Answer 1:

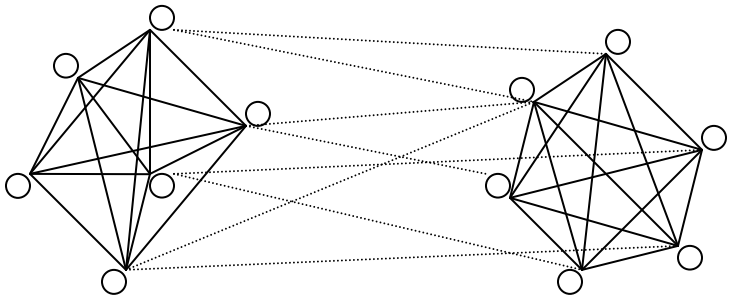
The purpose of generating suitable Benchmark Graphs for this problem is essentially to compare the results of a graph-partitioning heuristic under scrutiny with the partition that has optimum cost (minimum cut-set in this case). For this the optimum solution (minimum cost graph) must be known beforehand.
Generation of suitable benchmark graphs must be in a manner such that the following requirements are met:

- Graph should have reasonable complexity,
- Optimal or near optimal solutions must be known before hand, for comparison.

Graph generation strategies:

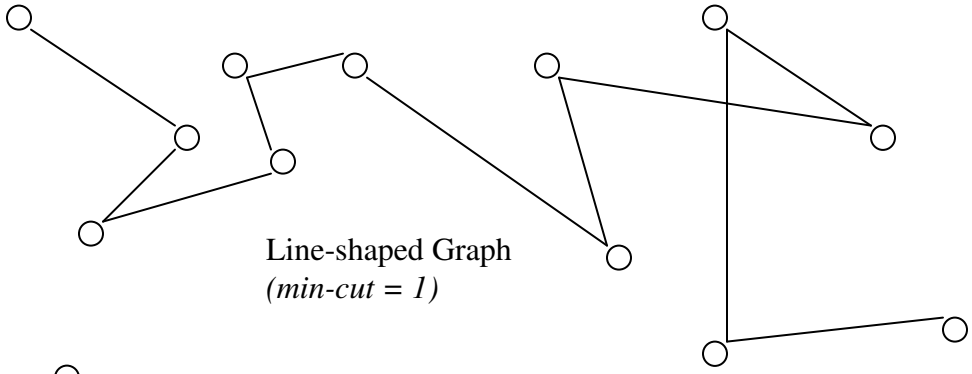
1. **Union of two complete graphs:** Two independent complete graphs with n nodes each (therefore $n-1$ vertices for each node) can be connected together by randomly assigning new vertices between nodes of different graphs, such that number of new inter-graph vertices $0 \leq v < 2n-2$. In this scenario, v will be the minimal cost. The vertices of this joined graph may then be randomly assigned to two equal partitions.

Complete Graph A Complete Graph B

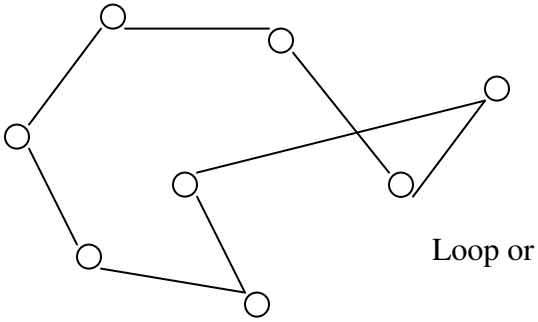


Number of Nodes = $2n$
Number of vertices of each node = $n-1$
Number of Intergraph vertices $v < 2n-2$
Minimum cost partition will cut only the Intergraph vertices.

2. **Use of Regular Geometric Patterns:** Line (Chain-shaped) graphs, Circular-Shaped Graphs etc can be used to test the algorithm:



Line-shaped Graph
(min-cut = 1)



Loop or Ring-shaped Graph: *(min-cut = 2)*

3. **Maximum-cut Graph Bi-Partitioning?** Take a randomly generated *Dense graph* with *2n* nodes, partition arbitrarily into two equal halves (each with *n* nodes), and then apply the following algorithm:
   {
   move the node in partition 1 with the maximum value of ***–D*** where, ***-D = I-E***, where *I* equals internal cost, and *E* equals external cost;

do the same with partition 2…
repeat the above steps until total cut-set reaches maximum…
when no further increase in cut-set is possible, *take the complement of the whole graph.* The maximum cut-set would (should?) then become the minimum cut-set of this new *sparse graph*.

}

(This algorithm was inspired by the K-L Algorithm, among other things, and as such may carry its weaknesses as well…it is based on intuition and paperwork, but has not been tried extensively, or of course, mathematically proven. It may or may not always provide the optimal solution).

Answer 2:

Step 1:

A={1, 2, 3} B = {4, 5, 6},
Set A' = A, and B' = B.

Step 2: Calculate D Values:

$D_i = E_i - I_i$,

$D_1 = 0 - 1 = -1$ $D_2 = 1 - 2 = -1$
$D_3 = 0 - 1 = -1$ $D_4 = 1 - 2 = -1$
$D_5 = 0 - 2 = -2$ $D_6 = 0 - 2 = -2$

Step 3: Calculate Gain for each Pair:

$g_{14} = -2$ $g_{15} = -3$ $g_{16} = -3$ $g_{24} = -1-1-2 = -4$ $g_{25} = -1-2 = -3$
$g_{26} = -1-2 = -3$ $g_{34} = -1-1 = -2$ $g_{35} = -1-2 = -3$ $g_{36} = -1-2 = -3$
g_{14} and g_{34} are minimum gains… we select 1,4 for swapping.

Step 4: Update D Values:

A' = {2, 3} B' = {5, 6}

$D'_2 = -1+2(1)-2(1) = -1$
$D'_3 = D_3 = -1$
$D'_5 = -2+2(1)-2(0) = 0$
$D'_6 = -2+2(1)-2(0) = 0$

Step 3: Calculate Gain for each Pair:

$g_{25} = -1+0-0 = -1$
$g_{26} = -1+0-0 = -1$
$g_{35} = -1+0-0 = -1$
$g_{36} = -1+0-0 = -1$

All Gains are equal… we randomly select 2, 6 for swapping.

Step 4: Update D Values:

A' = {3} B' = {5}

$D'_3 = -1+2(1)-0 = 1$
$D'_5 = 0+2(1)-0 = 2$

Step 3: Calculate Gain for each Pair:

$g_{35} = 1 + 2 – 0 = 3$

3, 5 are swapped.

Step 4: Update D Values:

A’ = { } B’ = { }

Since A’ an B’ are Empty, go to step 5

Step 5: Find maximum G:

$g_1 = -2$, $g_2 = -1$, $g_3 = 3$

$G_1 = g_1 = – 2$
$G_2 = g_1 + g_2 = – 3$
$G_3 = g_1 + g_2 + g_3 = 0$

Therefore, G_3 maximizes G.

So final solution:

A = {4, 5, 6}, B = {1, 2, 3}

Answer 3:

Partitioning using Simulated Annealing:

The values of T0 (initial temperature), Alpha, and BETA were modified and charts were generated for each set of values (a total of 17 charts). It has been noted that the effect of varying BETA is negligible for high values of T0, and a small problem size that involves only a few hundred iterations. Due to the small problem size, a minimal solution is usually attained randomly much before the algorithm terminates, and so variations in BETA have minimal effect.

An increase in the value of initial temperature T0 generally resulted in the minimal solution being reached much later. This is probably due to the increased acceptance of bad moves. In one instance, with a low value of BETA (= 1) and a high value of T0 (= 14), the minimal solution was never achieved. It can be concluded that a relatively high value of T0 may require longer runtimes (i.e. a high BETA as well).

Also visible from the graphs is that with a high value of T0 (= 14), there is no visible change in the initially high rate of acceptance of bad values with time… and thus the algorithm approximates a random walk. A similar situation occurs with low values of T0 (= 4), as rate of acceptance of bad moves remains low and unchanged throughout the run, approximating greedy behavior. Although due to the problem's simplicity, a solution is often achieved.

For T0 = 7, a constant reduction in the acceptance rate of bad moves is visible in the charts. Therefore, it is recommended, that for larger problem instances with ***similar weight assignments to nets***, a value of T0 close to 7 will be appropriate. For lesser values of ALPHA (approx .87), greater values of BETA will give a stronger move-acceptance gradient.

Therefore, in this particular implementation, and for this type of problem set, it can be concluded that approximately, T=7, Alpha = 0.85, and Beta > 1.2 will yield the best approximation of the 'annealing' process. (Please see the last three charts in the Excel File)

Code for Partitioning using Simulated Annealing (Results are in Attached Excel File):

