Using Genetic Algorithms To Solve The Yard Allocation Problem

Ping Chen
Department of Computer Science
National University of Singapore
Singapore 117543
chenp@comp.nus.ed.sg

Zhaohui Fu
Department of Computer Science
National University of Singapore
Singapore 117543
fuzh@comp.nus.ed.sg

Andrew Lim
Department of Computer Science
National University of Singapore
Singapore 117543
alim@comp.nus.ed.sg

Abstract

The Yard Allocation Problem (YAP) is a real-life resource allocation problem faced by the Port of Singapore Authority (PSA). We first show that YAP is NP-Hard. As the problem is NP-Hard, we propose a Genetic Algorithm approach. For benchmarking purposes, Tabu Search and Simulated Annealing are applied to this problem as well. Extensive experiments show very favorable results for the Genetic Algorithm approach.

1 INTRODUCTION

Singapore has one of the world's busiest ports in terms of shipping tonnage with more than one hundred thousand ship arrivals every year. One of the major logistical problems encountered is to use the minimum container yard necessary to accommodate all different requests. Each request consists of a single time interval and a series of yard space requirements during the interval. An interesting constraint applying to every request is that the length of the required space can either increase or remain unchanged as time progresses, and once yard space is allocated to a certain request, that portion of the yard space cannot be freed until the completion of the request. The current allocation is made manually, hence it requires a considerable amount of manpower.

This paper is organized in the following way. Section 2 gives a formal problem definition. This geometrical problem is then transformed into a graph problem in Section 3. For benchmarking purpose, we briefly discuss two heuristics, namely Tabu Search and Simulated Annealing, applied on YAP in Section 4 and 5 respectively. Section 6 illustrates our application of Genetic Algorithms on YAP in details, and various genetic operators are presented in this section. Section 7 compares the different experimental results obtained by those three heuristics. In Section 8, we present our conclusion.

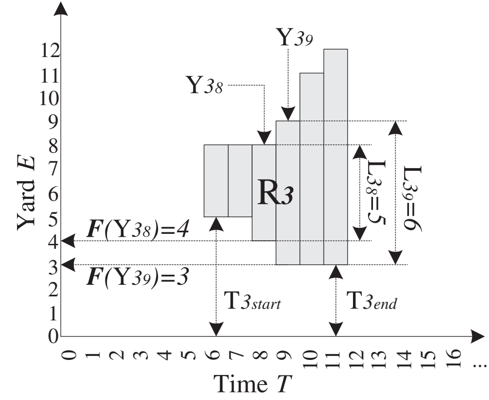

Figure 1: A *valid* request R_3.

2 PROBLEM DEFINITION

The main objective of the Yard Allocation Problem (YAP) is to minimize the container yard used while satisfying all the space requirements. The formal definition of the problem can be described as follows:

Instance: A set R of n yard space requests and an infinite container yard E. $\forall R_i \in R, R_i$ has series of (continuous) space requirements Y_{i_j} with length $L_{i_j}, j \in [T_{i_{start}}, T_{i_{end}}]$.
Output: A mapping function F, such that $F(Y_{i_j}) = e_k$, where $e_k \in E$ is some position on E.
Constraint: $\forall p, q \in [T_{i_{start}}, T_{i_{end}}]$ such that $p = q - 1$, $F(Y_{i_p}) \geq F(Y_{i_q})$ and $F(Y_{i_p}) + L_{i_p} \leq F(Y_{i_q}) + L_{i_q}$.
Objective: To minimize:

$$\max_{\forall R_i \in R, \forall Y_{i_j} \in R_i} (F(Y_{i_j}) + L_{i_j})$$

In other words, the objective is to accommodate all requests

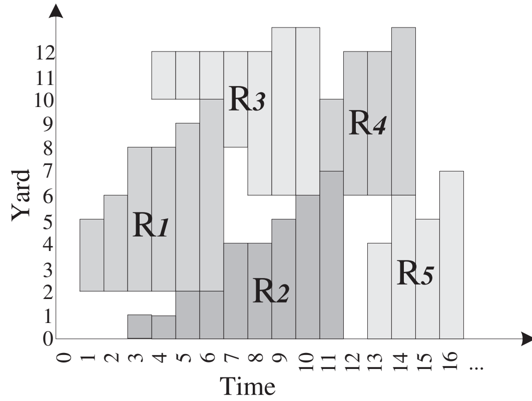

Figure 2: Five *valid* requests on yard

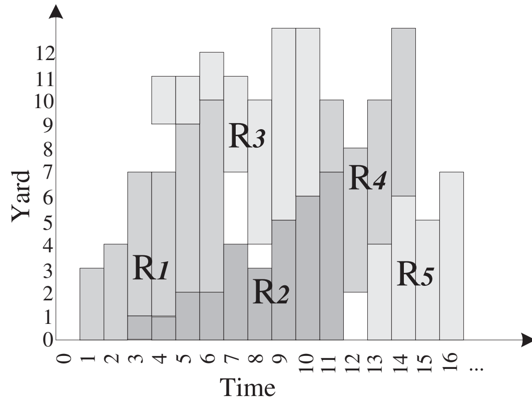

Figure 3: Five *invalid* requests on yard

with the minimum amount of yard space used.

We use an example to illustrate the definition. Figure 1 shows a layout with only one *valid* requests R_3. The yard E is treated as an infinite straight line. Time T becomes a discrete variable with a minimum unit of 1. R_3 has six space requirements within interval $[6, 11]$ ($T_{3_{start}} = 6, T_{3_{end}} = 11$). The final position for Y_{3_8} and Y_{3_9} are $F(Y_{3_8}) = 4$ and $F(Y_{3_9}) = 3$ respectively. The corresponding output for R_3 will then be $(5, 5, 4, 3, 3, 3)$. Note all our pre-defined constraints hold as $F(Y_{3_8}) \geq F(Y_{3_9})$ and $F(Y_{3_8}) + L_{3_8} \leq F(Y_{3_9}) + L_{3_9}$. The max comes from $Y_{3_{11}}$ with the value of $F(Y_{3_{11}}) + L_{3_{11}} = 12$.

We simply call each request a Stair Like Shapes (SLS) throughout this paper. Figure 2 shows five *valid* requests with the minimum yard required of 13. Though the packing in Figure 3 looks more compact, in fact, all allocations are *invalid* as the containment constraint is violated.

Theorem 1 *The Yard Allocation Problem (YAP) is NP-Hard.*

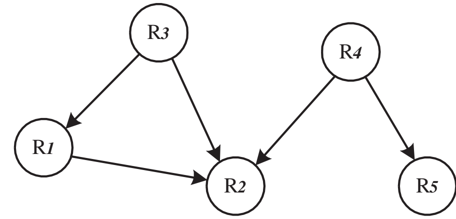

Figure 4: Graph Transformation of Figure 2

The Ship Berthing Problem (SBP) was first introduced in [Lim98]. The SBP has a similar configuration except all the requests are of rectangular shape instead of SLS. [Lim99] has provided an NP-Hard proof for SBP by reducing the Set Partitioning Problem to SBP. As SBP is special case of YAP and YAP is in the class NP, YAP is NP-Hard.

3 GRAPH TRANSFORMATION

Figure 2 illustrates the problem geometrically. However, the direct model may not be efficiently manipulated. We first transform the geometrical layout into a graph. Figure 4 is the corresponding graph transformation of the configuration in Figure 2. Each request R_i is represented by a vertex and there exists an edge E_{ij} connecting R_i and R_j iff R_i and R_j have an overlap at some time. The direction of the edge determines the relative position of the two requests in the physical yard. Take Figure 2 again as an example, both R_1 and R_2 require some space at time 3,4,5 and 6, therefore in Figure 4 there is an edge between R_1 and R_2. Since R_1 is located above R_2, the direction of edge is from R_1 to R_2. We name this edge E_{12}. Clearly, the transformed graph is a Direct Acyclic Graph (DAG). In a DAG, each vertex R_i can be assigned an Acyclic Label (AL) L_i and the edge E_{ij} implies $AL(R_i) < AL(R_j)$. Note that each $AL(R_i)(1 \leq i \leq n)$ is unique.

Lemma 1 *For each feasible layout of the yard, there exists at least one corresponding AL assignment of the vertices in the graph representation.*

A simple constructive proof can be obtained by the well-known Topological Sort algorithm. An AL assignment can also be interpreted as a permutation of $1, 2, \ldots, n$.

