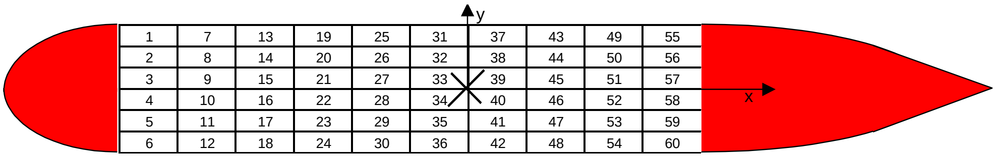

Heuristics Assignment – Ship Balancing

Due: Wed 30 May 2018 at 9pm via Canvas

Dr L. Search has contracted to transport about 100 containers by ship from Auckland to the UK. She is seeking your assistance in developing a loading pattern for the containers to ensure the ship is as well balanced as possible. Each container is packed tightly into an area 3m by 6m. The only restriction on the container packing is that the containers may be packed no more than two deep (i.e. one on top of the other). Each possible deck loading position is shown below, where up to two containers can be loaded at each of these 60 positions.



A measure of 'well balanced' is given by $5|d_y|+|d_x|$, where d_x and d_y are the co-ordinates of the centre of mass (in the horizontal plane) relative to the ship's centre point (marked with a cross above) in the x and y directions shown.[1] The container weights, w(1), w(2), …, w(n) are given in associated *weight files* for the $n \le 120$ containers. For convenience, we define w(0)=0; ie container '0' has zero weight.

Let the possible container loading positions be numbered 1-120, where loading positions i and 60+i represent the two loading positions (one above the other) associated with deck position i, i=1, 2, …, 60. (For our balance calculations, we are only concerned about the horizontal positions of the containers, and so we do not require that position i be filled before position i+60 is used.) You should assume (x_i,y_i), i=1, 2, …, 120 contains the co-ordinates of the centre of loading position *i* in the x, y co-ordinates shown. Let $\mathbf{s}=(s_i, i=1,2,\ldots,120)$ be a solution in which s_i gives the container in loading position i, or $s_i=0$ if the packing location is empty (ie filled by our special zero-weight 'container 0').

Theory Questions:

*Question 1: Give a formula for f(***s***), the objective function associated with some solution* **s**=(s_i, i=1,2,…,120). *Be sure to define any new notation you introduce. Hint: For convenience, we defined the mass of container 0, w(0), as being w(0)=0.*

Consider a neighbourhood rule in which we form a new neighbouring solution ***t***(**s**,a,b) from solution **s** by swapping the containers (which may be container '0', meaning an empty position) in any two loading positions *a* and *b*.

---

[1] Any non-Engineers should note that the centre of mass (d_x, d_y) of n objects of mass w_i, i=1, 2, …, n=120, at locations (x_i,y_i) , i=1, 2, …, n=120 respectively is given by $d_x = \sum_{i=1}^{n} w_i x_i / \sum_{i=1}^{n} w_i$, $d_y = \sum_{i=1}^{n} w_i y_i / \sum_{i=1}^{n} w_i$

Question 2: Give a formula for the neighbouring solution $\boldsymbol{t}(\boldsymbol{s},a,b)$ *constructed from solution* **s** *using this rule. (For example in the seating example, we defined* $y(\mathbf{x},i) = (x_1, x_2, ..., x_{i-1}, x_{i+1}, x_i, x_{i+2}, ..., x_n)$.) Your formula must be in terms of $\mathbf{s}=(s_1, s_2,..., s_{120})$, a and b.

Question 3: Give an English sentence describing, for the case where $s_b=0$, *the physical action we make when we generate a neighbour* $\boldsymbol{t}(\boldsymbol{s},a,b)$.