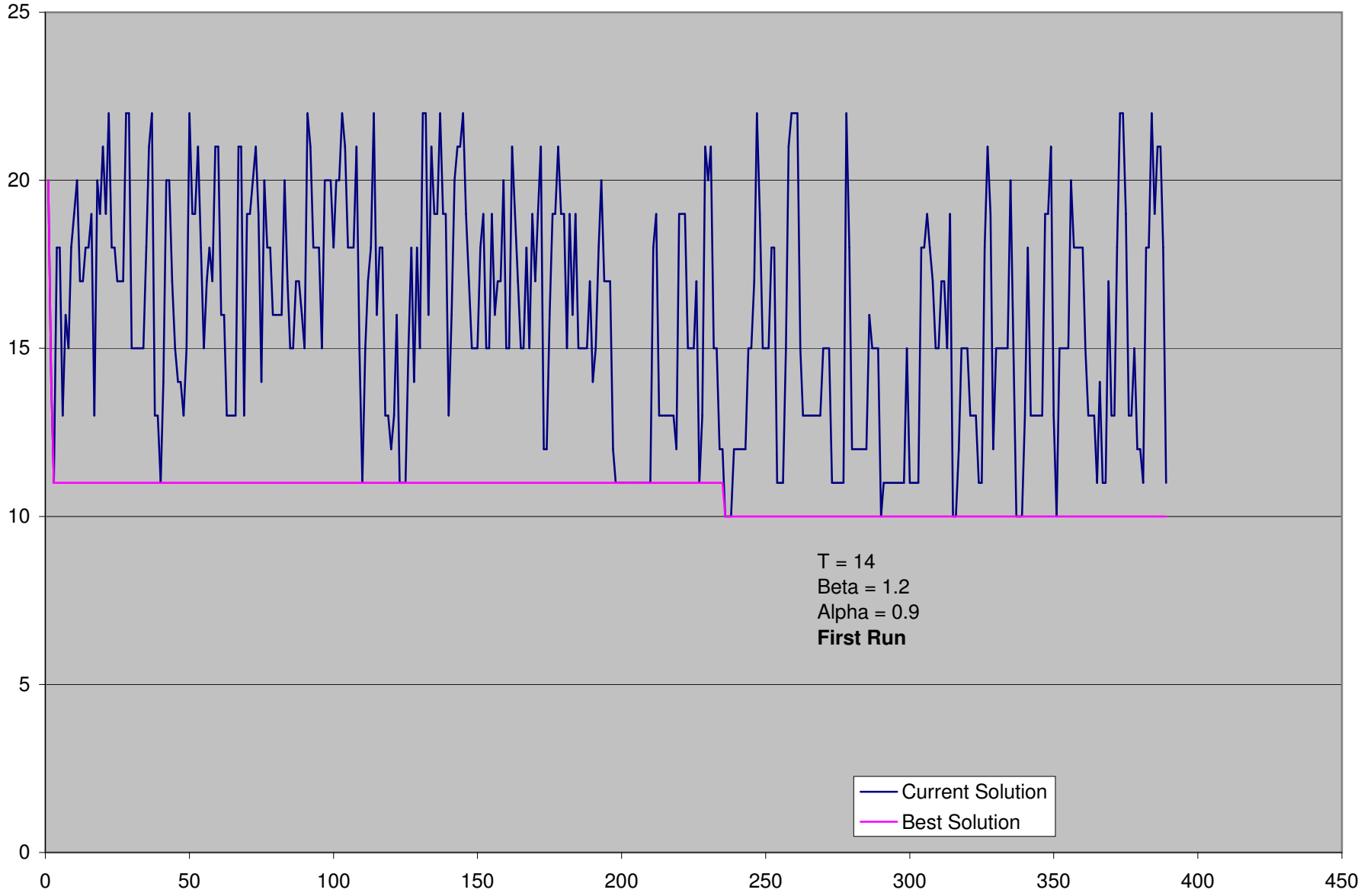
```
/*********************************  Homework Asignment 2: Solving the Graph      *********************************/
/*********************************       Partitioning Problem using              *********************************/
/*********************************            Simulated Annealing                *********************************/
#include "stdlib.h"
#include "stdio.h"
#include "conio.h"
#include "math.h"
#include "time.h"

////// Parameter Definitions
#define T0 14
#define M 10
#define ALPHA 0.9
#define BETA 1.2
#define TFINAL (T0*3/10)
#define FILENAME 'Results'
////////////////////////////


////// Function Prototypes
void Metropolis(int* Sol_A, int* Sol_B,  double Temp, int Cycle_Time);
int      Cost(int* Sol_A, int* Sol_B);
//////////////////////////


////// Global Variabes
/* initial solution*/
int Part_A[] = { 1, 2, 3, 4, 5};
int Part_B[] = { 6, 7, 8, 9, 10};
/* Best Solution */
int Best_A[5];
int Best_B[5];
int Best_Cost;
/* netlist and weights */
int Netlist[10][4] = {
                                    { 1,  2,  4,  5},
                                    { 2,  3,  5, -1},
                                    { 3,  6, 10,  4},
                                    { 4,  8,  3,  7},
                                    { 5,  7,  1,  6},
                                    { 6,  4,  7,  2},
                                    { 7,  9,  5, -1},
                                    { 8,  2, -1, -1},
                                    { 9, 10,  5, -1},
                                    {10,  5, -1, -1} };
int Netweight[] = { 1, 1, 2, 1, 3, 3, 2, 3, 2, 4};
//////////////////////


/********      Start of Main Function: Simulated Annealing   ********/
```

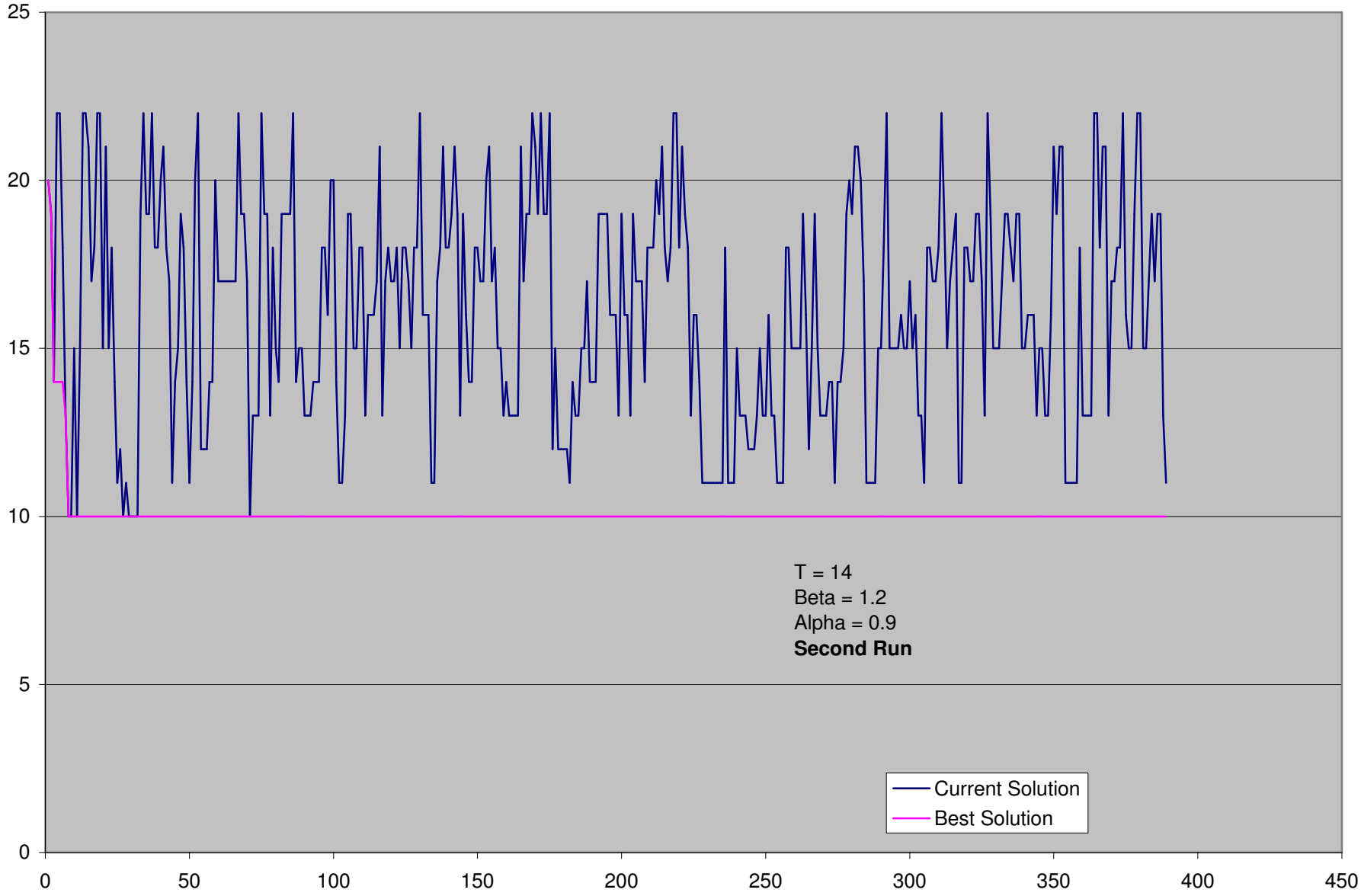
```
void main (void)
{
	//declare variables
	int Time = 0, count, Current_Cost;
	int Sol_A[6], Sol_B[6];
	double Temp = T0, Cycle_Time = M;
	time_t t;
	FILE *Result;

	//copy initial solution to working space
	for (count = 0; count <6; count++)
	{
		Sol_A[count] = Part_A[count];
		Sol_B[count] = Part_B[count];
	}

	// set initial values
	count = 0;
	Current_Cost = Cost(Sol_A, Sol_B);
	Best_Cost       = Current_Cost;
	srand ((unsigned) time(&t));

	//start Simulated Annealing Loop
	do
	{
		Metropolis(Sol_A, Sol_B, Temp, (int) Cycle_Time);
		Temp           = ALPHA * Temp;
		Cycle_Time     = BETA * Cycle_Time;
		//Time         = Time + Cycle_Time;
		count++;
	}while (Temp > TFINAL);

	//print the best solution found and exit.
	Result = fopen("c:\\Results.txt", "a+");
	printf("\nThe best solution found FINALLY has the following partitions:\n  A\t  B");
	fprintf(Result, "\nThe best solution found FINALLY has the following partitions:\n  A\t  B");
	for (count = 0; count < 5; count++)
	{
		printf("\n %d\t %d", Best_A[count], Best_B[count]);
		fprintf(Result, "\n %d\t %d", Best_A[count], Best_B[count]);
	}
	printf("\n\n And the Cost of the Best Solution is: %d", Best_Cost);
	fprintf(Result, "\n\n And the Cost of the Best Solution is: %d", Best_Cost);
	getch();
	fclose (Result);
	exit(0);
}


/********      Start of Metropolis Function   ********/
void Metropolis(int* Sol_A, int* Sol_B, double Temp, int Cycle_Time)
{
	int cost1 = 0, cost2 =0, i=0, j=0, temporary, count=0, count1=0;
	double randnum, Boltzmann=0;
```