A "free" SLS is the one with no other SLS above it, i.e. there is no obstacle blocking it from being popped out from the top of the layout. Again, use Figure 2 as example. At the first iteration of the loop, R_3 and R_4 are the only two "free" SLSs. If we assign $AL(R_3) = 0$, in the second

iteration, R_1 will become a new "free" SLS. The process continues until no more SLS is left in L.

The AL assignment only has the partial order property. Each physical layout may correspond to more than one AL assignments due to the lack of total order property. $[R_1 : 2, R_2 : 3, R_3 : 0, R_4 : 1, R_5 : 4]$ and $[R_1 : 2, R_2 : 4, R_3 : 1, R_4 : 0, R_5 : 3]$ are two possible AL assignments.

This one-to-many relationship between physical layout and AL assignments in the graph representation will incur a huge amount of confusion in heuristic searches, including Genetic Algorithms, etc. Heuristic methods tend to identify certain *good* patterns which may potentially lead to a better solution while exploring the search space. Two very different looking solutions, which may actually correspond to the same physical layout, will make it very difficult for the heuristic to indentify the correct patterns.

We can avoid such confusion by normalizing the AL assignment. When there are more than one SLSs to be popped out, we break the tie by selecting the SLS with the smallest label. Each un-normalized AL assignment is used to construct the corresponding DAG. Then a Topological Sort with above-mentioned tie-breaker will give the *unique* AL assignment. From this point onwards, all our solutions are represented by their normalized unique AL assignments.

Each physical layout now has a unique AL assignment. Naturally, the optimal layout has an optimal AL assignment. Our goal is to find out such an optimal AL assignment. One of the major operations, the evaluation of a given AL assignment, turns out to be non-trivial. In SBP [Lim98] [FL00], a longest path algorithm on a DAG was used to find the minimum berth length needed. However, YAP deals with SLS, whose relative position and distance cannot be calculated in a straight-forward way, unlike rectangles. We have to use a recursive procedure to find the minimum yard needed for a given AL assignment A.

Evaluate-Solution (A)
1 **while** exists unallocated SLS
2 pick SLS S with largest AL
3 Drop($S, S_{end}, 0$)
4 **foreach** time T_i
5 **if** $T_i > L$
6 $L := T_i$
7 **return** L

Drop (S, t, l)
1 L :=lowest position to drop all stairs (time t')
2 **if** $L < l$
3 $L = l$
4 **forall** stair s after $t' - 1$
5 drop s to position L
6 Drop($S, t' - 1, L$)

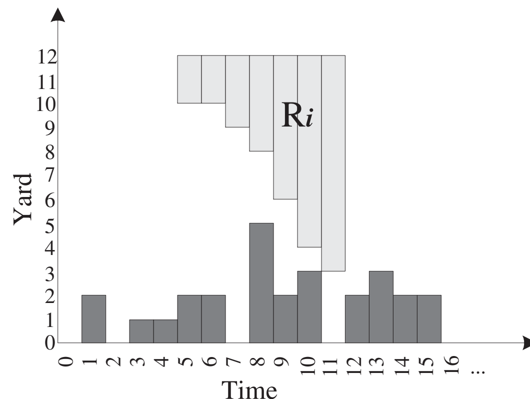

Figure 5: Before dropping: R_i is ceiling aligned

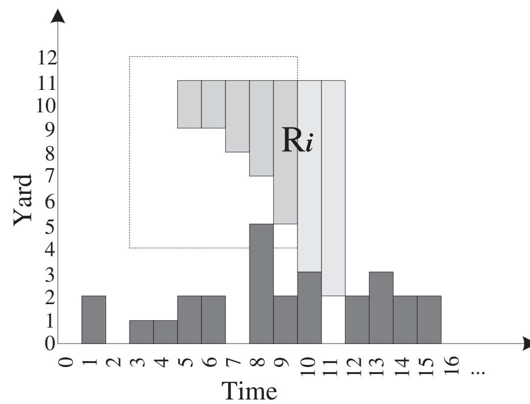

Figure 6: Each stair of R_i drop by 1. Stairs at time 10 and 11 are in their final positions. Those stairs which can drop further are in dark color surrounded by a rectangle

The recursive function $Drop$ uses a greedy approach to drop a given SLS to a position as low as possible. We illustrate the details through Figures 5, 6 and 7: R_i has seven space requirements starting from time 5 till 11. R_i is first aligned to the ceiling before the process starts (Figure 5). Then from time 5 to 11, we find the maximum distance that each "stair" can drop, without exceeding a lower bound of 0. The minimum amongst all the maximum possible drop is used. In this case, the minimum distance of 1 is given at time 10 and hence every stair is shifted down by 1 (Figure 6). Because of the initial ceiling alignment, no further shifting down is needed for all stairs at time 10 (inclusive) onwards. Note that the surface was touched at time 10.

