SOP optimization with simulate annealing

- ## Problem Description:

The Sequential Ordering Problem (SOP) with precedence constraints consists of finding a minimum weight Hamiltonian path on a directed graph with weights on the arcs and on the nodes, subject to precedence constraints among nodes.

- ## Instances Description:

Instances are provided by TSPLIB that it is a library of sample instances for the TSP (and related problems like SOP, ATSP, HCP) from various sources and of various types.

Each instance file consists of two part as **specification part** that contains information about the instance data and **data part**.

- ## Algorithm Description:

The algorithm designed base on the *LUCA MARIA GAMBARDELLA & MARCO DORIGO* that described in related paper as *(An Ant Colony System Hybridized with a New Local Search for the Sequential Ordering Problem).*

Constructive heuristic used for generating initial solution in the way that from the bingeing each time minimum possible length edge based on precedence condition selected.

For neighboring method to move from current solution to another, **Lexicographic Search** using **forwarding and back warding path-preserving-3-exchange** applied.

The only difference is that in this algorithm lexicographic search doesn't applied on whole search space by iteratively change the parameters "h, i, j", instead random "h" generated and according to that random "i,j" created to do the search.

With the use of loop with size half of dimension forward and with same size loop backward exchanging applied.

It means that in each simulated annealing iteration best solution selected from a list of solutions with size of problem dimension.

- Initial Constructive heuristic
- O(dimension/2) forward searching with random "h, i, j" parameters.
- O(dimension/2) backward searching with random "h, i, j" parameters.

- Selecting the best from search as next solution

Algorithm time complexity:

```
for it in range(int(dimension/2)):
        h = randrange(0, dimension-3)
        i = h + 1
        ...
        for j in range(i, len(solution)):
                for dep in deps[solution[j]]:
                        ...
```

As code shows the forward and backward search consist of 3 loops so the time complexities is $O(n^3)$.

```
def get_neighbor(problem, dependencies, state, cost):
  …
  new_state1 = fpp3exchange(problem, dependencies, state)
  new_state2 = bpp3exchange(problem, dependencies, state)
  …
```

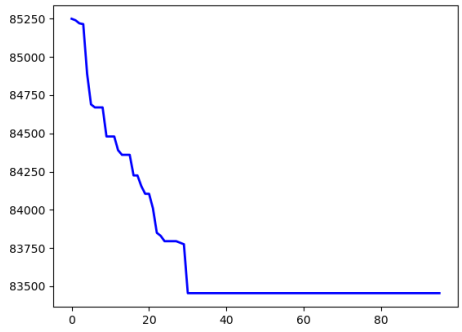
and the neighboring function calling both of them for selecting new solution so the searching algorithm complexity is O(n3).

For updating the temperature 2 methods (**logarithmic** and **exponential**) applied to find the best to work with.

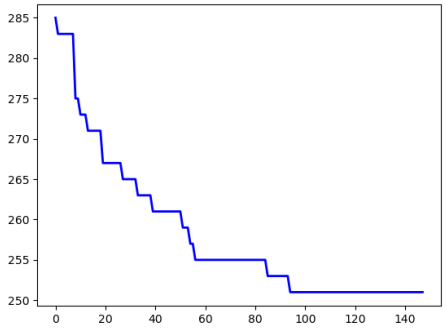
• Algorithm Results:

```
T = 1
ALPHA = 0.8              (for using in temperature updating)
TEMP_MODE = EXP          (temperature updating method)
INIT_HEURISTIC = True    (using initial heuristic)
NUM_ITERATIONS = 500
```

- Algorithm progress plot for sample instances:



p43.4.sop



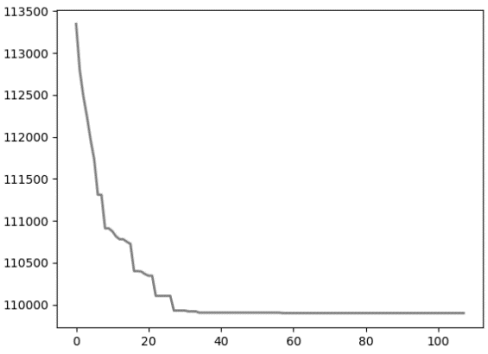
jpeg.4753.54.sop

The whole results (main, max, avg) came in table in excel file named “Results”.