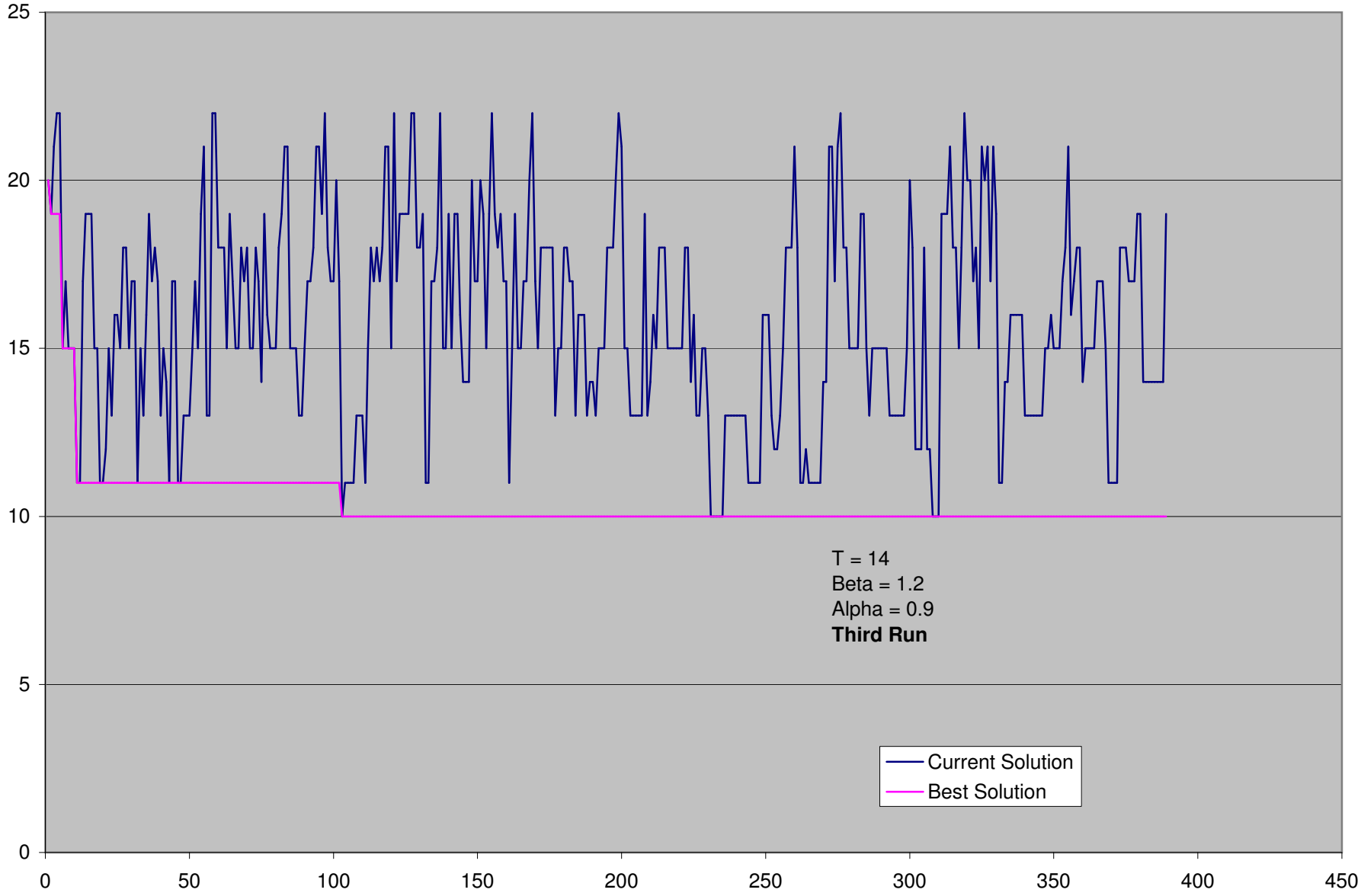
```
FILE *Result;

// open file for writing
Result = fopen("c:\\Results.txt", "a+");
/*fprintf(Result, "\nBest\tCost(O)\tCost(N)\tRandom\tBltzmn\tTemp\tCount");
printf("\nBest\tCost(O)\tCost(N)\tRandom\tBltzmn\tTemp\tCount");
//*/
//main Metropolis Loop
while (Cycle_Time > 0)
{
        //get cost
        cost1 = Cost(Sol_A, Sol_B);

        //perturb
        i = rand() % 5;
        j = rand() % 5;
        temporary = Sol_A[i];
        Sol_A[i]  = Sol_B[j];
        Sol_B[j]  = temporary;

        //get cost after perturb
        cost2 = Cost(Sol_A, Sol_B);

        //generate Random Number for Bad Solution Acceptance Criteria
        randnum = (double) (rand() % 100);
        randnum = randnum / 100;

        //Rejection based on SA Criteria
        Boltzmann = exp((-1)*(cost2-cost1)/Temp);
        //print Parameters of iteration to file as well as to output
        fprintf(Result, "\n%d\t%d\t%d\t%f\t%f\t%f\t%d", Best_Cost, cost1, cost2, randnum, Boltzmann, Temp, count1+1);
        printf("\n%d\t%d\t%d\t%f\t%f\t%f\t%d", Best_Cost, cost1, cost2, randnum, Boltzmann, Temp, count1+1);

        if( (cost2 >= cost1) && (randnum > Boltzmann) ) /* ie in both cases, if soln is unacceptable, do this*/
        {
                temporary = Sol_A[i];
                Sol_A[i]  = Sol_B[j];
                Sol_B[j]  = temporary;
                printf("\t REJECTED");
                fprintf(Result, "\t REJECTED");
        }
        else
        {
                if (cost2 < cost1)
                {
                        printf("\t ACCEPTED For Better Cost");
                        fprintf(Result, "\t ACCEPTED For Better Cost");
                }
                else
                {
                        printf("\t ACCEPTED Randomly");
                        fprintf(Result, "\t ACCEPTED Randomly");
                }
                cost1 = cost2;
```

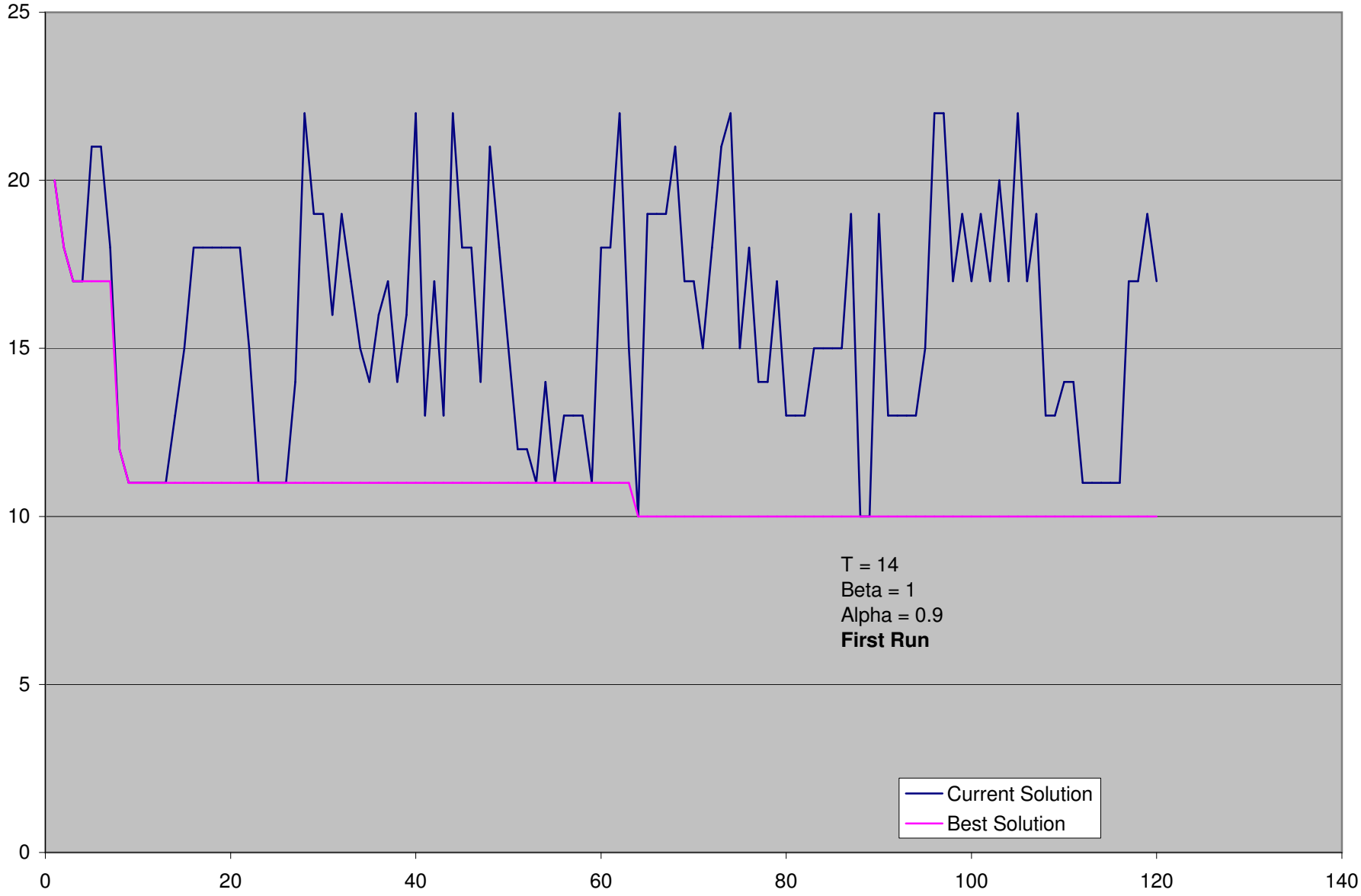
```
                }

                // If best cost found, then do this:
                if(Best_Cost > cost1)
                {
                        Best_Cost = cost1;
                        fprintf(Result, "\t ALSO BEST COST SO FAR");
                        printf("\t ALSO BEST COST SO FAR");
                        for (count = 0; count <6; count++)
                        {
                                Best_A[count] = Sol_A[count];
                                Best_B[count] = Sol_B[count];
                        }
                }
                Cycle_Time--;
                count1++;
        }
        //printf("\nBest Cost: \t%d\n", Best_Cost);
        //printf("\nCurrent Cost: \t%d\n", cost1);
        //fprintf(Result, "\nBest Cost: \t%d\n", Best_Cost);
        //fprintf(Result, "\nCurrent Cost: \t%d\n", cost1);
        //fprintf(Result, "\n Best Cost/Current Cost so far: %d\t%d\n", Best_Cost, cost1);
        fclose(Result);
}



/********      Start of Cost Calculation Function   ********/
int      Cost(int* Sol_A, int* Sol_B)
{
        int i=0, j=0, k=0, Cost=0, NetOfAIsCut = 0, NetOfBIsCut = 0;

        for (i=0; i<=4; i++)
        {
                NetOfAIsCut = 0;
                for (j=0; j<=4; j++)
                {
                        for (k=0; k<=2; k++)
                        {
                                if (Netlist[Sol_A[i]-1][k+1] == Sol_B[j])
                                {
                                        if(NetOfAIsCut != 0)
                                                break;
                                        Cost += Netweight[Sol_A[i]-1];
                                        NetOfAIsCut = 1;
                                }
                        }
                }
        }


        for (i=0; i<=4; i++)
        {
                NetOfBIsCut = 0;
```

