SOP optimization with Simulated annealing, GRASP

- ## 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 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 and best solution selected among them.

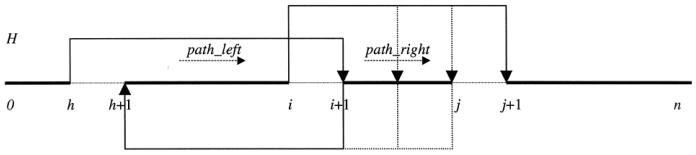


Figure 6. Lexicographic forward path-preserving-3-exchange.

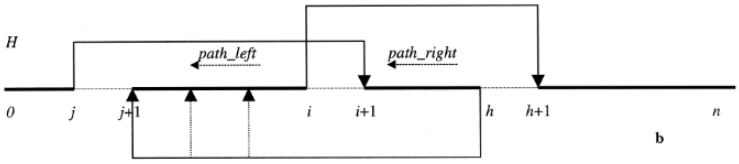


Figure 7. Lexicographic backward path-preserving-3-exchange.

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 complexity is $O(n^3)$.

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

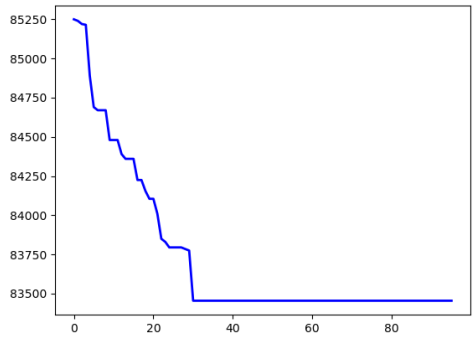
and the neighboring function calling both of them for selecting new solution so the searching algorithm complexity is O(n3).

For updating the temperature 3 methods (**linear** and **logarithmic** and **exponential**) applied to find the best to work with.

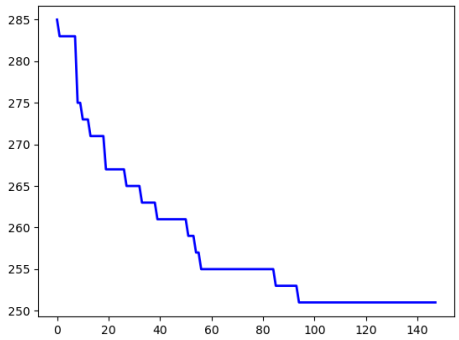
- Simulated annealing algorithm progress:

```
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



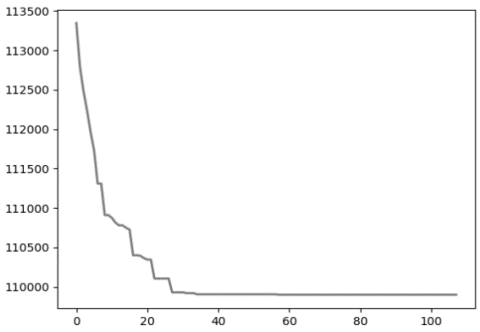
jpeg.4753.54.sop

The whole results (main, max, avg) came at the end.

