# Chapter 3

# Solving the Partitioning Problem

In this chapter, we discuss our graph partitioning solution for the BAP partitioning problem (henceforth termed as BAP-Part, to differentiate it from the entire BAP). We will show how to formulate the BAP-Part as a generalisation of the multi-way graph partitioning (GP) problem and describe our algorithm for solving the it.

## 3.1 A Generalised Graph Partitioning Model for the BAP Partitioning

Let node $v$ represent a vessel and $length(v)$ be the length of the vessel. $V = \{v_1, v_2, \ldots, v_n\}$ is a set of $n$ vessels in the BAP. An edge $c_{ij}$ captures the trans-shipment of $|c_{ij}|$ containers from $v_i$ to $v_j$. Let $s$ represent a section and $length(s)$ be the length of the section. $S = \{s_1, s_2, \ldots, s_m\}$ is a set of $m$ sections in the

BAP. $d_{kl}$ is the inter-section distance from $s_k$ to $s_l$. As stated in the assumptions in Section 2.2, this inter-section distance is estimated by taking the distance from the mid-point of $s_k$ to the mid-point of $s_l$.

We want to partition the nodes $V$ into $m$ disjoint parts $P_1, P_2, \ldots, P_m$ where each part $P_k$ denotes the set of vessels which are assigned to be berthed in $s_k$. Our objective in BAP-Part is to minimise the total *transshipment cost* (or the overall inter-section container traffic), and BAP-Part *objective function* is defined as

$$\text{Min} \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} d_{kl} \qquad \text{where } v_i \in P_k, v_j \in P_l \text{ and } c_{ii} = 0\ \forall i, d_{kk} = 0\ \forall k \tag{3.1}$$

The above objective function for the BAP-Part is a generalization of the GP objective function. In GP, the value of $d_{kl} = 1\ \forall k, l$, because the interpart distances are not considered. In the BAP, the actual distances travelled by the containers are important in determining the vessel turnaround time, hence just minimizing $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}$ is not enough as a container may end up travelling a long distance from one end of the port to the other end.

The constraint we have to satisfy is that the *part size* for $P_k$ cannot be larger than $length(s_k)$. In GP, the part size is computed by summing all the nodes that have been assigned to that part, ie. $\sum_{i=1}^{n} length(v_i) \leq length(s_k)\ \forall v_i \in P_k$. However, in the BAP-Part, computing the part size is not so trivial.

To see why this is so, consider Figure 3.1. In section 1 ($s_1$), which is of length 200m, the vessel 9 ($v_9$) also of length 200m is berthed for the first 6 hours. During