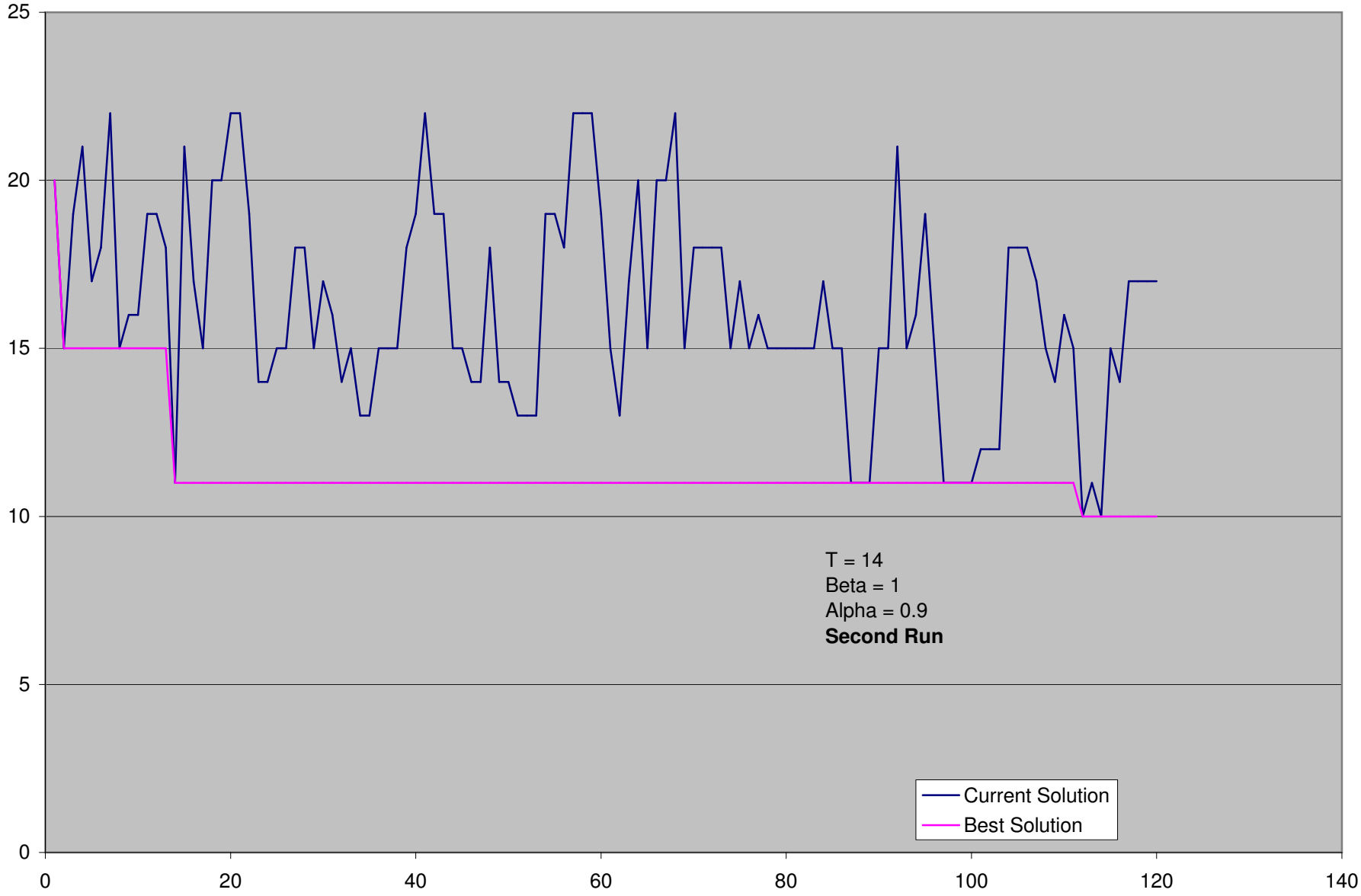
```
                for (j=0; j<=4; j++)
                {
                        for (k=0; k<=2; k++)
                        {
                                if (Netlist[Sol_B[i]-1][k+1] == Sol_A[j])
                                {
                                        if(NetOfBIsCut != 0)
                                                break;
                                        Cost += Netweight[Sol_B[i]-1];
                                        NetOfBIsCut = 1;
                                }
                        }
                }
        }
        return Cost;
}
```

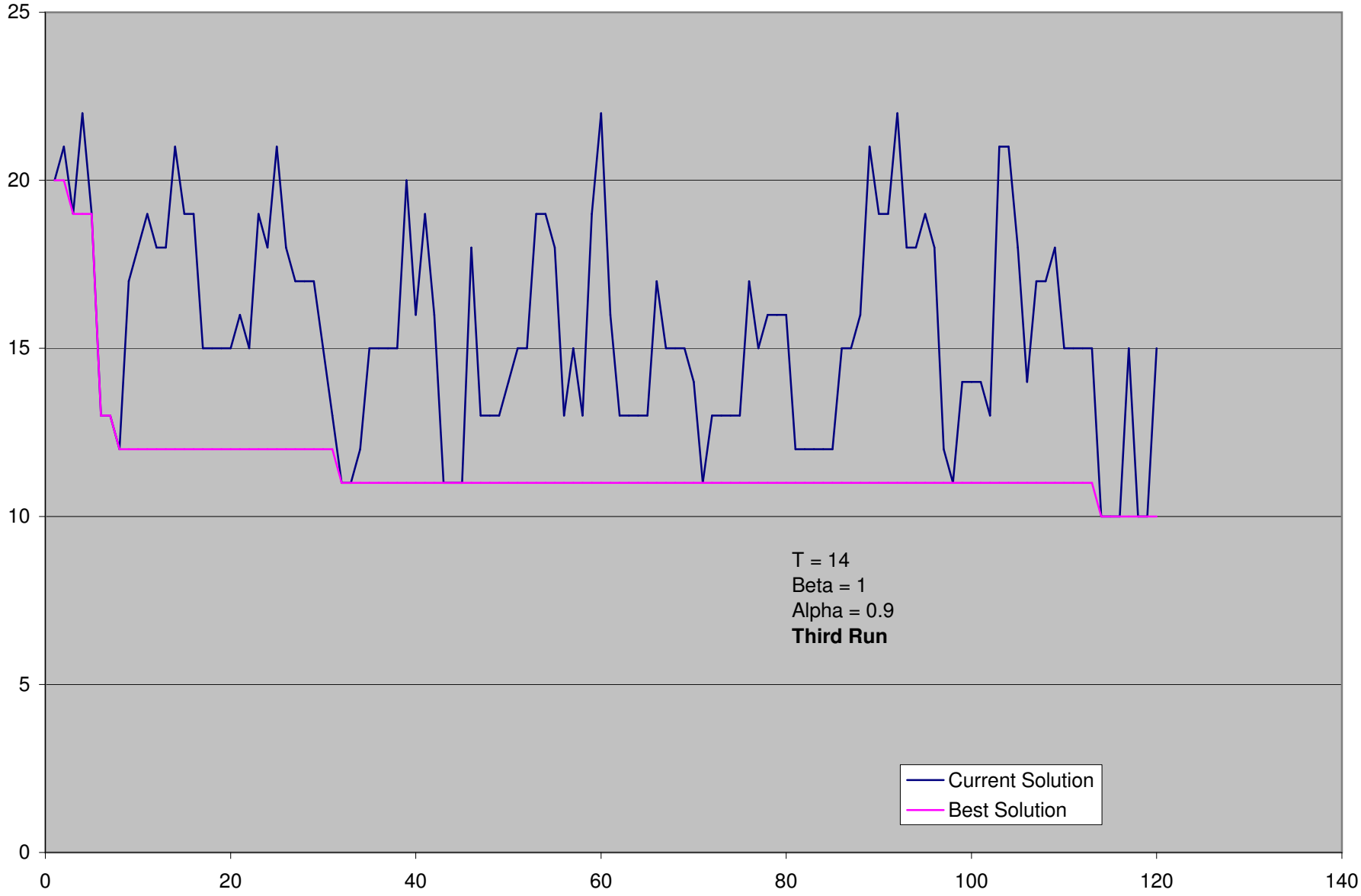

25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 14
Beta = 1.2
Alpha = 0.9
First Run
Current Solution
Best Solution

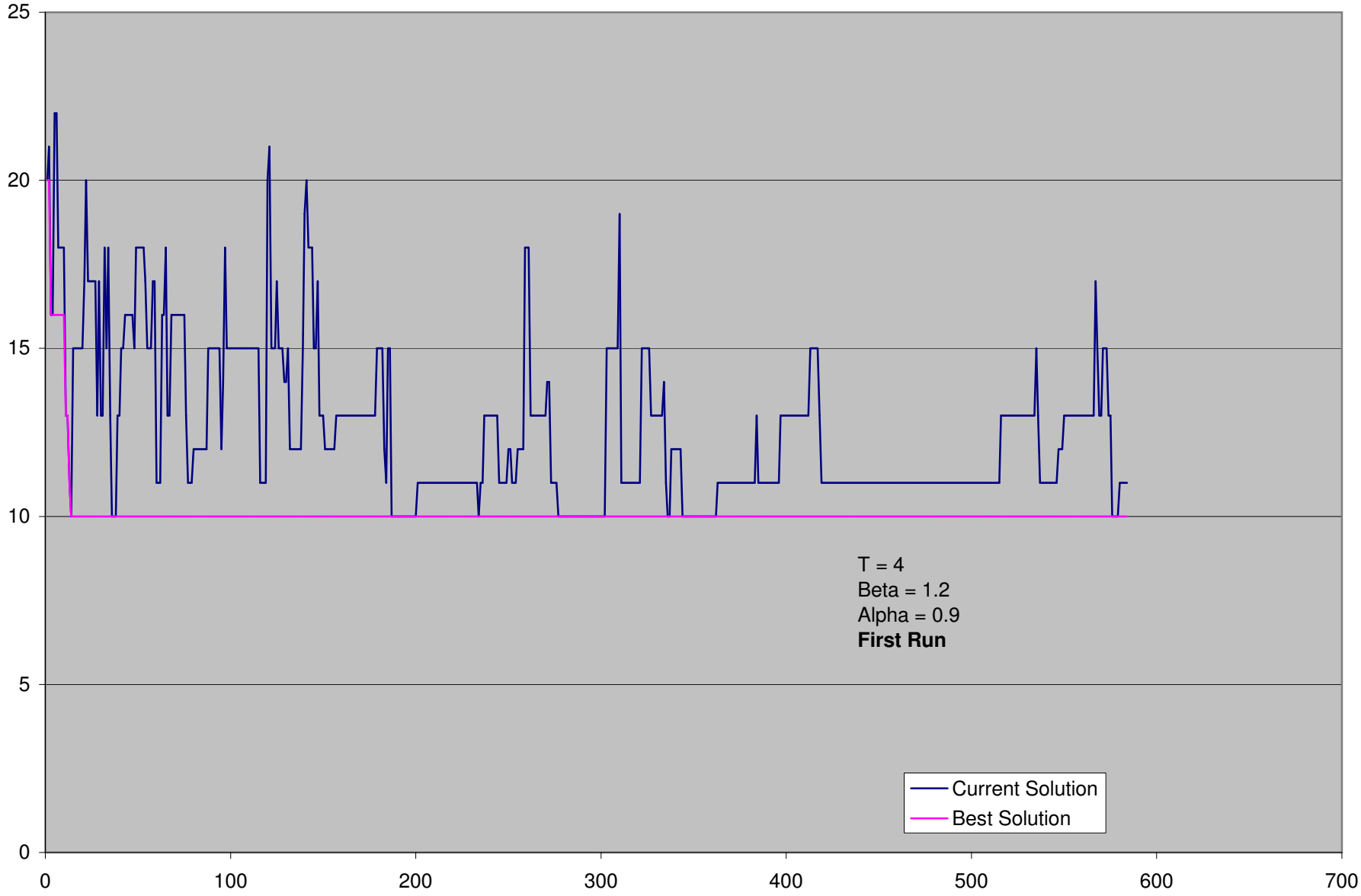

25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 14
Beta = 1.2
Alpha = 0.9
Second Run
Current Solution
Best Solution