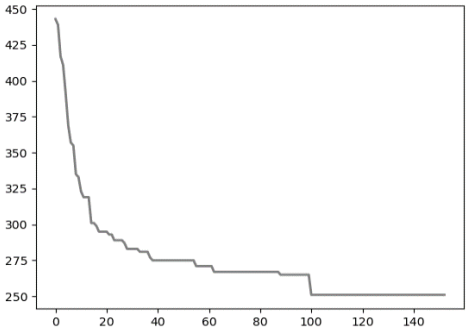
- Simulated annealing Initial methods comparison:

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

- Random

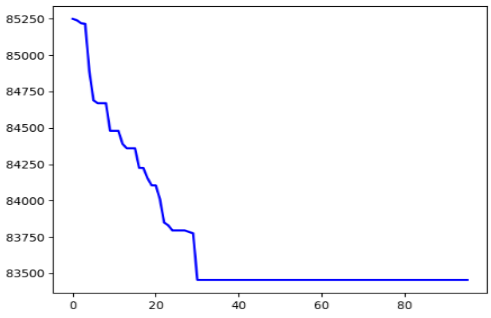


p43.4.sop

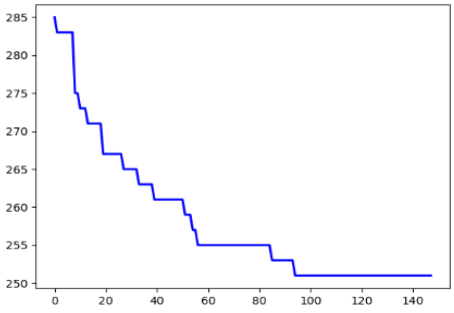


jpeg.4753.54.sop

- Heuristic:



p43.4.sop



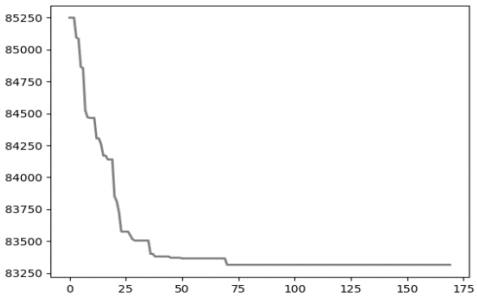
jpeg.4753.54.sop

as result shows with heuristic method algorithm start from much better initial solution and in some cases leads to better final solution.
For 10 instances as test heuristic method gave better solution.

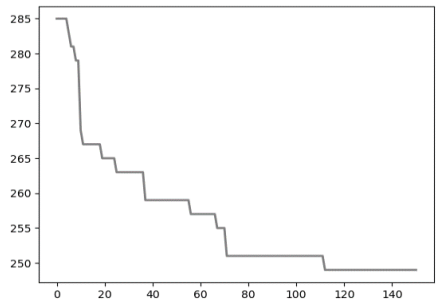
- Simulated annealing temperature update methods comparison:

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

- Linear (ALPHA * T):

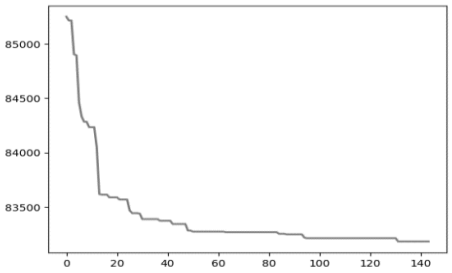


p43.4.sop

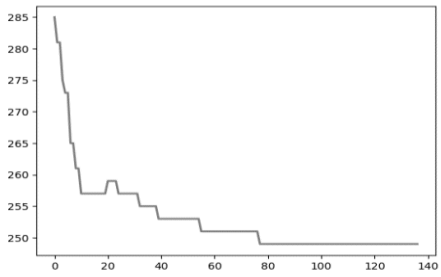


jpeg.4753.54.sop

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

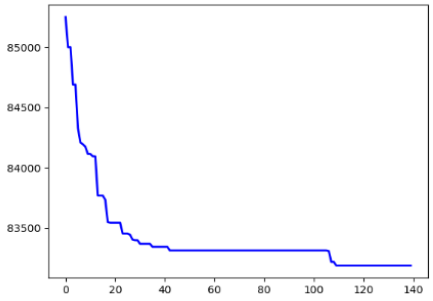


p43.4.sop

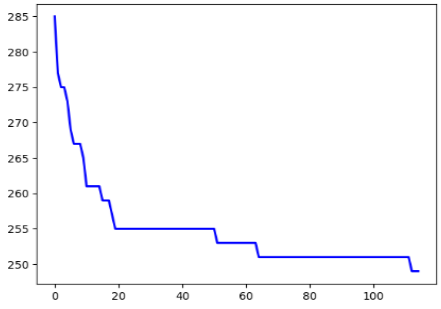


jpeg.4753.54.sop

- 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.

- Simulated annealing 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.

On the "E" instances folder, results were near to the BK answers except bellow instance types: kro124p.*, prob.100, prob.7. *, rbg109a.sop.
it seems that from view of this algorithm, these problems were harder than other.

On the "H" instances folder, results were almost similar to the BK answers (with maximum difference equal to 7).

the "M" instances were much more time consuming and the results weren't as good as "H" folder.

The whole results (main, max, avg) came at the end.

- Simulated annealing 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, in some instances it reaches a little bit worst result.