Question 4: For some combinations of a, b, s_a *and* s_b, *our neighbourhood rule will give a neighbouring solution* $\boldsymbol{t}(\mathbf{s},a,b)$ *that is physically equivalent to solution* **s**. *List these combinations (identifying them using maths, not words), and for each of these, give a few words stating why it gives a neighbouring solution that is physically unchanged.*

Question 5: By considering ***all*** *appropriate values for a and b, define the set of all* ***distinct*** *solutions that are neighbours to some solution* $\boldsymbol{s}$. *(In our seating problem, for example, we had* $N(\boldsymbol{x}) = \{y(\boldsymbol{x},i)$ *for all* $i=1,2,3,4\}$, *but remember this was an artificially small neighbourhood.) In your definition of* $N(\boldsymbol{s})$, *be sure to exclude any of the combinations found above that produced physically equivalent solutions.*

Question 6: Give a formal procedure that updates the objective function when moving from a solution $\boldsymbol{s}$ *to a neighbouring solution* $\boldsymbol{t}(\boldsymbol{s},a,b)$. *Be sure to define all your terms. Introduce (and clearly define) any intermediate values required to make your calculation efficient.*

We will need to generate random starting solutions. This can be done using a ‘card sorting’ algorithm. We place the *n* containers into the first *n* of the 120 loading positions, and then we a randomly shuffle all 120 of the loading positions.

```
Routine to shuffle 120 cards:
for i = 1 to 119
        let j = random number between i+1 and 120 inclusive
                (In C, we use: int j = i+1 + (int)(rand()/(1.0+RAND_MAX) * (120-i));  )
        swap cards in positions i and j
```

Data Provided:
On Canvas you will find the following:

- A plain text *‘positions file’* giving (x_i, y_i) co-ordinates for each of the 60 different deck positions on the ship (see printout below)
- Plain text *weight files* giving the weights of the containers to load for three problems ProbA, ProbB, and ProbC. (See sample printout below.)
- A solution checker spreadsheet *SolnCheckerViewer.xls* into which you will paste your answers.

Repeated Descent Programming Exercise:
Develop a well structured and efficient program in either VB, VBA[2], C, C++, C#, Python, Java, Javascript, Swift or Fortran to complete the following tasks:

1. Read in a ‘weights file’ giving container weights w(i) to be loaded. (See the sample file below.)
2. Read in a ‘positions file’ giving (x_i, y_i) co-ordinates of the deck positions. (See the sample file below.)
3. For the ProbA input weights file, perform and plot two next-descent local searches, each starting from a random starting solution and finishing with a local minimum. Your code should output the objective function for every neighbouring solution you evaluate, whether or not it is kept as an improvement, and also the best solution found so far. You should then plot these using the format shown in Figure 1. Your plot should have “function evaluation count” on the x axis, and “solution quality” plotted on the y axis using a log scale. Your plot should be printed in A4 landscape, and handed in for marking.
4. For each of the weight files (ProbA, ProbB, etc) provided, perform 200 next-descent local searches. Your code should write out the very best solution you find as a single-column text file giving the objective function, and then, for each packing position, the **index** (*not* the weight) of the container in that location (or 0 if the location is empty). Do not include any other data in this file. These files do not need to be handed in. However, you should also paste each solution into the appropriate

[2] You may choose to write your code in VBA inside the Solution Checker spreadsheet. If you do so, you should do your own calculations in VBA without relying on the formulae in the spreadsheet.

sheet of the *Solution Checker* spreadsheet (provide on Canvas), and then hand in the **first page** of a printout for each solution.

Notes:

- We wish to avoid unnecessary operations such as the division in the objective function. (Division is particularly slow.) For these experiments we wish to plot the true objective function, so you will need to include the division whenever you output an objective function value. However, your program should avoid performing any unnecessary divisions within the main neighbourhood search loops.
- Your code needs to efficiently determine when a local optimum has been found without evaluating more neighbours than are needed. Your code should implement a simple straightforward test for this.
- Your code should follow the next-descent examples shown in the lecture notes in that it must continue sweeping the neighbourhood whether or not the current solution is updated at some step of the sweep. Specifically, it must not re-start this neighbourhood sweep just because an improved solution is found.
- This is a simple programming task that should be completed without using classes, nor relying on complex data structures. All logic must be apparent in the main loop of the code, without calls to other methods; for example, do not calculate the neighbourhood beforehand and store it, but instead just loop intelligently. Do not update the best objective via a method; simply do it in the main loop. All data must be stored using simple arrays. Your main search loop should be about half a page of code.