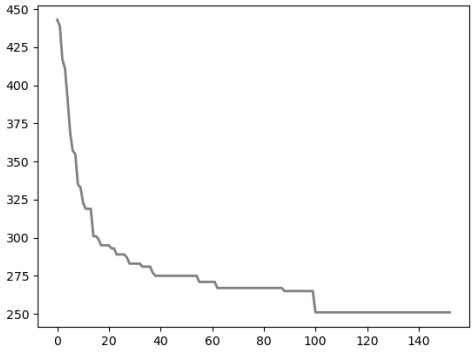
- Initial methods comparison:

T = 1
ALPHA = 0.8
TEMP_MODE = EXP
NUM_ITERATIONS = 500

- Random

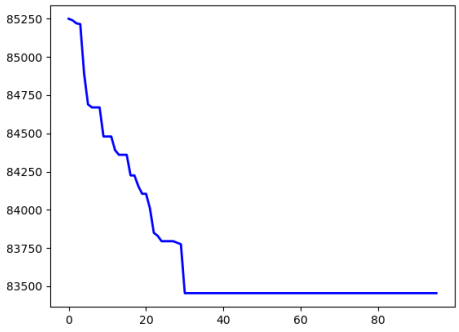


p43.4.sop

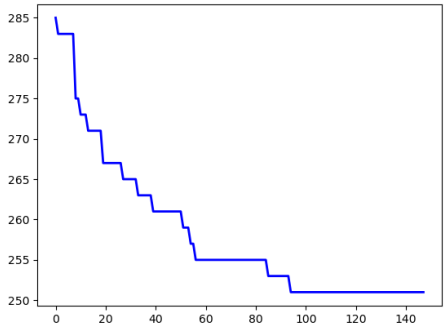


jpeg.4753.54.sop

- Heuristic:



p43.4.sop

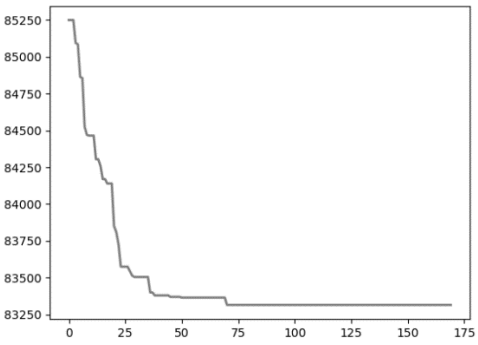


jpeg.4753.54.sop

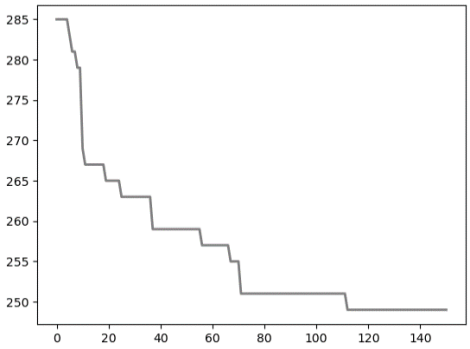
- Temperature update methods comparison:

T = 1
ALPHA = 0.9
INIT_HEURISTIC = True
NUM_ITERATIONS = 500

- Linear (ALPHA * T):

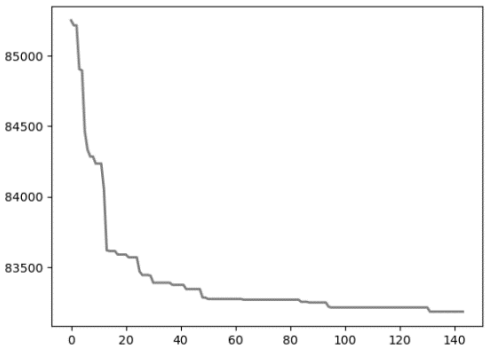


p43.4.sop

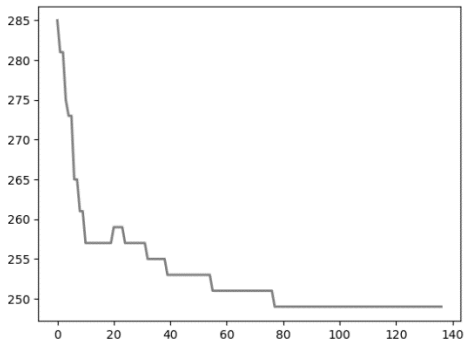


jpeg.4753.54.sop

- Logarithmic (T0 / math.log(step)):