Stairs from time 5 to 9, which are surrounded by a rectangle in Figure 6, can still be dropped further but this time with a lower bound of 3, which is the height of the pre-

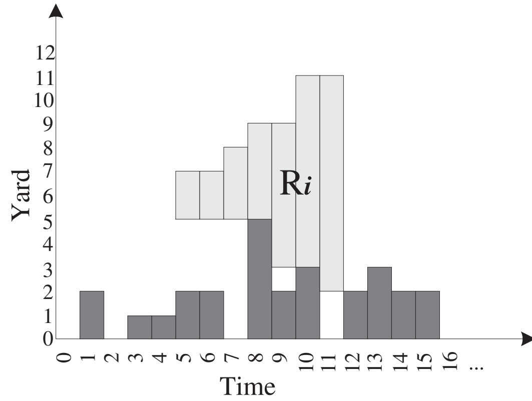

Figure 7: Final layout

vious touching surface at time 10. The dropping process completes after a few more recursions at time 8,7,6 and 5. The final layout is shown by Figure 7. Note the worst case time complexity for $Drop$ is $n \times T$, where n is the number of requests and T is the average time span for all requests.

Lemma 2 *For a given AL assignment, the greedy dropping approach always returns the layout with minimum yard used.*

Proof. The proof of the correctness of a greedy algorithm consists of two parts: First, the greedy choice always leads to an optimal solution, or any optimal solution can be transformed into a solution obtained by the greedy choice. Second, the problem has an optimal sub-structure, i.e. the global optimal implies local optimal. The optimal substructure property is obvious for YAP. To show the greedy choice property, we compare the solution G obtained by greedy dropping approach with any arbitrary optimal solution O. Consider the following algorithm:

Compact (A, G, O)
1 **let** $L :=$ set of SLSs;
2 **while** L is not empty
3 pick SLS S with largest AL
4 **for** $(i = S_{begin}; i \leq S_{end}; i++)$
5 **let** $G_{s_i} :=$ position of S_i in G
6 **let** $O_{s_i} :=$ position of S_i in O
7 **if** $O_{s_i} > G_{s_i}$
8 $O_{s_i} := G_{s_i}$

The algorithm $Compact$ will transform any optimal solution into a corresponding solution that can be obtained by the greedy approach without increase the amount of the yard used. Note line 7 is based on the fact that no optimal solution can allocate S_i in a lower position than greedy approach.

Up to now, we have built a one-to-one relationship between physical layout and the AL assignment $(0, 1, \ldots, n-1)$. The problem is to find the optimal AL assignment.

4 TABU SEARCH

Tabu Search [GL97] [Ham93] is a local search metaheuristic. According to the different usage of memory, conventionally, Tabu Search has been classified into two categories: Tabu Search with Short Term Memory (TSSTM) and Tabu Search with Long Term Memory (TSLTM) [GL97] [SY99]. Tabu Search can also be hybridized with other heuristics, like Squeaky Wheel Optimization [CFL02].

4.1 TABU SEARCH WITH SHORT TERM MEMORY

Our TSSTM implementation consists of two major components: neighborhood search and the tabu list. The neighborhood solution can be obtained by swapping any two ALs in the AL assignment. For example:

$$(2\ 3\ 0\ 1\ 4) \rightarrow (1\ 3\ 0\ 2\ 4)$$

by interchanging the positions of 1 and 2. However, certain swaps, after normalization, may be identical to the original AL assignment. Such solutions are excluded from the neighborhood for efficiency.

Our solution is represented in an AL assignment, which is just a series of numbers. Due to this simplicity, our Tabu List is implemented to record the whole AL assignment for a certain number of solutions recently visited. To be more efficient, string matching algorithms are used to identify the *tabu active* solutions.