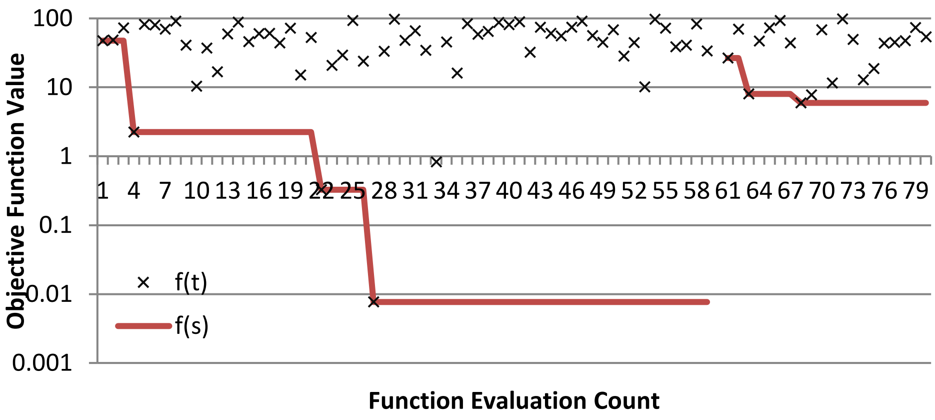


Figure 1: Expected plot format for the local search runs showing the objective function value f(s) for the current solution s (i.e. the best solution found for the current descent) as a line, and f(t), the objective function value for the last evaluated neighbouring solution t (shown as individual points). Note that a blank line is used, in Excel, to break and restart the f(s) line plot, showing when a new random starting solution is generated. This plot shows 2 descents. It uses a log scale for the objective function.

Meta Heuristic Programming Tasks

We now wish to implement Tabu Search for this problem. We will use a history list that remembers the containers that have moved positions during the last h=min(20,int(n/3)) iterations (where there are n containers), and mark as tabu (ie ban) any swaps involving any of these containers.

Question 7: Carefully explain how we can efficiently implement the banning of Tabu swaps in this way. Note that we do not want to actually store a list that has to be searched, but instead be smarter. Hint: Think about iteration counts and flagging the containers in some way.

1. Implement the Tabu Search meta heuristic for this problem. Start from a solution s_1=1, s_2=2, …, s_n=n, s_{n+1}=0, s_{n+2}=0, …, s_{120}=0, where n is the number of containers. Run your Tabu search code on ProbA to produce a plot equivalent to that shown in Figure 1, but now where the current solution s is the current solution (which won't change, but will appear many times in the plot, as we sweep the neighbours), and t is the most recently evaluated neighbour (which may be better or worse than s). You should run your Tabu search for enough iterations to clearly show how it accepts worse

solutions (ideally for about $2n_{up}$ iterations, where n_{up} is the number of iterations before the solution first worsened), but for no more than 10^5 iterations. Hand in this plot (printed on A4 landscape).

Question 8: We defined Tabu moves by recording the containers that were moved recently. Give another suitable alternative approach for recording and defining moves as Tabu.

Bonus Question – approx. 5%: Compare your Tabu search and Next Descent approaches by plotting solution quality against run time, and provide a recommendation as to which method is best. Be sure to hand in a plot comparing the two methods.

In summary you should hand in:

- Hand-written (or typed) answers to the theory questions above.
- A listing of your program(s). Your name & ID must be at the top of the code. You can hand in just one listing, as long as you highlight the two main procedures, being that used for repeated descent and that used for your meta heuristic.
- 3 spreadsheet pages, being the first page of printout obtained for each final repeated-descent solution when it is pasted into the checker spreadsheet.
- A plot of the repeated local descent run for ProbA (as detailed above)
- A plot for the meta-heuristic running on the ProbA test problem.
- A signed statement swearing (upon an appropriate entity of your choice) that the output was produced by running the code handed in without undue assistance from others.

Prize: A small prize will be awarded to the student generating the best solution averaged (in some weighted fashion reflecting problem difficulty) over the sample files provided. Ties shall be broken by earliest time/date of submission. To win the prize, you must submit your solution files and program to the "prize" submission on Canvas.

Container weights ProbA file All weights are in tens of kg. The first values give the number of containers.

