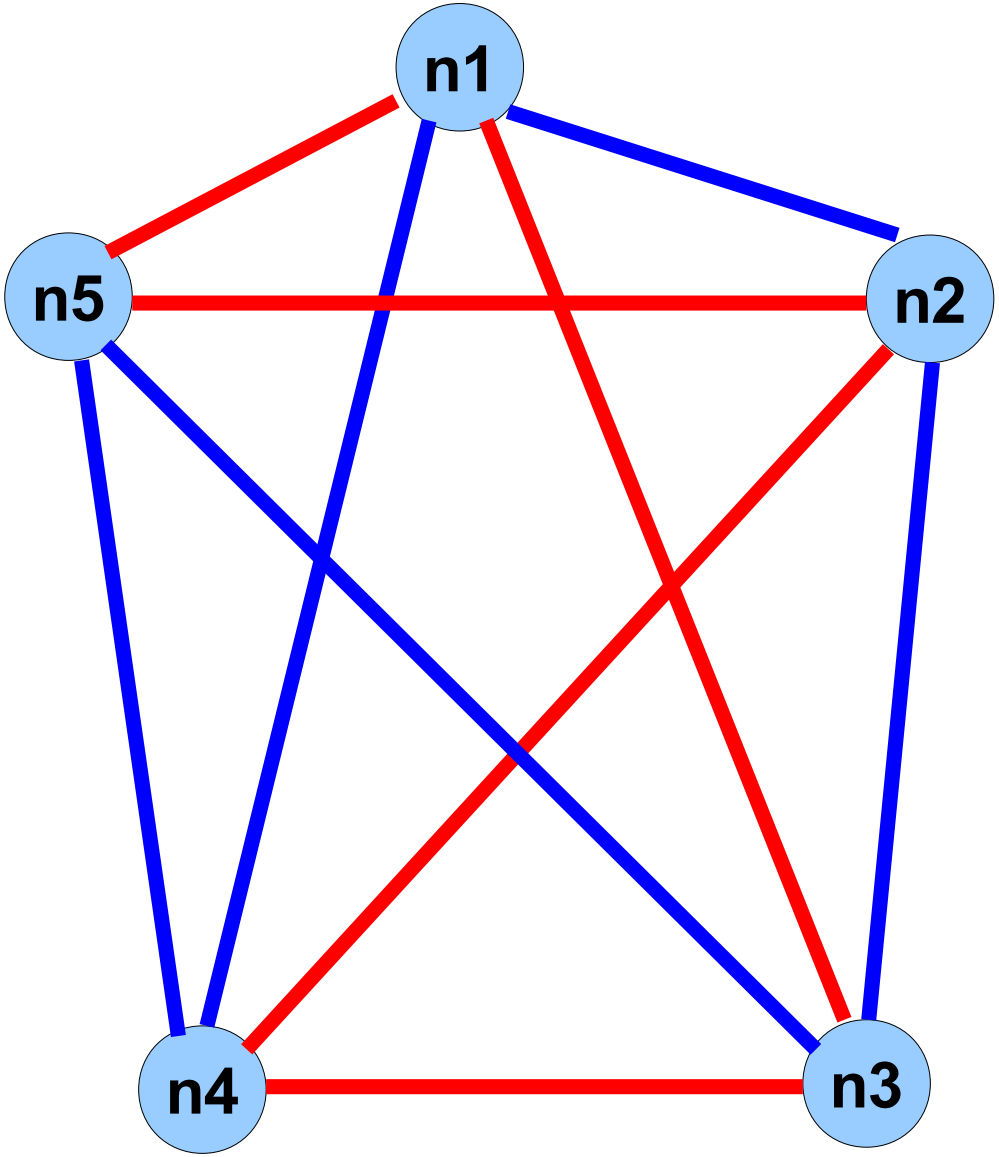


Find a minimum tour of the following complete graph K_5:



Edge costs:

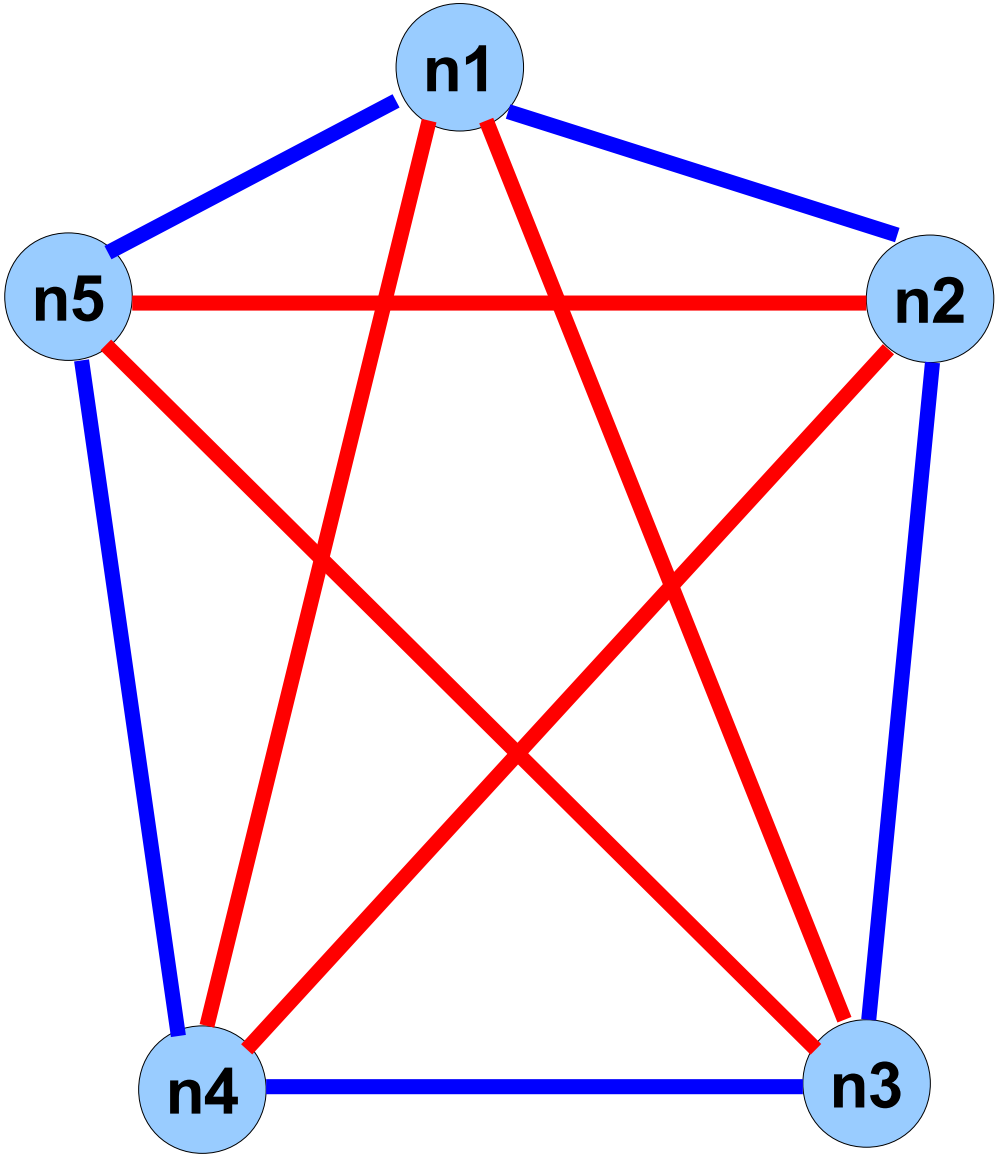
(n3,n5) = 91
(n4,n5) = 223
(n1,n4) = 255
(n2,n5) = 288
(n1,n2) = 328
(n1,n5) = 446
(n2,n4) = 492
(n2,n3) = 536
(n1,n3) = 589
(n3,n4) = 754



Edge costs:

(n3,n5) = 91
(n4,n5) = 223
(n1,n4) = 255
(n2,n5) = 288
(n1,n2) = 328
(n1,n5) = 446
(n2,n4) = 492
(n2,n3) = 536
(n1,n3) = 589
(n3,n4) = 754