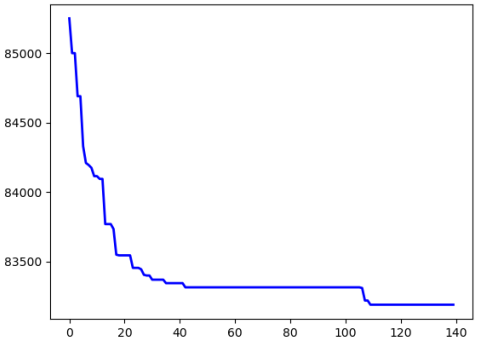


p43.4.sop

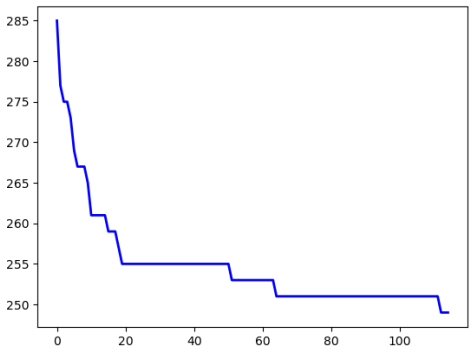


jpeg.4753.54.sop

- Exponential exp(-ALPHA * step)*T0:



p43.4.sop



jpeg.4753.54.sop

as plots show exponential method perform a little bit better search in compare with other.

- Comparison with BKSs:

Instances run with bellow config:

```
T = 1
ALPHA = 0.9
TEMP_MODE = EXP
INIT_HEURISTIC = True
NUM_ITERATIONS = 500
```

Instances with run time under 30 seconds ran 20 times and other with ran 10 times

- Algorithm analysis:

Strength:

this algorithm is much faster that algorithm explained in the related origin paper cause instead of searching whole space with time complexity $O(n^3)$, perform the search just for ***"problem.dimension"*** times.

The results are really close to the paper method in most of the cases.

Weakness:

because of searching the less problem area that related paper method it has a bit

Our code has two main methods:

```
static int LUPdecompose(int size, Type **A, int *P);
```

which return LU matrix in A and permutation matrix in P.

*static int LUPinverse(int size, int *P, Type** LU, Type **B, Type *X, Type *Y);*

which return invers of matrix in A in LU.

- Compiling:

Space complexity of algorithm is ***O(n2)*** which for large size of ***n*** may cause problem due to default stack size per application as ***2MB*** in my OS and compiler base config.

Because of that I preserve more space for stack size to prevent segment fault of code that cause sudden execution termination at the start of running.

```
gcc -Wall -pg lup_matrix_inverse.c -o0 -o int_500_out.exe
```

`-o`: specify output exe file name

`-o0`: without optimizing

`-pg`: Generate extra code to write profile information suitable use gprof.

- System Information's:

CPU: core i5 8th generation

RAM: 8GB

OS: windows 10

Cache: 1L = 256KB, 2L = 1MB, 3L = 6MB

- Performance profiling with Gprah:

```
gprof int_500_out.exe > int_500_profile-data.txt
```