100
280.7545651
669.389006
207.764796
47.55080316
830.8559438
275.2896374
942.0694392
514.9914207
346.4314901
387.0415543
888.9294079
589.9274618
967.760778
585.6589384
405.801276
369.0249948
389.7930657
956.3362658
745.1120058
391.4605218
802.0751587
548.2731645
615.755722
319.5605772
262.8152157
864.8866193
617.4171225
925.0998972
903.1375392
206.4649028
734.3442763
354.7014578
562.4827792
101.3492675
855.9097874
398.4975647
733.6510712
737.9617477
809.9507825
287.5062238
394.7740738
459.1368951
754.6327064
345.1321969
419.8283072
578.4755627
263.7579827
584.9763192
37.50877195
926.5681888
254.425745
59.16333354
235.9075284
539.2551248
931.201296
189.8706725
520.1077837
898.2088159
734.1092436
399.2997436
975.6835325
751.7324464
127.9442425
974.128563
835.939599
663.9471146
268.1450586
704.7201729
600.3964355
603.644137
686.8335523
530.7479388
202.2522575
671.8247557
497.6540574
520.953704
460.9825503
344.4913609
589.4647352
850.5628895
913.0067443
291.2841543
40.22807832
32.97310875
921.6802409
215.121714
872.4166373
340.6639945
760.8318233
516.9075965
148.7939701
652.6766589
988.1316923
229.7909383
299.1567523
487.8411394
614.2222293
527.7876892
923.156141
244.5486029

Loading Positions sample file. The first value gives the number of loading positions. Each row then gives the loading position index, and its x,y co-ordinates relative to the ship's centre

120
1 -33 7.5
2 -33 4.5
3 -33 1.5
4 -33 -1.5
5 -33 -4.5
6 -33 -7.5
7 -27 7.5
8 -27 4.5
9 -27 1.5
10 -27 -1.5
11 -27 -4.5
12 -27 -7.5
13 -21 7.5
14 -21 4.5
15 -21 1.5
16 -21 -1.5
17 -21 -4.5
18 -21 -7.5
19 -15 7.5
20 -15 4.5
21 -15 1.5
22 -15 -1.5
23 -15 -4.5
24 -15 -7.5
25 -9 7.5
26 -9 4.5
27 -9 1.5
28 -9 -1.5
29 -9 -4.5
30 -9 -7.5
31 -3 7.5
32 -3 4.5
33 -3 1.5
34 -3 -1.5
35 -3 -4.5
36 -3 -7.5
37 3 7.5
38 3 4.5
39 3 1.5
40 3 -1.5
41 3 -4.5
42 3 -7.5
43 9 7.5
44 9 4.5
45 9 1.5
46 9 -1.5
47 9 -4.5
48 9 -7.5
49 15 7.5
50 15 4.5
51 15 1.5
52 15 -1.5
53 15 -4.5
54 15 -7.5
55 21 7.5
56 21 4.5
57 21 1.5
58 21 -1.5
59 21 -4.5
60 21 -7.5
61 -33 7.5
62 -33 4.5
63 -33 1.5
64 -33 -1.5
65 -33 -4.5
66 -33 -7.5
67 -27 7.5
68 -27 4.5
69 -27 1.5
70 -27 -1.5
71 -27 -4.5
72 -27 -7.5
73 -21 7.5
74 -21 4.5
75 -21 1.5
76 -21 -1.5
77 -21 -4.5
78 -21 -7.5
79 -15 7.5
80 -15 4.5
81 -15 1.5
82 -15 -1.5
83 -15 -4.5
84 -15 -7.5
85 -9 7.5
86 -9 4.5
87 -9 1.5
88 -9 -1.5
89 -9 -4.5
90 -9 -7.5
91 -3 7.5
92 -3 4.5
93 -3 1.5
94 -3 -1.5
95 -3 -4.5
96 -3 -7.5
97 3 7.5
98 3 4.5
99 3 1.5
100 3 -1.5
101 3 -4.5
102 3 -7.5
103 9 7.5
104 9 4.5
105 9 1.5
106 9 -1.5
107 9 -4.5
108 9 -7.5
109 15 7.5
110 15 4.5
111 15 1.5
112 15 -1.5
113 15 -4.5
114 15 -7.5
115 21 7.5
116 21 4.5
117 21 1.5
118 21 -1.5
119 21 -4.5
120 21 -7.5