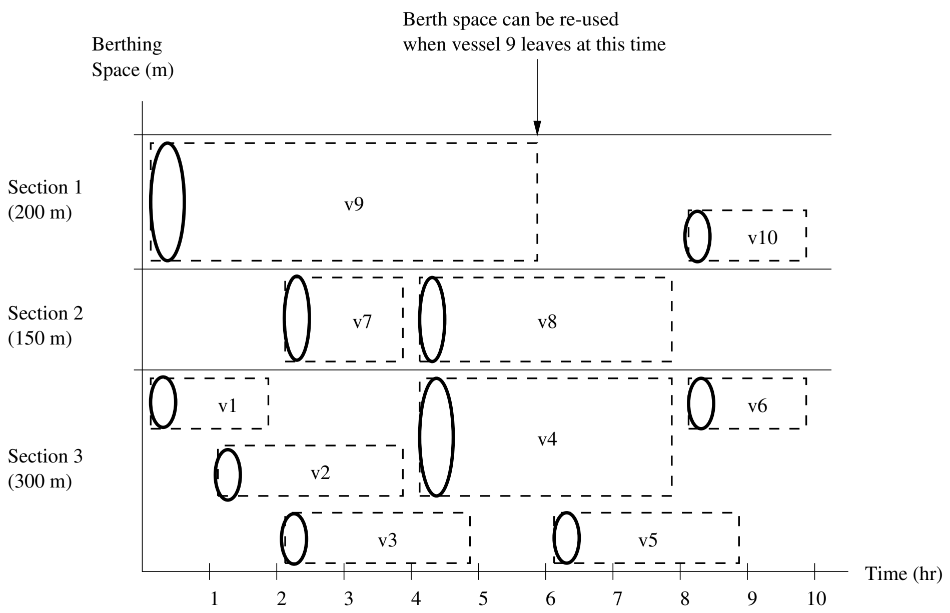


Figure 3.1: Computing Part Size

this time, $s_1$ is full and no other vessel may be berthed in it. This usage of the berth resource is represented schematically by the rectangle for $v_9$. But after $v_9$ leaves the port at the 6th hour, $v_{10}$ of length 100m which arrives at hour 8 can be berthed into $s_1$ by "re-using" part of the space formerly occupied by $v_9$. Hence, computing the part size for $s_1$ as in a standard GP problem results in $200m + 100m = 300m$ which is $> length(s_1) = 200m$. This gives the indication that $v_9$ and $v_{10}$ cannot be assigned to $s_1$ together, which is incorrect. The correct way to compute the part size for $s_1$ is to calculate the density of all vessels assigned to it.

We define

$$density(P_k) = max\{L_t\} \qquad \forall t = 1, \ldots, T \tag{3.2}$$

$$\begin{aligned} \text{where } T &= \text{number of hours in planning time horizon} \\ L_t &= \textstyle\sum_{i=1}^{n} x_{ik} y_{it} length(v_i) \end{aligned}$$

For our example, the density yields a value of $200m$ which is $\leq length(s_1)$. Our constraint is thus

$$density(P_k) \leq length(s_k) \qquad \forall k = 1, \ldots, m \tag{3.3}$$

In other words, if we view GP from the perspective of a BAP generalisation, we can treat all the nodes in GP as occupying a single time zone.

Another subtle difference between the BAP-Part and GP is that in GP, empty parts are generally not desired. This is particularly so in a graph bi-partition because it will result in the trivial solution of assigning all the nodes into one part, which would not be meaningful as the aim is to separate the nodes into 2 parts while minimizing interpart connections. However, for the BAP, we allow empty parts because a section can be unused if the remaining sections can accommodate all the vessels. In fact, we encourage empty parts in the BAP-Part! If we can assign all vessels into a single section, then we have effectively obtained the minimum objective value of 0 transshipment cost. In the real world, this means that all transshipment operations take place in close proximity within a single section, speeding up the transshipment process.

| | **BAP** | **GP** |
|---|---|---|
| Obj Function | Min $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}d_{kl}$ | Min $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}$ |
| Interpart Distance | Important | Not considered ($d_{kl} = 1$) |
| Time Dimension | Dynamic Vessels come and go | Non-existent Nodes are always present |
| Part Size | Because of time dimension, computed by vessel density | Computed by summing all node sizes |
| Empty Parts | Actively encouraged | Generally not desired |

Table 3.1: Differences between BAP-Part and Graph Partitioning

We define a *feasible solution* for the BAP-Part as one in which every vessel has been assigned to a part or $\bigcup_{k=1}^{m} P_k = V$. In this case, the number of unassigned vessels = 0. In the event when constraint (3.3) cannot be satisfied, we will end up with an *infeasible solution* where the number of unassigned vessels > 0. In this case, we attempt to find the best infeasible solution with the least number of unassigned vessels. We will describe how this can be done in Section 3.3.2 when we deal with the penalty cost function in our algorithm.

Table 3.1 summarises the differences between the BAP-Part and the GP problems. Here we see that BAP-Part is a generalisation of GP with two additional dimensions: the interpart distances and the time dimension.