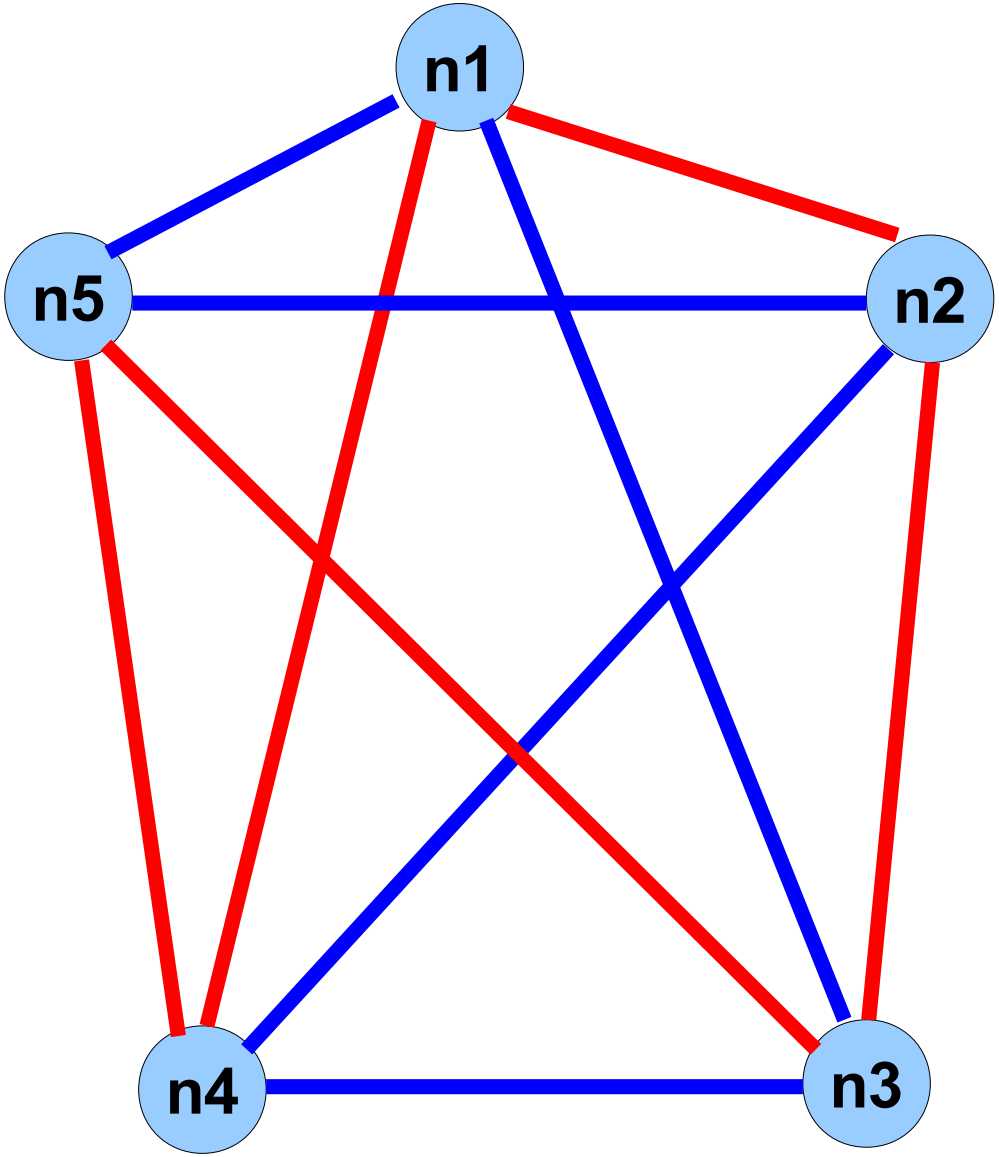
Initial cost = 2569



Edge costs:

(n3,n5) = 91
(n4,n5) = 223
(n1,n4) = 255
(n2,n5) = 288
(n1,n2) = 328
(n1,n5) = 446
(n2,n4) = 492
(n2,n3) = 536
(n1,n3) = 589
(n3,n4) = 754
Next lower cost = 1715
No further local changes will reduce the cost

However, here is a tour with an even smaller cost:



Edge costs:

(n3,n5) = 91
(n4,n5) = 223
(n1,n4) = 255
(n2,n5) = 288
(n1,n2) = 328
(n1,n5) = 446
(n2,n4) = 492
(n2,n3) = 536
(n1,n3) = 589
(n3,n4) = 754
Minimum cost = 1433

This means that, by subsequently reducing the cost of an initial tour by local changes, we may get trapped in a local minimum. Therefore this method will not always find a minimum tour. It may be, however, used as a basis for a heuristic. Surprisingly, it was inspired by a physical process used in engineering which is called annealing. Steel is annealed to improve its quality. The atoms of steel form a crystalline lattice. The lower the internal energy of steel, the more perfect lattice is formed. By cooling down hot steel, we may reduce its energy. However, if this process is too rash, the lattice, assuming a fixed structure, will not continue to be rearranged. Therefore, the cooling should progress slowly being interrupted by short increases of temperature. Exactly this is done by a simulated annealing method used to reduce the cost of a tour.

The method of simulated annealing was first published in 1983 by IBM researchers as a heuristic to be used to find good approximations of optimal solutions to combinatorial problems. We will describe a modification of this method for the travelling salesman problem.

As usual, we start with an initial tour. To each tour T we can define local changes as described above to obtain new "neighbouring" tours $T_{1,} T_{2,} \ldots, T_k$. Each tour T_i either increases or decreases the cost of T. We keep performing local changes. If a change will decrease the cost, it is performed. If it would result in an increase of the tour cost, it may still be performed with a certain probability to prevent the process from being locked in a local minimum. This probability is usually defined as $P = e^{-An}$ where n is the iteration number and A is a suitable constant that may be determined by experimenting. The cost of every tour is matched against the best one found and stored if better. The process is usually stopped after a certain number of iterations have been performed.