4.2 TABU SEARCH WITH LONG TERM MEMORY

We implemented TSLTM in two phases: Diversification and Intensification. We used two kinds of diversification techniques, one is a random re-start and the other is to randomly pick a sub-sequence and insert it into a random position. For example:

$$(0 \mid 1\ 2 \mid 3\ 4) \rightarrow (0\ 3\ 4 \mid 2\ 1 \mid)$$

if $(\mid 1\ 2 \mid)$ is chosen as the sub-sequence and its inverse (or original, if random) is inserted at the back. Intensification is similar to TSSTM. TSLTM uses a *frequency* based memory by recording both *residence* frequency and *transition* frequency of the visited solutions. In our implementation, residence frequency is taken as the number of times that the $AL(R_i) < AL(R_j), 1 \leq i, j \leq n$ in the selected solution

in each iteration. The transition frequency is taken as the summation of the improvements when $AL(R_i)$ is swapped with $AL(R_j)$. The sum can be either positive or negative.

Diversification and Intensification are interleaved and during either phase, the residence frequency and transition frequency are updated according to the current selected solution. The objective function has three contributors. Besides the length of the yard space required, both residence frequency and transition frequency are used to evaluate the solution.

5 SIMULATED ANNEALING

Simulated annealing [Haj88], [KGV83], [OG89] is a very general optimization method which stochastically simulates the slow cooling of a physical system.

We used the following Simulated Annealing algorithm on our problem:

Step 1. Choose some initial temperature T_0 and a random initial starting configuration θ_0. Set $T = T_0$. Define the Objective function (Energy function) to be $En()$ and the cooling schedule σ.

Step 2. Propose a new configuration, θ' of the parameter space, within a neighborhood of the current state θ, by setting $\theta' = \theta + \phi$ for some random vector ϕ.

Step 3. Let $\delta = En(\theta') - En(\theta)$. Accept the move to θ' with probability

$$\alpha(\theta, \theta') = \begin{cases} 1 & \text{if } \delta < 0 \\ exp(-\frac{\delta}{T}) & \text{otherwise} \end{cases}$$

Step 4. Repeat Step 2 and 3 for K of iterations, until it is deemed to have reached the equilibrium.

Step 5. Lower the temperature by $T = T \times \sigma$ and repeat Steps 2-4 until certain stopping criterion, for our case $T < \epsilon$, is met.

Due to the logarithmic decrement of T, we set $T_0 = 1000$. The Energy function is simply defined as the length of the yard required. The probability $exp(-\frac{\delta}{T})$ is known as the Boltzmann factor. The number of iterations K is proportional to the input size n. The neighborhood is defined similarly as the one in Tabu Search, which are swapping of any two AL and re-positioning of a random AL subsequence.

6 GENETIC ALGORITHM

Genetic Algorithms [Hol75] are search procedures that use the mechanics of natural selection and natural genetics. It is clear that the classical binary representation is not a suitable in YAP, in which a list of Acyclic Labels $(0, 1, \ldots, n-1)$ is used as the solution representation. The solution space is a permutation of $(0, 1, \ldots, n-1)$. The binary codes of these AL do not provide any advantage. Sometimes the situation is even worse: the change of a single bit may not result in a valid solution. We adopt a vector representation, i.e. use the AL assignment directly as the chromosome in the genetic process. We will illustrate the two major genetic operators used in our approach, crossover and mutation.

6.1 CROSSOVER OPERATOR

Using AL assignment as chromosome, we have implemented three crossover operators:

- Classical crossover with repair.
- Partially-mapped crossover.
- Cycle crossover.

All these operators are be tailored to suit our problem domain. A tiny change in the crossover operator may act in totally different manners.

6.1.1 Classical Crossover with Repair

The Classical Crossover operator is the simplest among the three methods mentioned above. It builds the offspring by appending the head from one parent with the tail from the other parent, where the head and tail come from random cut of the parents' chromosomes. A repair procedure may be necessary after the crossover [Mic96]. For example, the two parents (with random cut point marked by '|'):

$$\begin{aligned} p_1 &= (0\,1\,2\,3\,4\,5 \,|\, 6\,7\,8\,9) \text{ and} \\ p_2 &= (3\,1\,2\,5\,7\,4 \,|\, 0\,9\,6\,8). \end{aligned}$$

will produce the following two offsprings:

$$\begin{aligned} o_1 &= (0\,1\,2\,3\,4\,5 \,|\, 0\,9\,6\,8) \text{ and} \\ o_2 &= (3\,1\,2\,5\,7\,4 \,|\, 6\,7\,8\,9). \end{aligned}$$