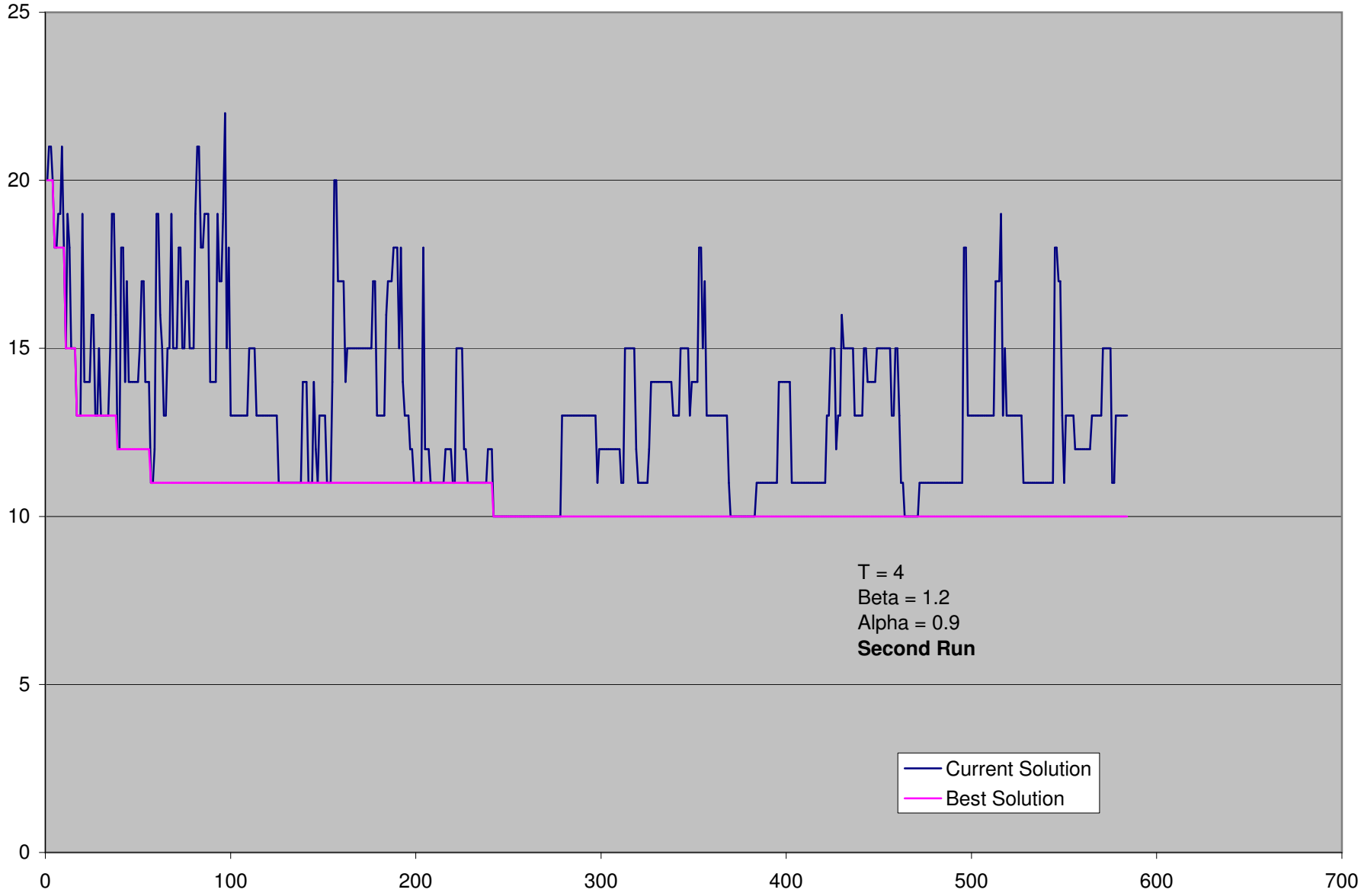

25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 14
Beta = 1.2
Alpha = 0.9
Third Run
Current Solution
Best Solution

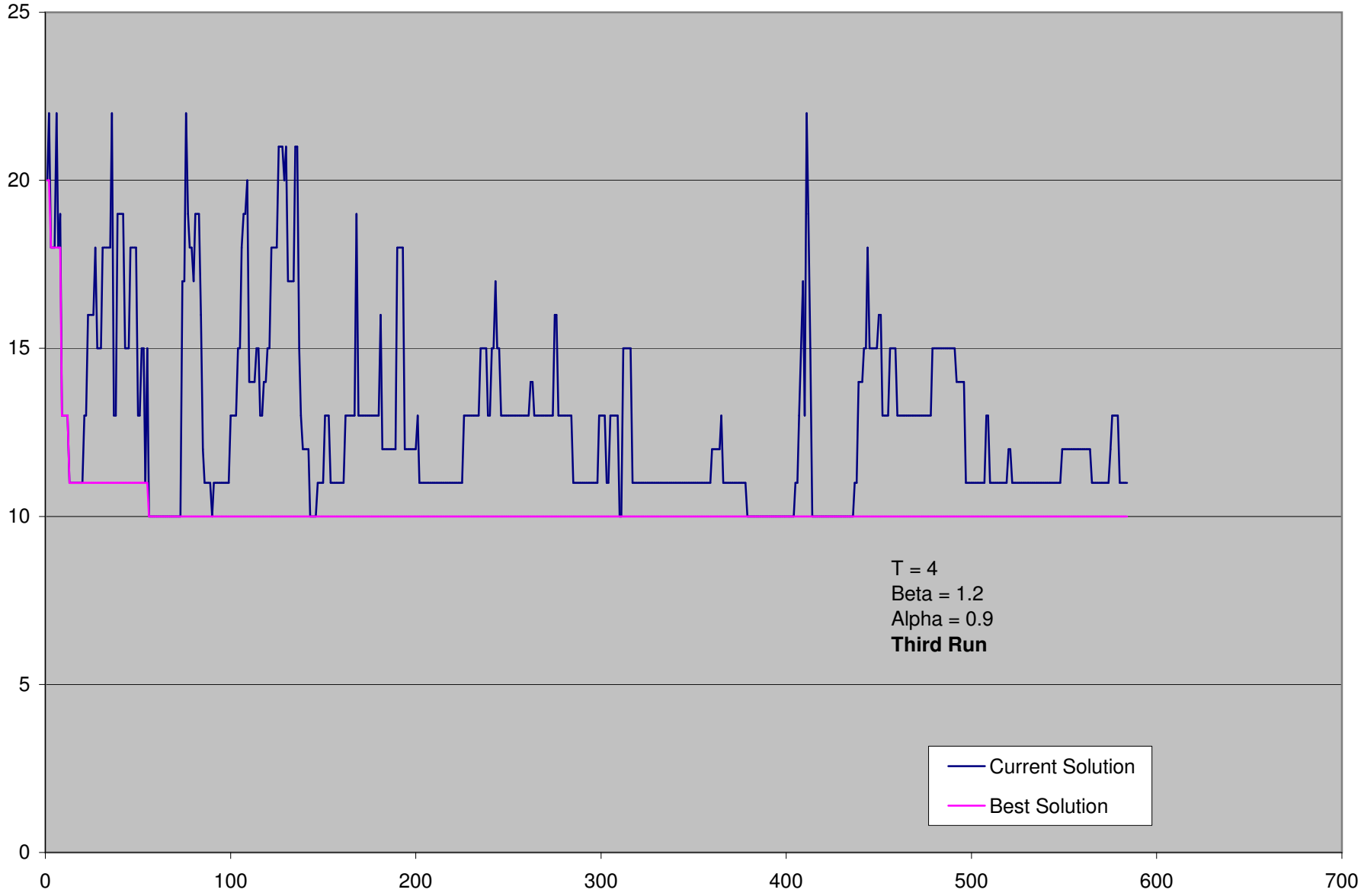

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 14
Beta = 1
Alpha = 0.9
First Run
Current Solution
Best Solution

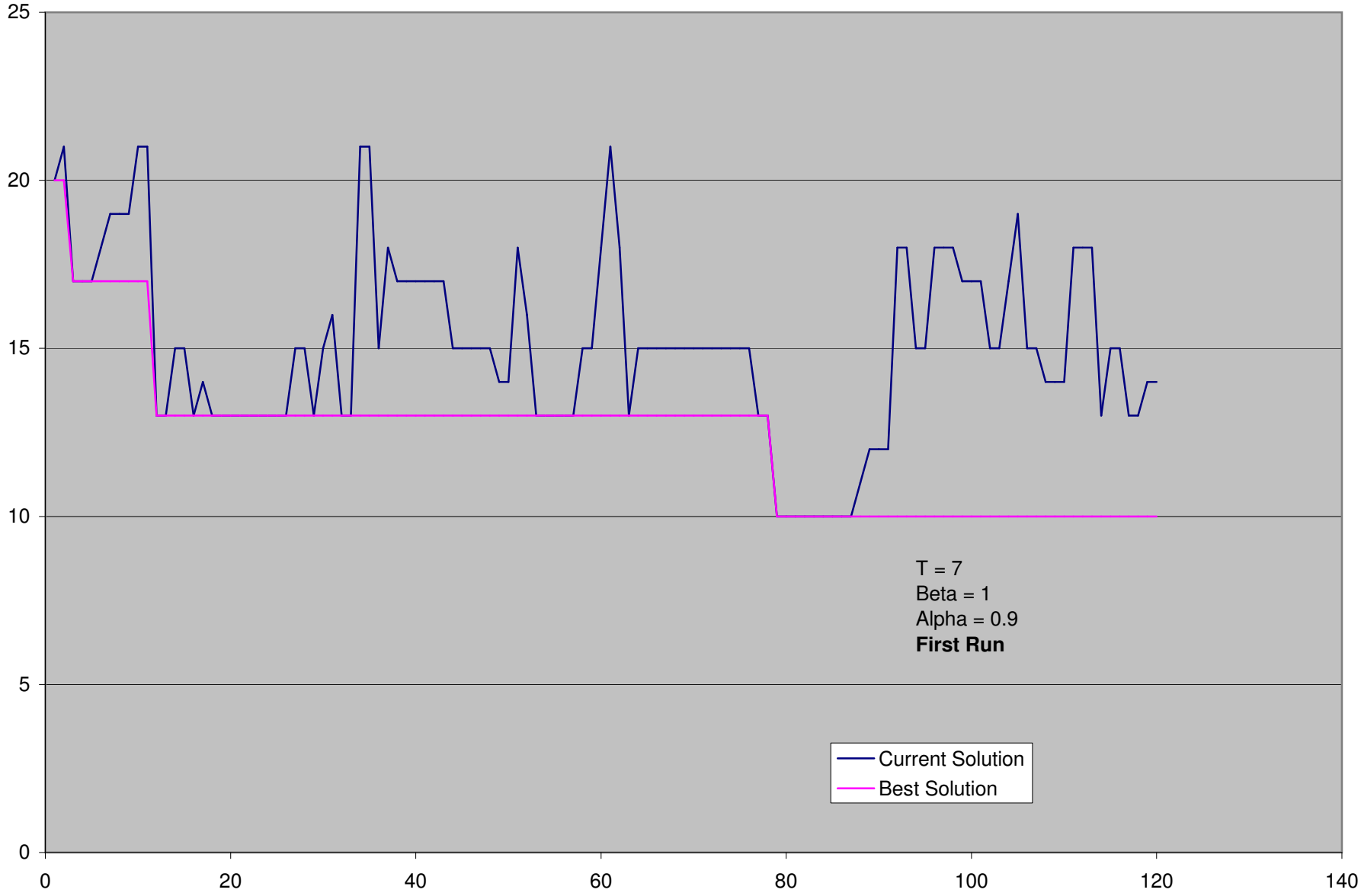

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 14
Beta = 1
Alpha = 0.9
Second Run
Current Solution
Best Solution