In overall the algorithm is a less time-consuming version of paper method with good acceptable results.

- Simulated annealing results:

Instance	BKS	best	average	worst	min_time	avg_time	max_time	Diff of Best
ft53.2.sop	8026	9473	10312.8	11041	1.1050	1.2013	1.3194	1447
ft70.2.sop	40419	44076	45597.4	46640	2.0604	2.3552	2.7556	3657
kro124p.1.sop	39420	46934	48537.15	49917	7.9588	8.9360	9.8073	7514
kro124p.3.sop	49499	59438	62762.4	66501	3.4378	4.8182	5.6962	9939
p43.1.sop	28140	28290	28598.75	28810	0.8178	0.9116	1.0894	150
p43.4.sop	83005	83140	83270.5	83445	0.3390	0.3805	0.4697	135
prob.100.sop	1123	3158	3870.05	4849	4.0428	4.2812	4.5359	2035
prob.5.sop	243	421	548.15	682	0.7549	0.8216	0.9136	178
prob.7.40.sop	1071	1788	2301.3	2981	0.6532	0.7296	0.8986	717
prob.7.60.sop	912	1952	2545.5	2968	1.4327	1.5482	1.6655	1040
prob.7.70.sop	879	2310	2881.25	3525	1.9058	2.1013	2.2778	1431
rbg050a.sop	400	407	439.4	478	0.8008	0.9093	1.0082	7
rbg050b.sop	397	403	432.75	463	0.8028	0.9041	1.1270	6
rbg050c.sop	467	468	480.9	494	0.7749	0.8580	0.9614	1
rbg105a.sop	1023	1064	1104.55	1143	1.9378	2.1143	2.4140	41
rbg118a.sop	1423	1424	1450.35	1507	1.8151	1.9227	2.1562	1
rbg124a.sop	1361	1366	1397.25	1436	1.8051	1.9065	2.0226	5
rbg126a.sop	1381	1398	1421.6	1481	1.9942	2.2430	2.6624	17
rbg143a.sop	1765	1774	1801.35	1832	2.1233	2.2379	2.4345	9
rbg219a.sop	2544	2578	2605.35	2632	6.6901	7.2614	7.8829	34
rbg247a.sop	3062	3101	3140.35	3187	8.1642	8.7001	9.8681	39
rbg341a.sop	2568	3117	3217.9	3342	26.782	29.577	33.403	549
ry48p.2.sop	16666	18290	20884.7	23105	0.9273	1.0249	1.1865	1624
ry48p.3.sop	19894	22029	23826.5	25251	0.7129	0.8665	1.0328	2135
prob.7.65.sop	915	1649	1930.65	2188	1.6960	2.2015	2.6070	734
rbg109a.sop	198	1046	1081.1	1110	2.0425	2.5092	2.7716	848
rbg117a.sop	1494	1497	1516.55	1548	1.4388	1.6960	1.8951	3
rbg150a.sop	1750	1783	1829.5	1866	3.7965	4.4883	5.2027	33
rbg174a.sop	2033	2059	2114.5	2146	5.1369	6.4195	7.2307	26
rbg190a.sop	2241	2269	2290.0	2311	5.2462	6.6285	7.6026	28
rbg285a.sop	3482	3557	3604.55	3668	14.519	15.807	18.102	75
rbg358a.sop	2545	2884	3001.15	3141	37.397	41.883	47.849	339
gsm.153.124.sop	1109	1110	1121.05	1129	0.6336	0.7397	0.9579	1
gsm.462.77.sop	577	578	581.45	587	0.5404	0.5812	0.6931	1
jpeg.3184.107.sop	791	798	808.0	817	0.8498	0.9889	1.1526	7
jpeg.4753.54.sop	245	247	256.5	269	0.3554	0.4670	0.7355	2
susan.260.158.sop	1016	1022	1035.65	1055	1.7578	2.1034	2.4285	6
typeset.15577.36.sop	155	155	160.65	171	0.2309	0.2598	0.3679	0