### 3.1.1 Introducing Time Zones

Consider Figure 3.1 again. To determine if we can assign $v_6$ to $s_3$, assuming that $v_1, v_2, v_3, v_4$ and $v_5$ have already been assigned to $s_3$, we need to check for overlaps

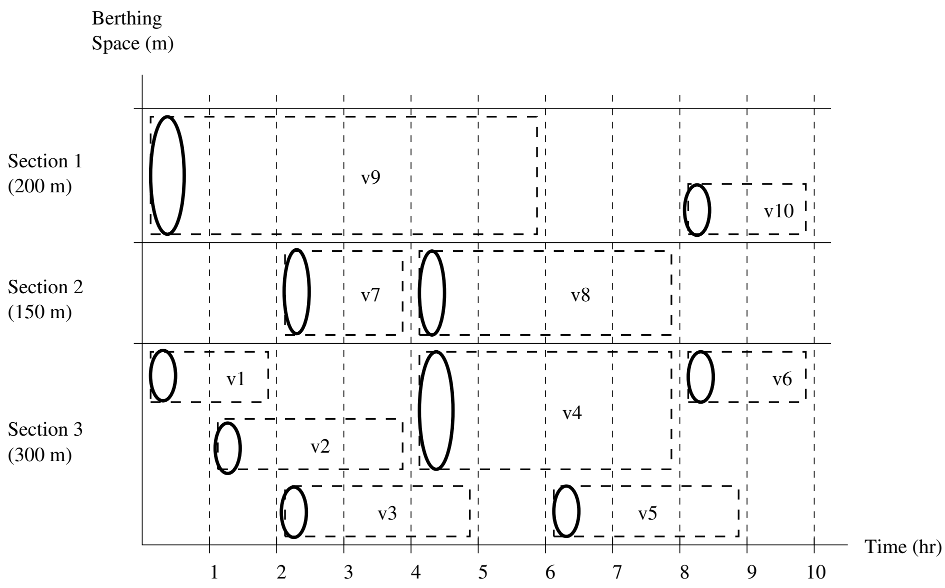


Figure 3.2: Dividing into 10 Time Zones

between the rectangle of $v_6$ and those of $v_1, \ldots, v_5$. If any overlap occurs and we cannot shift $v_6$ vertically up/down (horizontal shifts are not allowed as the arrival time and departure time are fixed) into a free space large enough to accommodate it, then $v_6$ cannot be assigned into $s_3$.

Looking at the figure, we see that only $v_5$ has a chance of overlapping with $v_6$. $v_1, \ldots, v_4$ would have left the port by the time $v_6$ arrives, so they would have no direct impact on $v_6$. We can say that $v_1, \ldots, v_4$ exist in different time zones from $v_6$, while $v_5$ and $v_6$ overlap in time zone(s). In other words, when trying to assign $v_6$, we need to check for overlap with $v_5$ only. Going along this idea, we can reduce the number of comparisons among vessels when checking for overlaps by dividing the schedule of vessels into time zones.