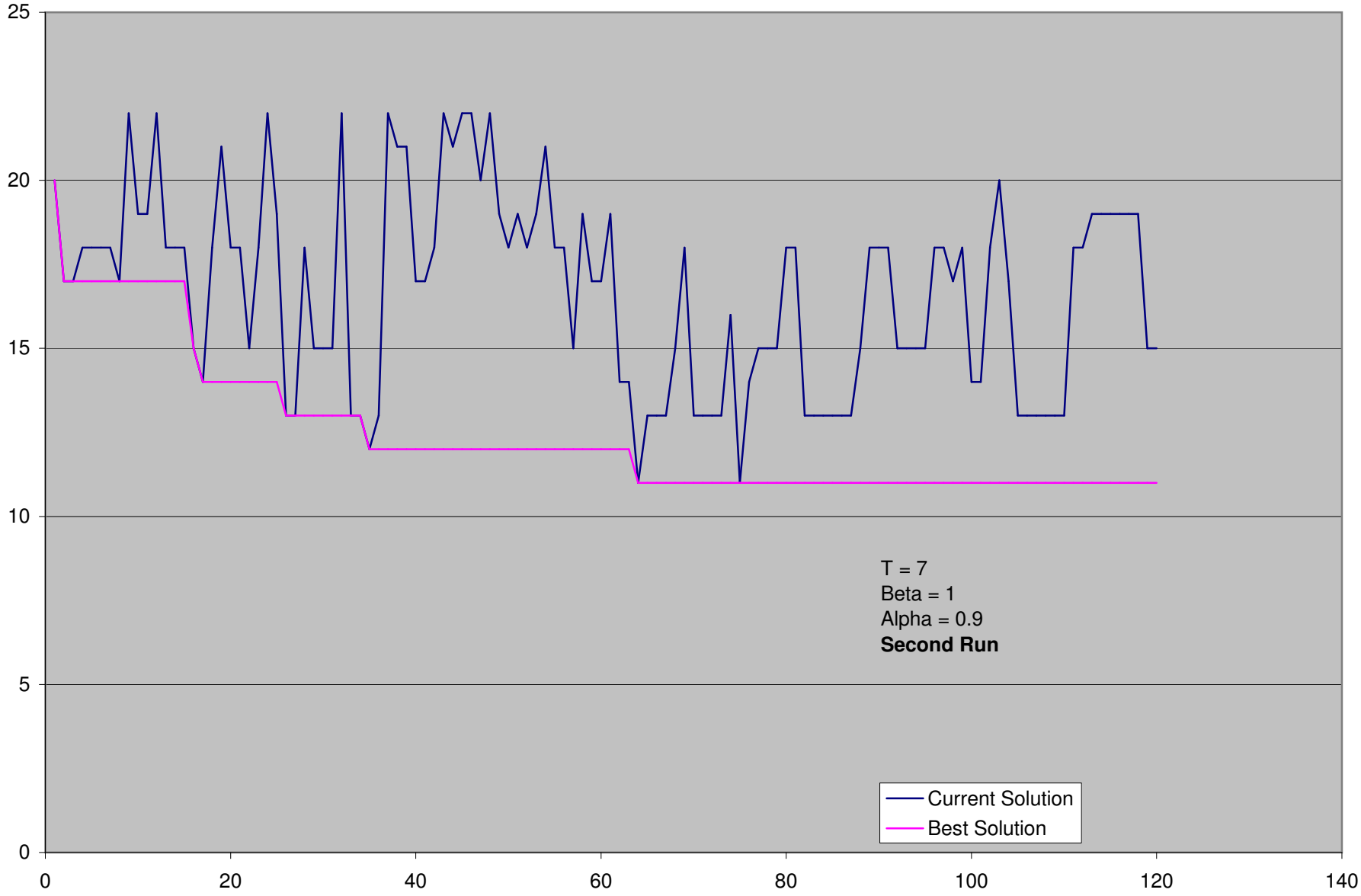

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 14
Beta = 1
Alpha = 0.9
Third Run
Current Solution
Best Solution

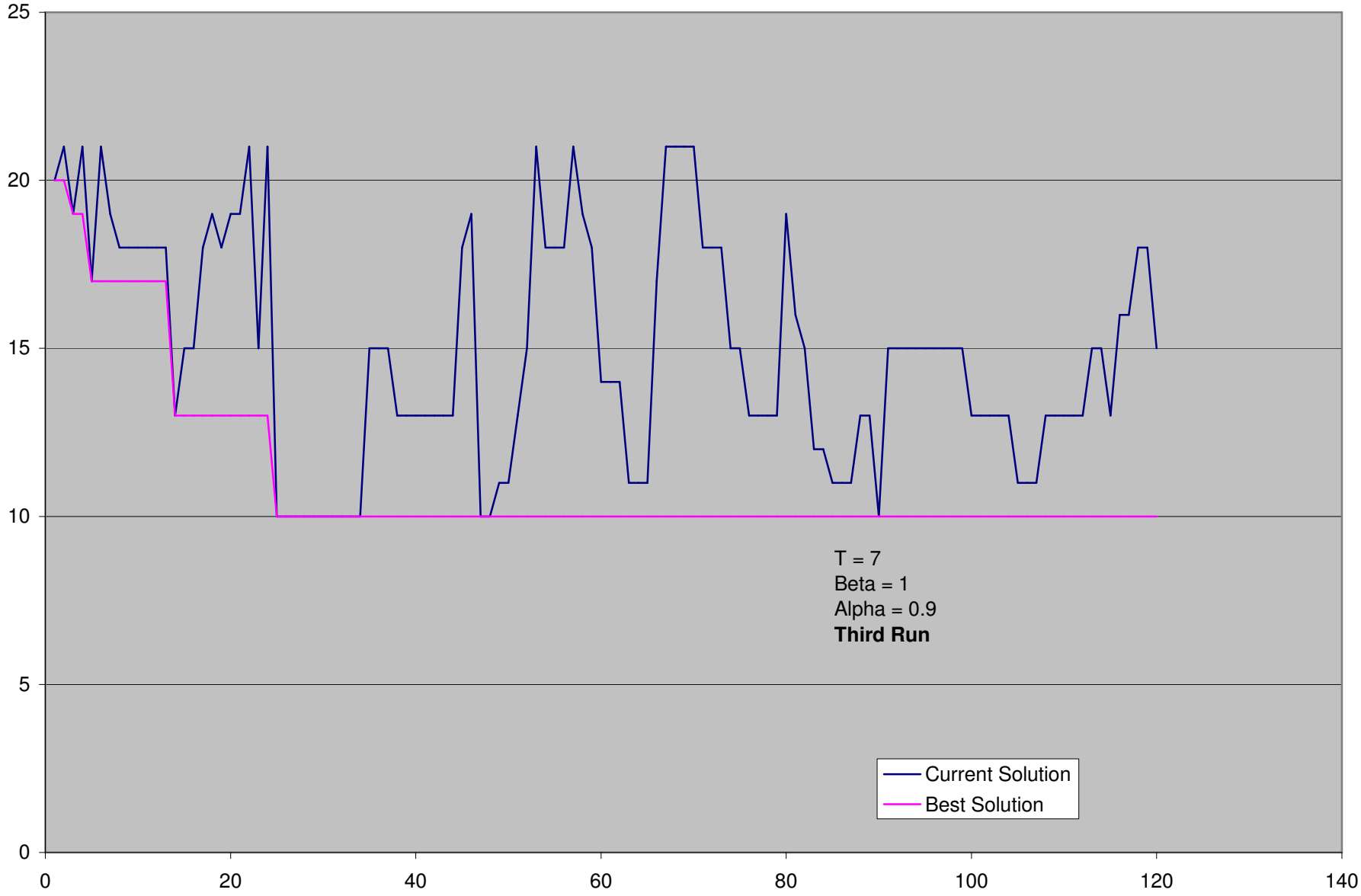

25
20
15
10
5
0
0
100
200
300
400
500
600
700
T = 4
Beta = 1.2
Alpha = 0.9
First Run
Current Solution
Best Solution

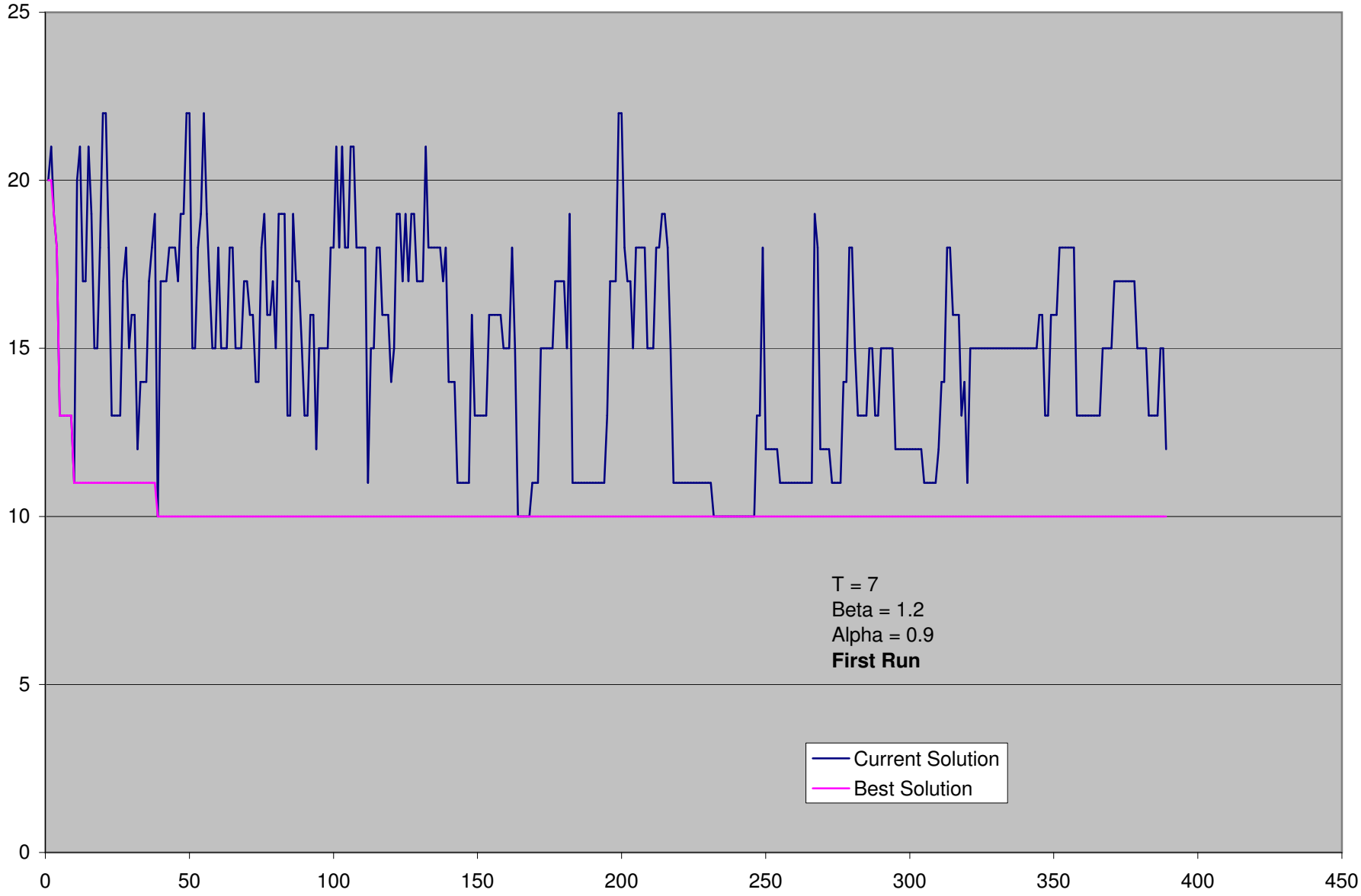

25
20
15
10
5
0
0
100
200
300
400
500
600
700
T = 4
Beta = 1.2
Alpha = 0.9
Second Run
Current Solution
Best Solution