typeset.1723.25.sop	64	64	69.85	78	0.1578	0.1951	0.3143	0
typeset.19972.246.sop	2018	2018	2021.6	2034	1.3684	1.4857	1.7511	0
typeset.4724.433.sop	3466	3468	3478.2	3496	6.0954	6.8138	8.1372	2
typeset.16000.68.sop	84	84	85.2	90	0.6667	0.8752	1.1691	0
typeset.10835.26.sop	127	127	130.9	137	0.1950	0.2186	0.2806	0
R.200.100.1.sop	61	340	402.3	453	27.408	28.773	30.956	279
R.200.100.60.sop	71749	72804	74300.15	75808	1.8221	1.9594	2.2568	1055
R.200.1000.30.sop	41196	46190	49303.0	52981	2.2330	2.5983	3.2566	4994
R.200.1000.60.sop	71556	72846	74722.2	76561	1.9925	2.5362	2.8859	1290
R.300.1000.60.sop	109471	110993	112747.95	114203	5.2474	6.6597	9.1146	1522
R.400.1000.15.sop	38963	64354	66147.15	68407	21.304	22.863	25.351	25391
R.500.1000.1.sop	1316	3532	3733.14	3926	631.41	738.70	858.61	2216
R.600.100.60.sop	23293	24300	24479.8	24711	42.649	49.824	66.338	1007
R.600.1000.1.sop	1337	3676	3681.5	3687	1073.2	10774.5	1081.7	2339
R.600.1000.60.sop	214608	224197	226373.6	228394	29.580	33.008	39.903	9589
R.700.1000.15.sop	65678	121526	123669.0	126399	77.331	81.251	92.250	55848
R.700.1000.60.sop	245589	257974	259705.3	261393	80.584	92.586	100.08	12385

- GRASP algorithm:

```
procedure grasp()
1    InputInstance();
2    for GRASP stopping criterion not satisfied →
3        ConstructGreedyRandomizedSolution(Solution);
4        LocalSearch(Solution);
5        UpdateSolution(Solution,BestSolutionFound);
6    rof;
7    return(BestSolutionFound)
end grasp;
```

As the pseudo-code shows the algorithm contains of 3 main part.

For SOP problems we implement the methods as bellow:

"ConstructGreedyRandomizedSolution()":

Like what we did in initial heuristic of simulated annealing, step by step we choose next feasible greedy node to add to the path, but instead of using the best node from candidate list we choose from randomly between (0, ALPH) percentages best of candidate list for next node selection (rank-based selection) and by this manner we add random factor beside greedy factor as GRASP behaves.

```
index = int(rnd.uniform(0, ALPHA) * graph.dimension)
dest = rnd.choice(list(candidates[0:1+index]))[0]
solution.append(dest)
```

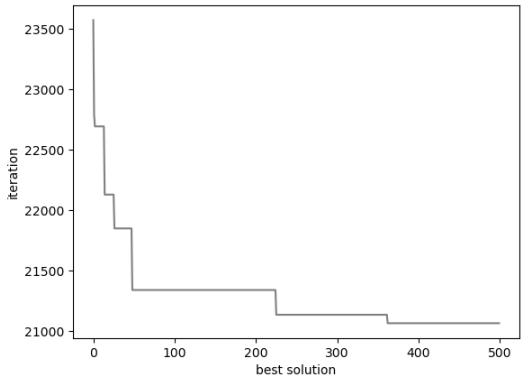
"LocalSearch()":

For searching the local area just like we mentioned in ***"get_neighbor()"*** previously ,we choose local optimal from best of backward and forward solutions.

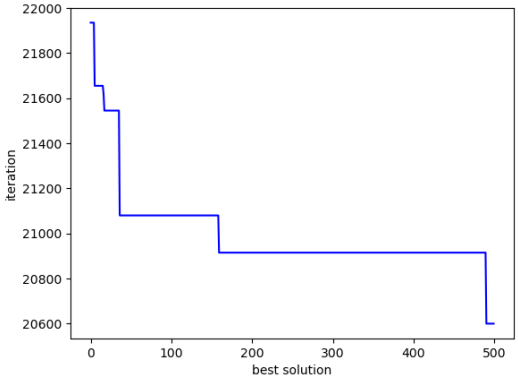
"UpdateSolution()":

At the end we update global best solution if local optimal was better (smaller cost).