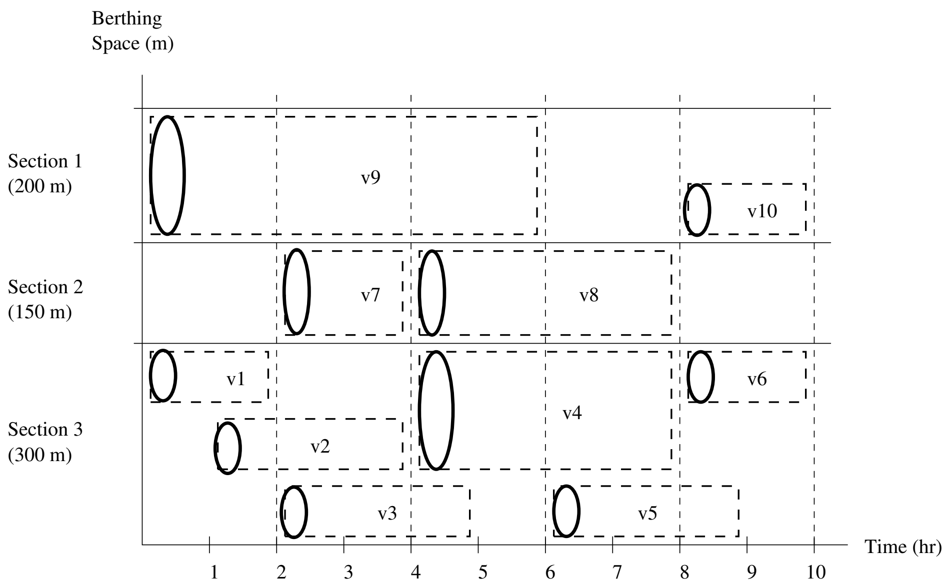


Figure 3.3: Minimal Set of Time Zones

We state the assumption that our model for BAP-Part assumes all vessels arrive and depart on the hour, and not in between the hour. We discretise the time axis into intervals of 1-hour.

One extreme case would be to take each 1-hour time interval as a time zone, as shown in Figure 3.2, resulting in 10 1-hour time zones. The other extreme is to assume all vessels fall within 1 10-hour time zone as we have seen in Figure 3.1. However, some time zones can be subsumed into other time zones. For example, $v_6$ occupies time zones 9 and 10 in Figure 3.2, but time zone 10 can be subsumed into time zone 9 because if we can ascertain that $v_6$ does not overlap with $v_5$ in time zone 9, this will also be true for time zone 10. Alternatively, we can look at time zone 10 as having a subset of the vessels in time zone 9. Similarly, the vessels in time zone 1 is a subset of the vessels in time zone 2.

The minimal set of time zones for the schedule is shown in Figure 3.3 where we have 5 2-hour time zones. For every time zone in this minimal set, the vessels in it will not be a subset of the vessels in another time zone. This allow us to minimise the number of comparisons when checking for vessel overlaps by comparing only those vessels that fall within the same time zones as the vessel to be assigned. The following plane sweep algorithm finds the minimal set of time zones in $O(nlogn)$ time in the discretised time schedule:

**Plane Sweep Algorithm To Find Minimal Set Of Time Zones**

1. Initialise array of time points $TP[2n]$. Set of time zones $TZ = \emptyset$.

2. For $i = 1$ to $n$ do

   (a) $TP[i]$ = arrival-time($v_i$).

   (b) $TP[i + n]$ = departure-time($v_i$).

3. Sort array $TP$ in non-descending order of time. Break ties by placing departure times before arrival times.

4. $st = TP[1]$, $depart$ = false.

5. For $i = 2$ to $2n$ do

   (a) If $TP[i]$ is departure time, then $depart$ = true.

   (b) If $TP[i]$ is arrival time and $depart ==$ true, then $TZ = TZ \bigcup \{[st, TP[i]]\}$. Start new time zone with $st = TP[i]$, $depart$ = false.

When the algorithm terminates, set $TZ$ holds the minimal set of time zones in [start-time, end-time] intervals. This minimal set of time zones will be used in Section 3.2.4.

## 3.2 Algorithm for Solving BAP-Part

We present the following *BAP-GP* algorithm for solving the BAP-Part:

**BAP-GP — BAP Partitioning Algorithm**

1. Generate an initial solution[1].
2. $\forall$ vessel $v$, set $v$ as free (unlocked) and compute its initial gain.
3. While $\exists$ free vessels do
   (a) Select $v$ with maximum gain and that is feasible to be moved.
   (b) If $v$ is not found, then goto 4.
   (c) Move $v$ and lock it.
   (d) Update gains of $v$ and of all neighbouring vessels to $v$.
4. Compute partial sum gain (starting with 0) across all moves made.
5. Undo moves made from the last move to the move yielding the highest partial sum $\geq 0$.
6. If highest partial sum $> 0$, then goto 2.

The initial solution of assigning vessels to sections can be generated randomly or in some determined order: assigning vessels from largest size to smallest size;

[1] Note that the initial solution need not be feasible.

assigning vessels from highest transshipment to smallest transshipment; etc. Different initial solutions will give rise to different solution quality. This dependency on the initial solution is a characteristic of the FM-family heuristic. The different methods we used to generate initial solutions are listed in Section 3.4.1 and we will look more into their relative performances in Section 5.2 on experimental results.