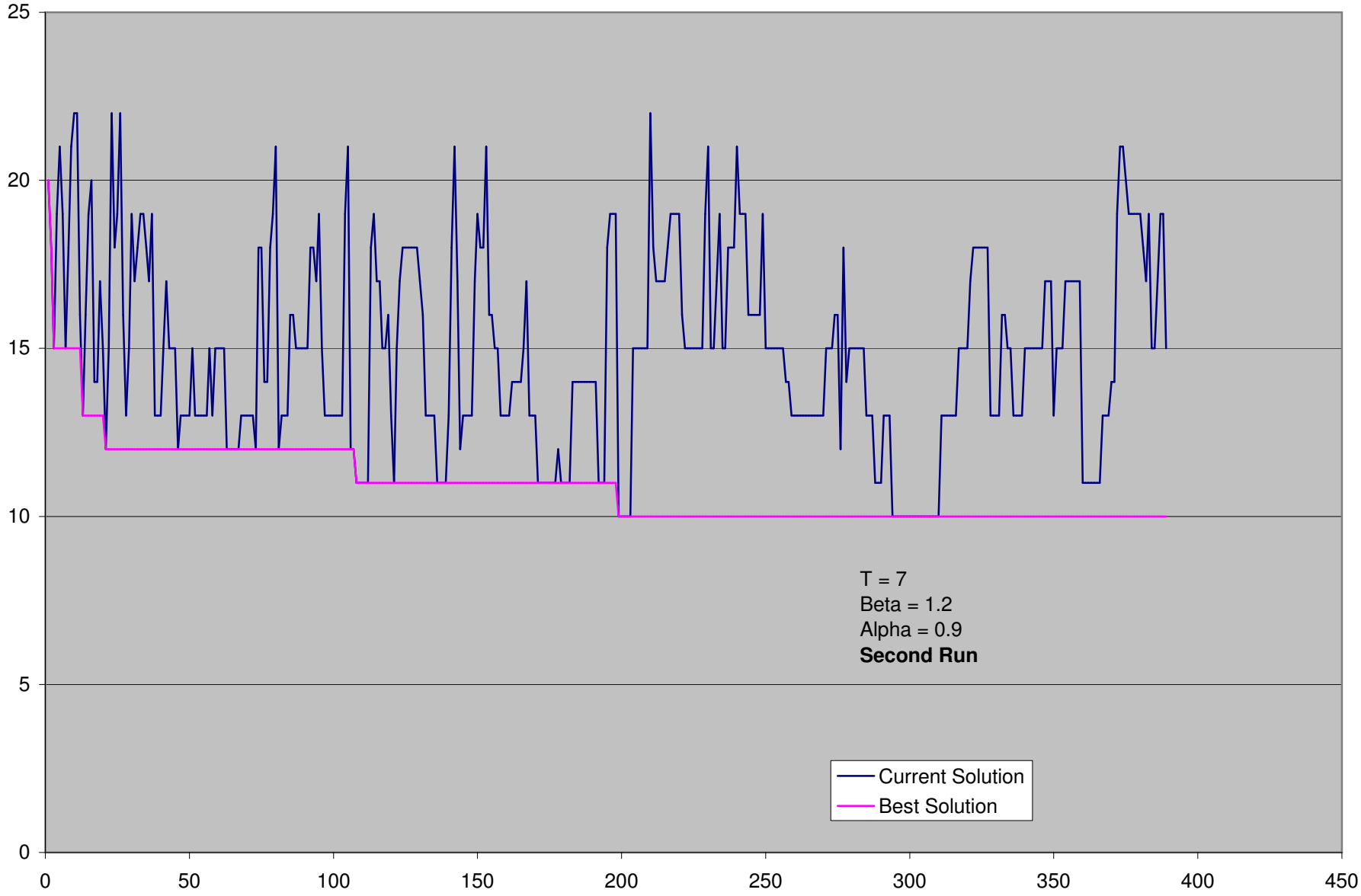

25
20
15
10
5
0
0
100
200
300
400
500
600
700
T = 4
Beta = 1.2
Alpha = 0.9
Third Run
Current Solution
Best Solution

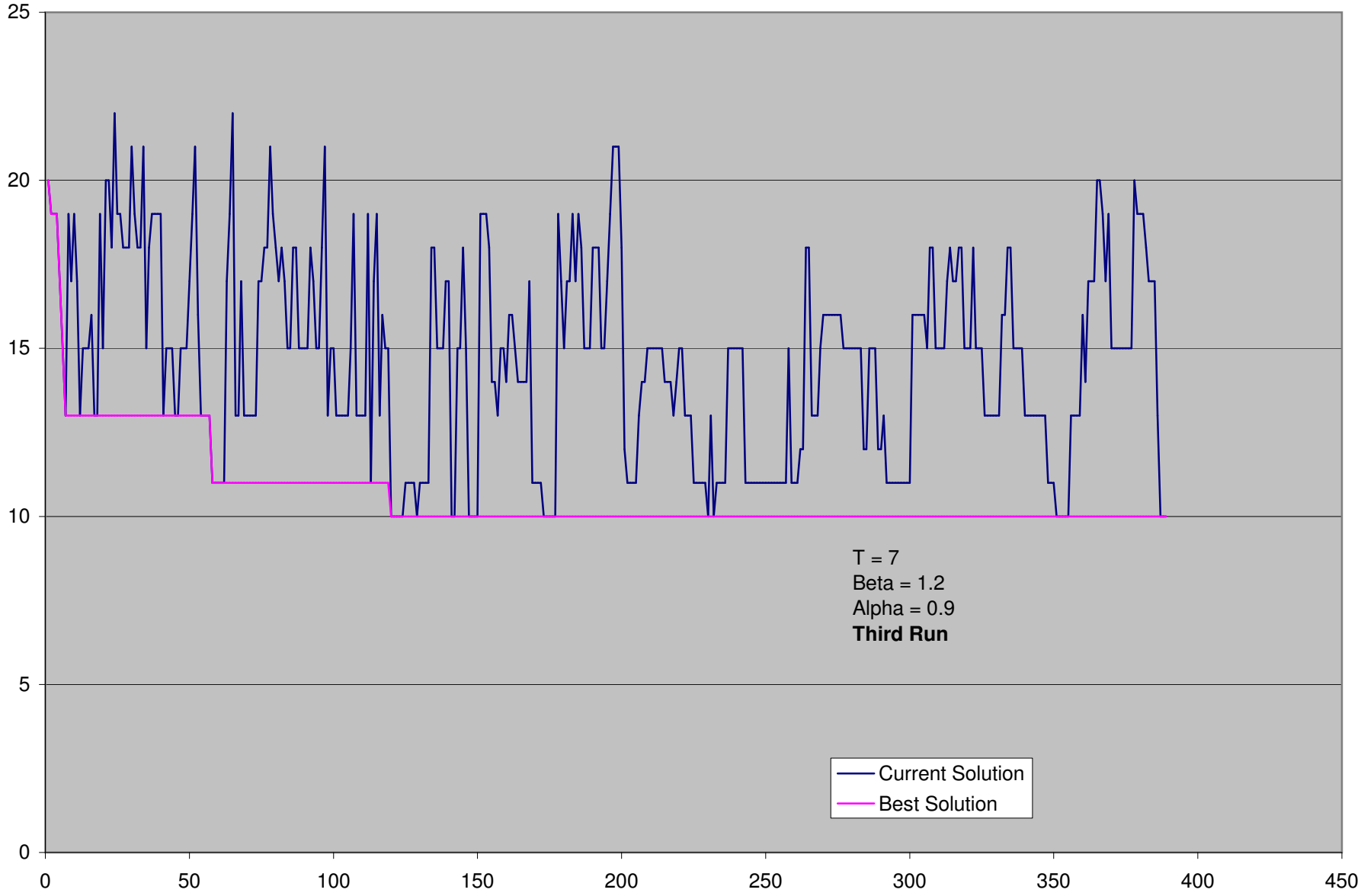

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 7
Beta = 1
Alpha = 0.9
First Run
Current Solution
Best Solution