- Local search role analysis:
  - ESC78.sop

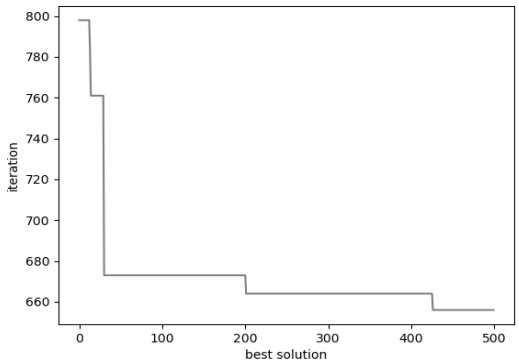


Without local search

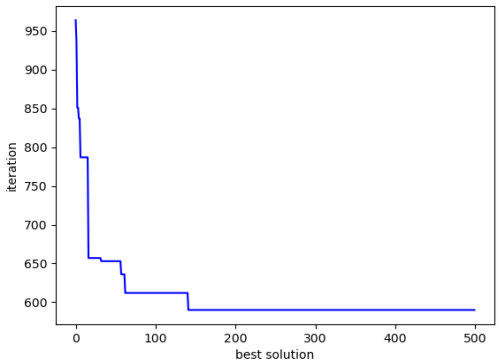


With local search

  - R.200.100.1



Without local search



With local search

As plots shows local search operator cause finding better quality of solution at the end, it may or may not start from better solution. But the benefit of using it is obvious.

ALPHA = 0.02
NUM_ITERATIONS = 500

- **Without local search:**

Instance	best	worst	average	avg_time
ESC78.sop	20465	21370	21194.5	1.8862
R.200.100.60.sop	85057	87786	85892.3	24.435
susan.260.158.sop	1124	1143	1134.6	6.5894

- **With local search:**

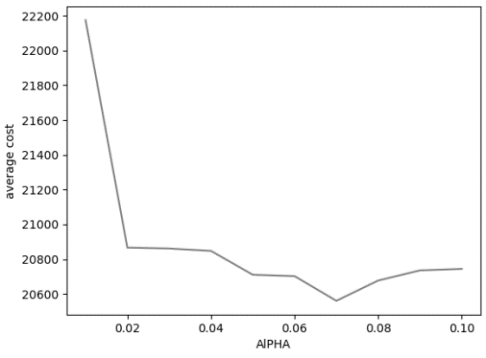
Instance	best	worst	average	avg_time
ESC78.sop	20405	21220	20895.0	3.9388
R.200.100.60.sop	83444	87524	85955.7	26.316
susan.260.158.sop	1121	1139	1131.3	8.2227

(Results per 10 time of run for each instance)

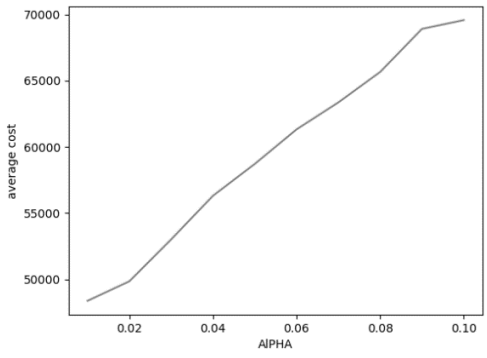
Local search operation gives better result overall and due to its searching process, it takes more time.

- ALPHA value (maximum size of RCL list) comparison:

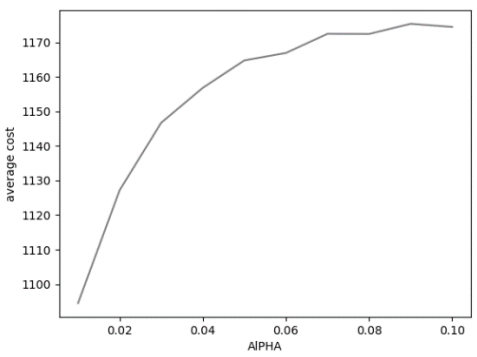
Plot of average cost for 20 time run of each instance per AlPHA comes in bellow



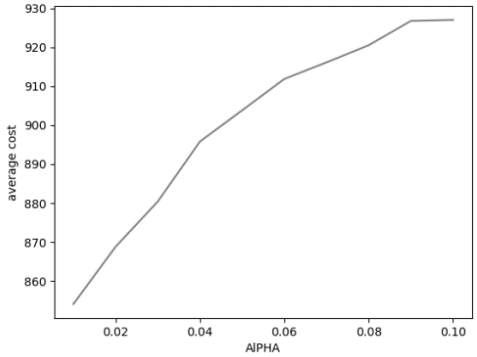
ESC78.sop



kro124p.1.sop



sasan.260.158.sop



jpeg.3184l.107.sop

based on the result of some instances that plot of some of them came above, ALPHA=0.2 selected for all instances. In some cases, bigger alpha made better best solution.