(500 * 500) Matrix

int data type:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 63.41      0.26     0.26                             LUPinverse
 36.59      0.41     0.15                             LUPdecompose
  0.00      0.41     0.00        4     0.00     0.00  allocate_2d
  0.00      0.41     0.00        1     0.00     0.00  initial_matix
```

float data type:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 68.09      0.32     0.32                             LUPinverse
 31.91      0.47     0.15                             LUPdecompose
  0.00      0.47     0.00        4     0.00     0.00  allocate_2d
  0.00      0.47     0.00        1     0.00     0.00  initial_matix
```

double data type:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 70.00      0.35     0.35                             LUPinverse
 30.00      0.50     0.15                             LUPdecompose
  0.00      0.50     0.00        4     0.00     0.00  allocate_2d
  0.00      0.50     0.00        1     0.00     0.00  initial_matix
```

(1000 * 1000) Matrix

float data type:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 68.83      2.54     2.54                             LUPinverse
 31.17      3.69     1.15                             LUPdecompose
  0.00      3.69     0.00        4     0.00     0.00  allocate_2d
  0.00      3.69     0.00        1     0.00     0.00  initial_matix
```

double data type:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 69.05      2.61     2.61                             LUPinverse
 30.69      3.77     1.16                             LUPdecompose
  0.26      3.78     0.01        1    10.00    10.00  initial_matix
  0.00      3.78     0.00        4     0.00     0.00  allocate_2d
```

- Performance profiling with Vtune:

1st function address always is ***LUPinverse*** , 2nd function address always is ***LUPdecompose***

(500 * 500) Matrix

int data type:

Elapsed Time: 0.475s

CPU Time:	0.455s
Total Thread Count:	2
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401d5d	int_500_out.exe	0.288s
func@0x401aaa	int_500_out.exe	0.148s
_sin_default	msvcrt.dll	0.010s
_open	msvcrt.dll	0.009s

*N/A is applied to non-summable metrics.

float data type:

Elapsed Time: 0.539s

CPU Time:	0.515s
Total Thread Count:	2
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401d39	float_500_out.exe	0.347s
func@0x401a75	float_500_out.exe	0.168s

*N/A is applied to non-summable metrics.

double data type:

Elapsed Time: 0.546s

CPU Time: 0.525s
Total Thread Count: 2
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401d41	double_500_out.exe	0.357s
func@0x401a7d	double_500_out.exe	0.157s
_sin_default	msvcrt.dll	0.011s

*N/A is applied to non-summable metrics.

(1000 * 1000) Matrix

float data type:

Elapsed Time: 4.141s

CPU Time: 4.097s
Total Thread Count: 2
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401d39	float_1000_out.exe	2.781s
func@0x401a75	float_1000_out.exe	1.250s
_sin_default	msvcrt.dll	0.029s
_math_exit	msvcrt.dll	0.020s
_87except	msvcrt.dll	0.010s
[Others]	msvcrt.dll	0.007s

*N/A is applied to non-summable metrics.

double data type:

Elapsed Time: 4.229s

CPU Time:	4.189s
Total Thread Count:	2
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. typically results in improving overall application performance.

Function	Module	CPU Time
func@0x401d41	double_1000_out.exe	2.837s
func@0x401a7d	double_1000_out.exe	1.279s
_math_exit	msvcrt.dll	0.039s
_sin_default	msvcrt.dll	0.021s
Sleep	KernelBase.dll	0.011s
[Others]		0.002s

*N/A is applied to non-summable metrics.

- ## Execution analysis:

Program has two main method as:

- ***LUPdecompose:***
  With time complexity as O(n2) based on code reviewing and space complexity as O(n+n2) =O(n2).
  But according to the algorithm documents, LU decomposition can be computed in time O(M(n)). $M(n) \geq n^a$ where a > 2. It means O (n2.376)

- ***LUPinverse:***
  With time complexity as O(n3) and space complexity as O(3n+2n2) =O(n2)

And according to code result on the used machine, size (in bytes) and precision (in number of decimal digits) of

float: 4 and 6,
double: 8 and 15

- ## Proposed improvement:

As matrix size and comparison accuracy increases the execution times growth.

Matrix space always is a good application for parallelism due to high ILP property of matrix-based problems.

As the result shows ***LUPinverse*** takes the most of execution time.

This method solving mathematical equation iteratively over each column.

We can divide this work over multi thread tasks that independently solve equation for specific column vector and in this way make the code much faster.

For another method ***LUPdecompose*** that take O(n2) time we can split the outer loop in specific sizes and pass them to some thread and make it faster.in other word the partial pivoting process that is comparison-based process across each column could be done in parallel manner.