However, the two offsprings are not valid AL assignments after the crossover. A repair routine replaces the repeated ALs with the missing ones randomly. The repaired offsprings will be:

$$\begin{aligned} o_1 &= (7\,1\,2\,3\,4\,5 \,|\, 0\,9\,6\,8) \text{ and} \\ o_2 &= (3\,1\,2\,5\,7\,4 \,|\, 6\,0\,8\,9). \end{aligned}$$

The classical crossover operator tries to maintain the absolute AL positions in the parents.

6.1.2 Partially Mapped Crossover

Partially Mapped Crossover (PMX) was first used in [GL85] to solve the Traveling Salesman Problem (TSP). We have made several adjustments to accommodate our chromosome (AL assignment) representation. The modified PMX builds an offspring by choosing a subsequence of an AL assignment from one parent and preserving the order and position of as many ALs as possible from the other parent. The subsequence is determined by choosing two random cut points. For example, the two parents:

$$\begin{aligned} p_1 &= (0\ 1\ 2 \mid 3\ 4\ 5\ 6 \mid 7\ 8\ 9) \text{ and} \\ p_2 &= (3\ 1\ 2 \mid 5\ 7\ 4\ 0 \mid 9\ 6\ 8). \end{aligned}$$

would produce offspring as follows. First, two segments between cutting points are swapped (symbol 'u' represents 'unknown' for this moment):

$$\begin{aligned} o_1 &= (u\ u\ u \mid 5\ 7\ 4\ 0 \mid u\ u\ u) \text{ and} \\ o_2 &= (u\ u\ u \mid 3\ 4\ 5\ 6 \mid u\ u\ u). \end{aligned}$$

The swap defines a series of mappings implicitly at the same time:

$$3 \leftrightarrow 5, 4 \leftrightarrow 7, 5 \leftrightarrow 4 \text{ and } 6 \leftrightarrow 0.$$

The 'unknown's are then filled in with AL from original parents, for which there is no conflict:

$$\begin{aligned} o_1 &= (u\ 1\ 2 \mid 5\ 7\ 4\ 0 \mid u\ 8\ 9) \text{ and} \\ o_2 &= (u\ 1\ 2 \mid 3\ 4\ 5\ 6 \mid 9\ u\ 8). \end{aligned}$$

Finally, the first u in o_1 (which should be 0, who will cause a conflict) is replaced by 6 because of the mapping $0 \leftrightarrow 6$. Note such replacement is transitive, for example, the second u in o_1 should follow the mapping $7 \leftrightarrow 4, 4 \leftrightarrow 5, 5 \leftrightarrow 3$ and is hence replaced by 3. The final offspring are:

$$\begin{aligned} o_1 &= (6\ 1\ 2 \mid 5\ 7\ 4\ 0 \mid 3\ 8\ 9) \text{ and} \\ o_2 &= (7\ 1\ 2 \mid 3\ 4\ 5\ 6 \mid 9\ 0\ 8). \end{aligned}$$

The PMX crossover exploits important similarities in the value and ordering simultaneously when used with an appropriate reproductive plan [GL85].

6.1.3 Cycle Crossover

Original Cycle Crossover (CX) was proposed in [OSH87], again for the TSP problem. Our CX builds offspring in such a way that each AL (and its position) comes from one of the parents. We explain the mechanism of the CX with following example. Two parents:

$$\begin{aligned} p_1 &= (0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9) \text{ and} \\ p_2 &= (3\ 1\ 2\ 5\ 0\ 4\ 7\ 9\ 6\ 8). \end{aligned}$$

will produce the first offspring by taking the first AL from the first parent:

$$o_1 = (0\ u\ u\ u\ u\ u\ u\ u\ u\ u).$$

Since every AL in the offspring should come from one of its parents (for the same position), the only choice we have at this moment is to pick AL 3, as the AL from parent p_2 just "below" the selected 0. In p_1, it is in position 3, hence:

$$o_1 = (0\ u\ u\ 3\ u\ u\ u\ u\ u\ u).$$

which, in turn, implies AL 5, as the AL from p_2 "below" the selected 3:

$$o_1 = (0\ u\ u\ 3\ u\ 5\ u\ u\ u\ u).$$

Following the rule, the next AL to be inserted is 4. However, selection of 4 requires the selection of 0, which is already in the list. Hence the cycle is formed as expected.