It seems that bigger alpha cause bigger variance of solutions and due to bigger variance, it causes higher average.
Higher alpha value does more exploration and add more randomness on the other side lower alpha value does more exploitation.

- GRASP solutions variance analysis:

  - Solution variance of 10 times of running algorithm.

Instance	best	average	worst	variance
ESC78.sop	20400	20776.0	20970	200.88
kro124p.3.sop	62362	64364.6	66162	1332.38
p43.4.sop	84820	84820.0	84820	0.0
prob.7.65.sop	1689	1719.0	1774	30.78
jpeg.3184.107.sop	865	869.4	873	3.2
susan.260.158.sop	1116	1124.0	1132	5.32
R.200.100.1.sop	340	402.3	453	28.773
R.600.100.60.sop	28711	28765.5	28820	54.5

  - Solution variance of global optimal during running the algorithm.

Instance	variance
ESC78.sop	301.89
kro124p.3.sop	1749.65
p43.4.sop	47.29
prob.7.65.sop	71.70
jpeg.3184.107.sop	8.47

susan.260.158.sop	13.26
R.200.100.1.sop	24.96

It seems that in some cases data variance is high and it's because of randomness aspect of the algorithm.

- Simulated annealing vs GRASP comparison:

  - **Simulated annealing:**

Instance	BKS	best	average	worst	avg_time	Diff of Best
ESC78.sop	18230	18250	18400.5	18535	1.7924	20
kro124p.3.sop	49499	59438	62762.4	66501	4.8182	9939
prob.7.65.sop	915	1649	1930.65	2188	2.2015	734
susan.260.158.sop	1016	1123	1128/8	1136	8.6370	6
R.200.100.1.sop	61	340	402.3	453	28.773	279
R.400.1000.15.sop	38963	64354	66147.15	68407	22.863	279

  - **GRASP:**

Instance	BKS	best	average	worst	avg_time	Diff of Best
ESC78.sop	18230	20405	20895	21220	3.8190	2175
kro124p.3.sop	49499	60161	63202.9	67032	6.2631	10662
prob.7.65.sop	915	1560	1658/4	1812	3.2334	645
susan.260.158.sop	1016	1123	1128/8	1136	8.6370	107
R.200.100.1.sop	61	393	448.7	483	56.704	332
R.400.1000.15.sop	38963	144545	148052.9	150760	120.86	105582

The GRASP algorithm average solution is worse than simulated annealing in compare and its variances of solution are bigger.
It also takes more time in average.in compare with pure greedy, GRASP perform better due to having the random aspect beside greedy.