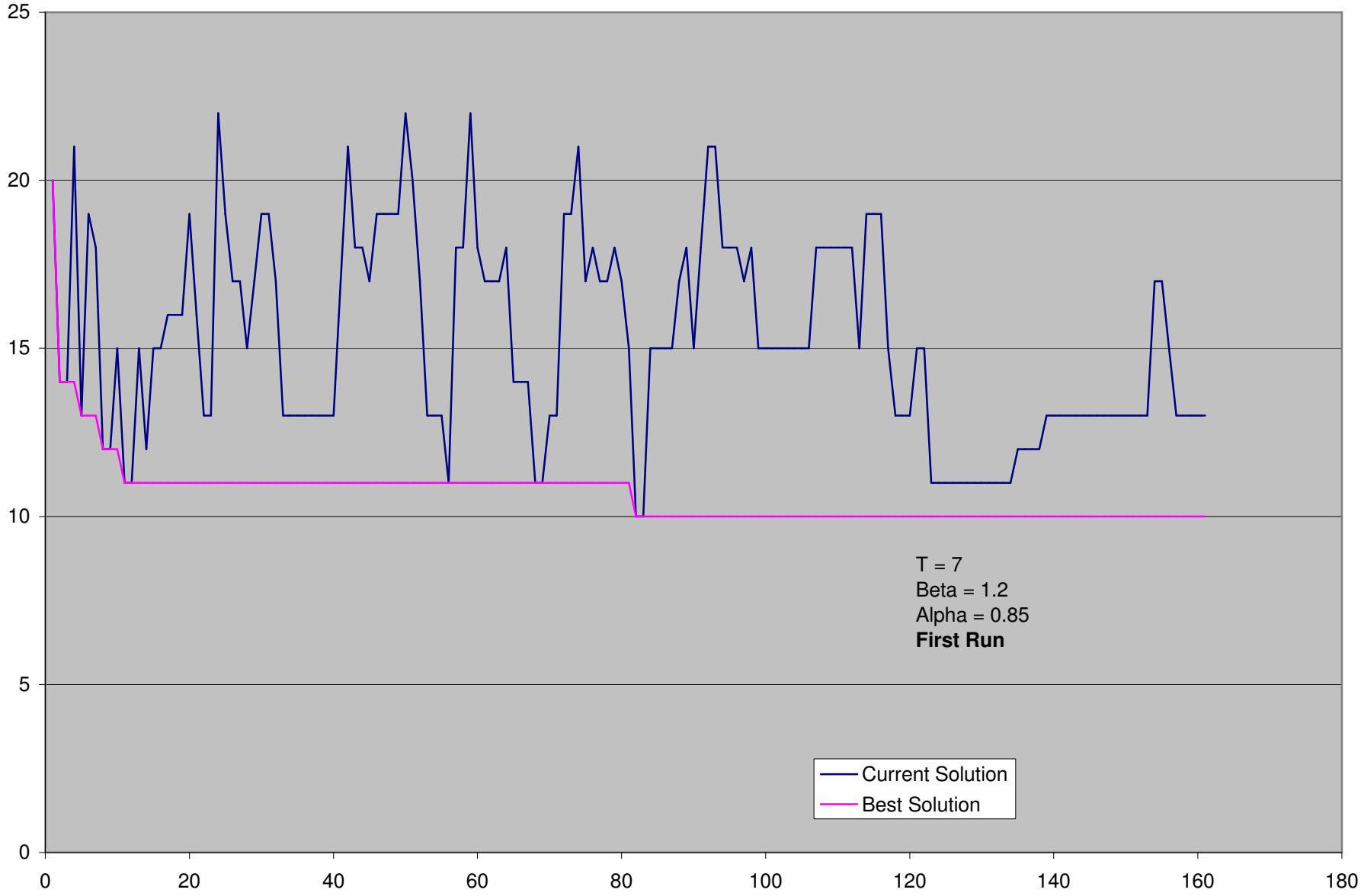

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 7
Beta = 1
Alpha = 0.9
Second Run
Current Solution
Best Solution

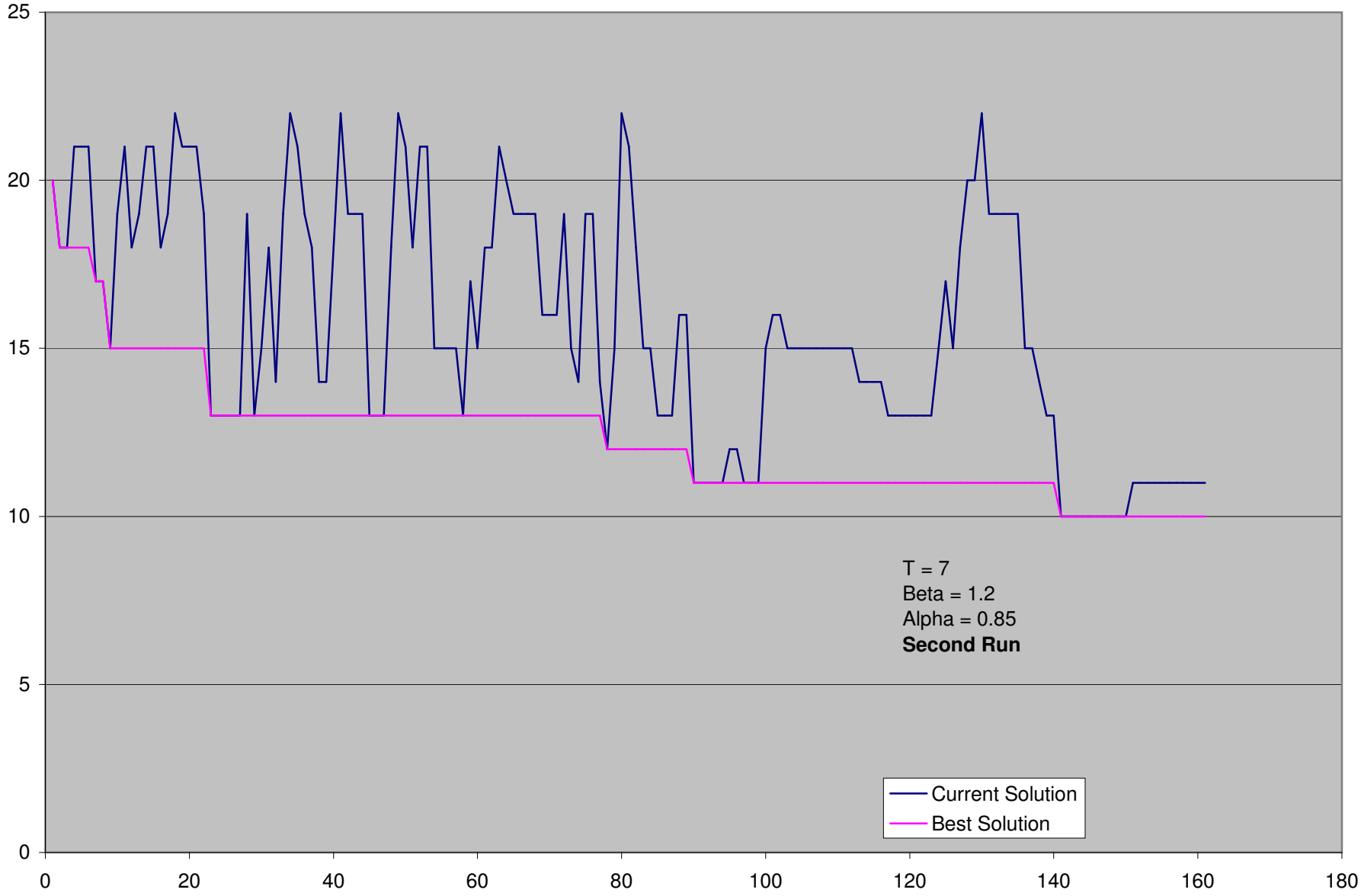

25
20
15
10
5
0
0
20
40
60
80
100
120
140
T = 7
Beta = 1
Alpha = 0.9
Third Run
Current Solution
Best Solution

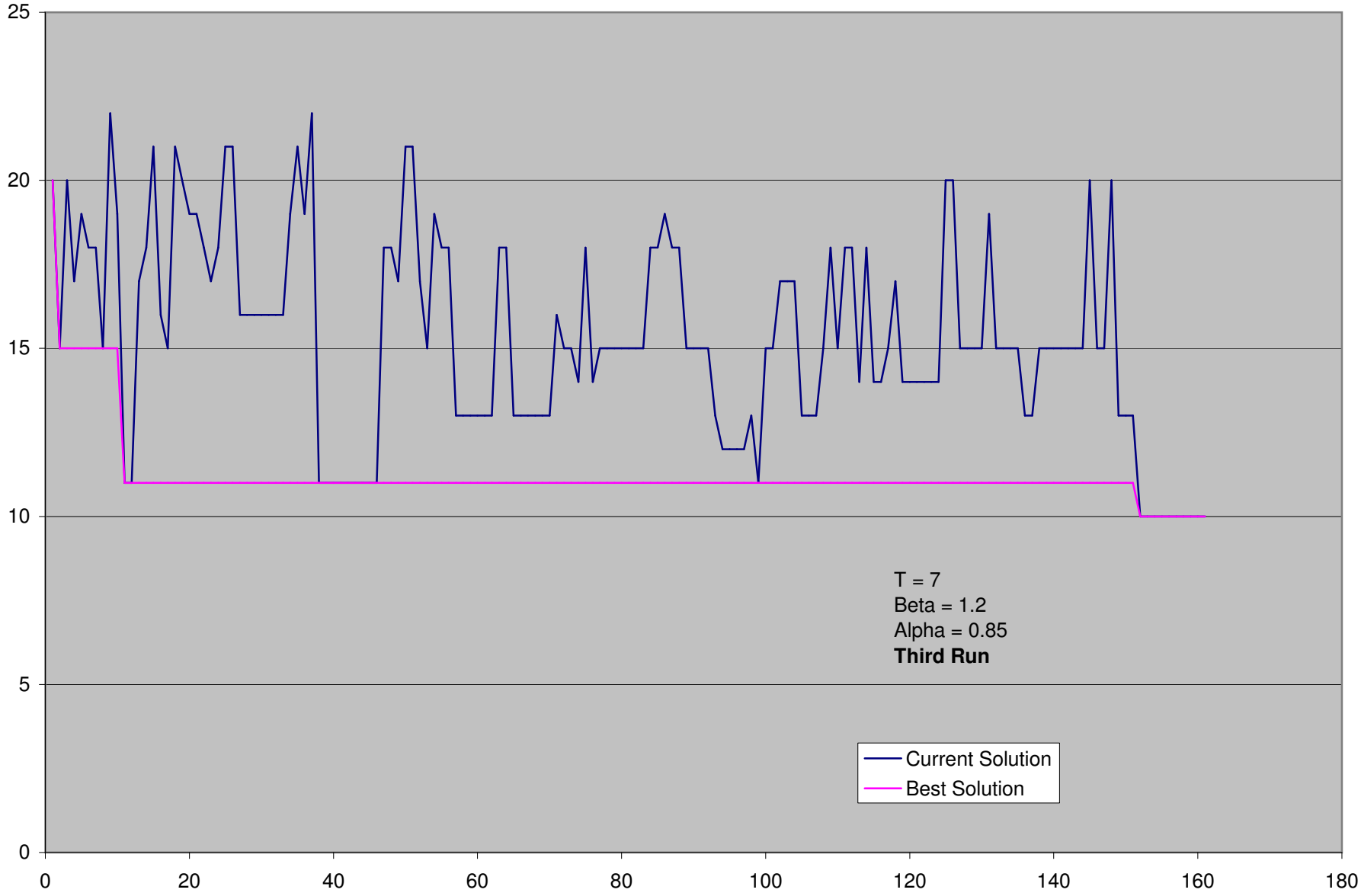

25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 7
Beta = 1.2
Alpha = 0.9
First Run
Current Solution
Best Solution


25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 7
Beta = 1.2
Alpha = 0.9
Second Run
Current Solution
Best Solution


25
20
15
10
5
0
0
50
100
150
200
250
300
350
400
450
T = 7
Beta = 1.2
Alpha = 0.9
Third Run
Current Solution
Best Solution


25
20
15
10
5
0
0
20
40
60
80
100
120
140
160
180
T = 7
Beta = 1.2
Alpha = 0.85
First Run
Current Solution
Best Solution


25
20
15
10
5
0
0
20
40
60
80
100
120
140
160
180
T = 7
Beta = 1.2
Alpha = 0.85
Second Run
Current Solution
Best Solution


25
20
15
10
5
0
0
20
40
60
80
100
120
140
160
180
T = 7
Beta = 1.2
Alpha = 0.85
Third Run
Current Solution
Best Solution