$$o_1 = (0\ u\ u\ 3\ 4\ 5\ u\ u\ u\ u).$$

The remaining 'u's are filled from p_2:

$$o_1 = (0\ 1\ 2\ 3\ 4\ 5\ 7\ 9\ 6\ 8).$$

Similarly,

$$o_2 = (3\ 1\ 2\ 5\ 0\ 4\ 6\ 7\ 8\ 9).$$

The CX preserves the absolute position of the elements in the parent sequence [Mic96].

Our experiments shows Classical crossover and CX have a stable but slow improvement rate according to time while PMX demonstrates an oscillating but fast convergence trend. In our later experiments, majority of the crossover is done by PMX. Classical crossover and CX are applied at a much lower probabilities.

6.2 MUTATION OPERATOR

Mutation is another classical genetic operator, which alters one or more genes (portion of the chromosome) with a probability equal to the mutation rate. There are several known mutation algorithms which work well on different problems:

- Inversion: invert a subsequence.
- Insertion: select an AL and insert it back in a random position.
- Displacement: select a subsequence and insert it back in a random position.
- Reciprocal Exchange: swap two ALs.

In fact the Inversion, Displacement and Reciprocal Exchange are quite similar to our neighborhood solution and diversification techniques used in Tabu Search and Simulated Annealing in previous sessions. We adopt a relatively low mutation rate at 1%.

The population size $P = 1000$ is set for most cases. The evolution process starts with a random population. The population is sorted according to the objective function, the best the quality, the higher the probability it will be selected for reproduction. At each iteration, a new generation with population size $2P$ is produced and the better half, which is of size P, survive for the next iteration. The evolution process continues until certain stop criterion are met.

7 EXPERIMENTAL RESULTS

Table 1: Experimental results (Entries in the table shows the minimum length of the yard required. Name of Data Set shows the number of SLSs in the file; LB:Lower Bound)

Data Set	LB	TSSTM	TSLTM	SA	GA
R126	21	28	26	25	24
R117	34	39	37	34	34
R145	39	50	45	42	39
R178	50	69	69	66	55
R188	74	105	98	102	79
R173	77	98	91	98	79
R250	83	141	119	117	89
R236	97	139	130	114	101
R213	164	245	246	245	187

We conducted extensive experiments on randomly generated data[1] The graph for each test case contains one connected components, in other words, the test cases cannot be partitioned into more than one independent sub-case. Due to the difficulties of finding any optimal solution in the experiments, a trivial *lower bound* is taken to be the sum of the space requirements at each time slot and used for benchmarking purpose.

Table 1 illustrates the results. It is not surprising to see that GA outperforms all other heuristics in all test cases by a considerable margin. TSSTM has the simplest implementation with the worst results. TSLTM has an obvious improvement from TSSTM, though the improvement is not very stable. We believe one of the major difficulties with Long Term Memory is the assignment of relative weights to yard length, residence frequency and transition frequency in the objective function. SA is relatively easy to implement with comparable results to TSLTM. The most successful approach, using Genetic Algorithm, gives the best results, which are within 8% of the trivial lower bound at most of the time.

[1]All test data are available on the WWW with URL: http://www.comp.nus.edu.sg/˜fuzh/YAP.

Table 2: Experiment running time (seconds) for Table 1.

Data Set	TSSTM	TSLTM	SA	GA
R126	594	3423	2023	302
R117	753	2521	1873	442
R145	1215	3693	2233	784
R178	2568	5362	2576	1023
R188	2523	7822	3529	1321
R173	3432	6743	3431	1675
R250	4578	10239	5031	3632
R236	5027	11053	5892	2453
R213	4891	10476	6342	4322

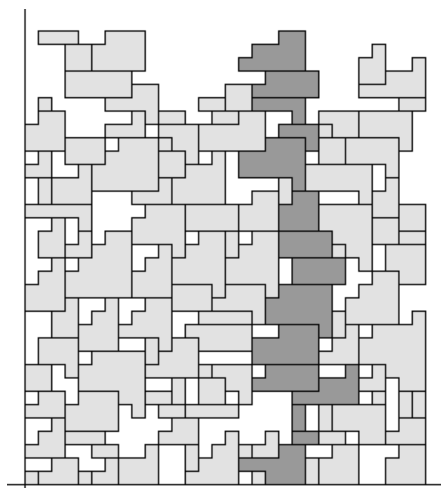

Figure 8: Physical layout of 117 SLSs (requests). Data Set: R117

Table 2 shows the running time for each of the test performed in Table 1 on a Dual-CPU (Pentium III 800MHz each) Linux machine. It is clear that GA is also the most cost-effective approach.