- GRASP results:

Instance	BKS	best	average	worst	min_time	avg_time	max_time	Diff of Best
br17.1.sop	41	61	61	61	0.2543	0.2572	0.2632	20
br17.10.sop	55	63	63	63	0.2124	0.2153	0.2170	8
br17.12.sop	55	63	63	63	0.2064	0.2315	0.2732	8
ESC78.sop	18230	20405	20895	21220	3.7371	3.8190	4.0162	2175
ESC98.sop	2125	2125	2125	2125	7.9125	8.2295	8.5833	0

ft53.2.sop	8026	10741	11249.2	11475	1.9532	2.1721	2.4002	2715
ft70.2.sop	40419	44566	45606.9	46400	3.3997	3.6471	3.8445	4147
kro124p.1.sop	39420	48588	49968.0	51023	9.2160	9.4826	9.6200	9168
kro124p.3.sop	49499	60161	63202.9	67032	5.9466	6.2631	6.6445	10662
p43.1.sop	28140	29120	29120	29120	1.3682	1.4517	1.5105	980
p43.4.sop	83005	84820	84820	84820	1.0476	1.1261	1.2562	1815
prob.100.sop	1123	2062	2248/6	2378	7.8009	8.0781	8.3549	939
prob.5.sop	243	417	417	417	1.2846	1.3691	1.5334	174
prob.7.40.sop	1071	2477	2477	2477	1.0850	1.1679	1.3081	1406
prob.7.60.sop	912	1530	1563	1623	2.6015	2.7256	2.9206	618
prob.7.65.sop	915	1560	1658/4	1812	2.9690	3.2334	3.5530	645
prob.7.70.sop	879	1608	1665/2	1762	3.4080	3.7098	4.1187	729
rbg050a.sop	400	474	475/2	476	1.7683	1.8415	1.9509	74
rbg050b.sop	397	501	507/8	517	1.6675	1.7497	1.9154	104
rbg050c.sop	467	543	546/8	549	1.6695	1.8106	1.9277	76
rbg105a.sop	1023	1300	1343/2	1362	6.0060	6.0823	6.1843	277
rbg109a.sop	198	1323	1336/2	1352	6.5961	6.8382	7.0040	1125
rbg117a.sop	1494	1654	1668/6	1681	6.6315	6.8512	7.0958	160
rbg118a.sop	1423	1641	1683	1699	6.7349	7.0234	7.1377	218
rbg124a.sop	1361	1623	1635/4	1645	7.8546	8.1561	8.5900	262
rbg126a.sop	1381	1678	1709/2	1735	8.3840	8.5852	8.8323	297
rbg143a.sop	1765	2077	2091/6	2111	12.001	10.228	9.8187	312
rbg150a.sop	1750	2101	2128/4	2144	13.088	13.528	14.159	351
rbg174a.sop	2033	2461	2488/8	2501	18.461	19.102	19.628	428
rbg190a.sop	2241	2826	2857/2	2904	21.700	21.946	22.479	585
rbg219a.sop	2544	3301	3353/8	3396	30.687	31.354	31.793	757
rbg247a.sop	3062	3930	3992/8	4043	38.060	38.995	39.407	868
rbg285a.sop	3482	4576	4617/2	4674	54.148	55.410	56.633	1094
rbg341a.sop	2568	4583	4643/2	4716	100.00	99.833	99.956	2015
rbg358a.sop	2545	4746	4802/6	4859	118.75	120.36	123.48	2201
ry48p.2.sop	16666	20407	20407	20407	1.5763	1.6015	1.6224	3741
ry48p.3.sop	19894	26118	26118	26118	1.2955	1.3214	1.3502	6224
gsm.153.124.sop	1109	1177	1185	1194	4.7513	4.8959	5.1867	68
gsm.462.77.sop	577	585	586/4	587	2.1583	2.2547	2.4721	8
jpeg.3184.107.sop	791	856	865/2	875	3.9890	4.0605	4.1194	65
jpeg.4753.54.sop	245	257	259	261	1.1449	1.2211	1.3740	12
susan.260.158.sop	1016	1123	1128/8	1136	8.5176	8.6370	8.7726	107
typeset.10835.26.sop	127	137	137	137	0.4084	0.4323	0.4775	10
typeset.15577.36.sop	155	175	175	175	0.6161	0.6451	0.6712	20
typeset.16000.68.sop	84	85	86/2	87	1.6542	1.7778	1.8768	1
typeset.1723.25.sop	64	72	72	72	0.3459	0.3598	0.3951	8
typeset.19972.246.sop	2018	2068	2073/6	2080	23.296	23.464	23.676	50
typeset.4724.433.sop	3466	3657	3668/8	3679	107.35	107.63	107.94	191

R.200.100.1.sop	61	393	448.7	483	56.464	56.704	56.955	332
R.200.100.60.sop	71749	84814	85629/8	86253	24.674	25.005	25.163	13065
R.200.1000.30.sop	41196	71241	73088/8	74544	19.661	19.811	20.100	30045
R.200.1000.60.sop	71556	83161	86006	87744	24.430	24.736	24.873	11605
R.300.1000.60.sop	109471	134691	135201/8	136134	72.406	72.763	73.572	25220
R.400.1000.15.sop	38963	144545	148052.9	150760	120.11	120.86	121.68	105582
R.500.1000.1.sop	1316	11371	12200.9	12815	1025.4	1084.3	998.53	
R.600.100.60.sop		28711	28765.5	28820	522.35	543.22	564.09	