Another interesting discovery from the experiments is that the normalization routine does not improve the results of GA as much as our expectation. Besides the slowing down factor, it sometimes even degenerates the results. We believe that normalization should make the search process more stable and focus by removing the *confusing* factors. But its side effects, for example reducing the solution space, sometimes overwhelm its merits.

Figure 8 and Figure 9 provide the graphic results obtained by GA for input file R117 and R145. The heavily-shaded SLSs contain the region that is the densest (lower bound) in both figures. Due to the stair-like shapes, the packing layout looks sub-optimal. A closer look reveals further improvements to be very unlikely.

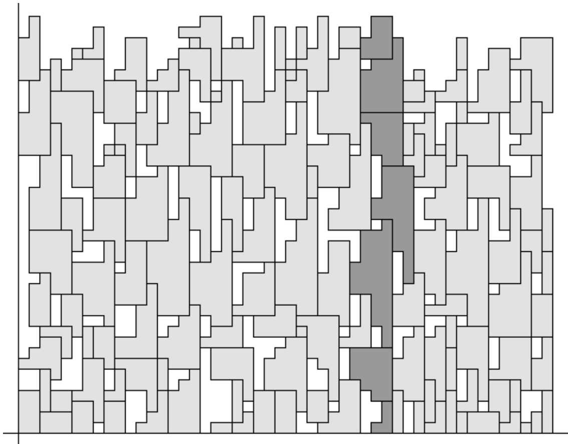

Figure 9: Physical layout of 145 SLSs (requests). Data Set: R145

8 CONCLUSION

In this paper, we have shown the Yard Allocation Problem (YAP) is NP-Hard by reducing the Ship Berthing Problem (SBP) to it. The geometrical representation of YAP is then transformed into a Direct Acyclic Graph (DAG) for efficient manipulation. A normalization procedure is proposed to guarantee a one-to-one relationship between geometric layout and Acyclic Label Assignment of the DAG. Finding the optimal layout is transformed to the search for the optimal Acyclic Label Assignment. Two heuristic methods, Tabu Search and Simulated Annealing are first applied. The results obtained are not very attractive.

A Genetic Algorithm approach is applied after TS and SA's failing to achieve good results against the lower bound. Various genetic operators are proposed and applied. Extensive experiments showed our GA approach, outperformed both the original Tabu Search and Simulate Annealing by a margin of more than 10%.

References

[CFL02] Ping Chen, Zhaohui Fu, and Andrew Lim. The yard allocation problem. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02), Edmonton, Alberta, Canada*, 2002.

[FL00] Zhaohui Fu and Andrew Lim. A hybrid method for the ship berthing problem. In *Proceedings of the Sixth Artificial Intelligence and Soft Computing*, 2000.

[GL85] D. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem, 1985.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Haj88] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13:311–329, 1988.

[Ham93] Peter L. Hammer. *Tabu Search*. Basel, Switzerland: J.C. Baltzer, 1993.

[Hol75] John H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[KGV83] S. Kirpatrick, C.D. Gelatt, Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[Lim98] Andrew Lim. On the ship berthing problem. *Operations Research Letters*, 22(2-3):105–110, 1998.

[Lim99] Andrew Lim. An effective ship berthing algorithm. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 594–599, 1999.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structure = Evolution Programs*. Springer-Verlag Berlin Heidelberg New York, 1996.

[OG89] Ralph H. J. M. Otten and Lukas P. P. P. van Ginneken. *The Annealing Algorithm*. Kluwer Academic Publishers, Boston, U.S.A., 1989.

[OSH87] I. Oliver, D. Smith, and J. Holland. A study of permutation crossover operators on the tsp, 1987.

[SY99] Sadiq Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE, 1999.