The *gain* of $v_i$ is defined to be the decrease in transshipment cost as a result of moving $v_i$. If moving $v_i$ results in an increase in transshipment cost, then its gain is negative. A vessel $v_j$ is said to be a *neighbouring vessel* to $v_i$ if $c_{ij} > 0$ or $c_{ji} > 0$.

Steps 2 to 5 make up one *pass* of the algorithm. The algorithm will iterate through several passes until it finally settles down on a locally optimal solution. The running time of each pass is dependent on how the gain computation and updates are performed, and on how the candidate vessel to be moved is selected. We'll explain next how the gain computation and updates are performed and how the data structures used support efficient selection of the candidate vessel and the updating of gains.

### 3.2.1 Gain Computation

We now explain how to compute the gain.

Let $g_{ik}$ = gain of moving vessel $i$ to part $P_k$.

Let $C_{ik} = \sum_{j=1}^{n} c_{ij} \qquad \forall v_j \in P_k$

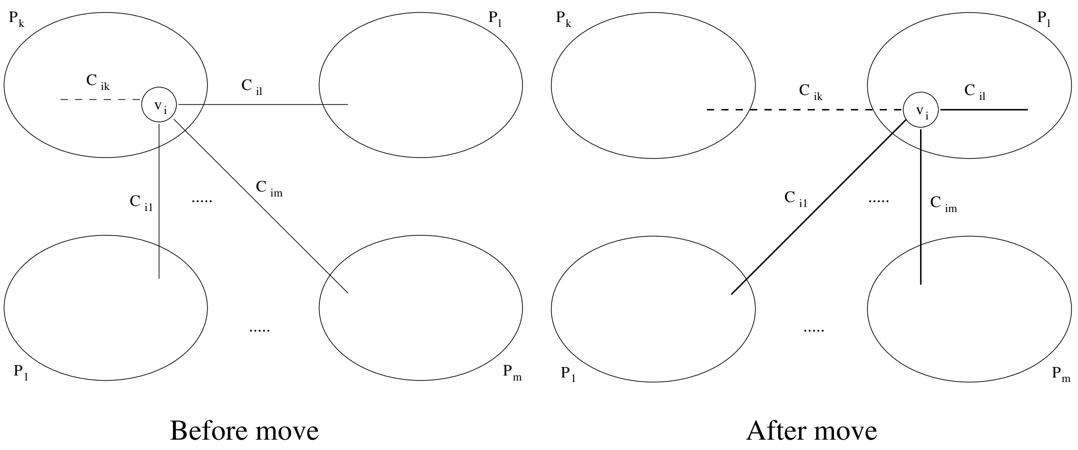


Figure 3.4: Gain Computation

$C_{ik}$ is the total transshipment or total connection between $v_i$ and all the vessels in $P_k$. If $v_i \in P_k$, then $C_{ik}$ is termed an *internal connection*. If $v_i \notin P_k$, then $C_{ik}$ is termed an *external connection*. Internal connections do not contribute to the transshipment cost; *only external connections contribute to the transshipment cost*. Let $d_{kl}$ = distance between $s_k$ and $s_l$. Note that $d_{kl} = d_{lk}$. We also define $d_{kk} = 0$.

Consider Figure 3.4. $v_i$ is originally in $P_k$ and we want to compute $g_{il}$. The left diagram shows the state of the graph prior to making the move. The solid lines $C_{il}, C_{i1}, \ldots, C_{im}$ represent the *external connections* between $v_i$ and $P_l, P_1, \ldots, P_m$ respectively. The dashed line $C_{ik}$ represents the *internal connection* between $v_i$ and $P_k$. The right diagram shows the state after making the move, in which case the line $C_{il}$ becomes an internal connection and the line $C_{ik}$ becomes an external connection.

Prior to the move, the total transshipment cost

$$
\begin{aligned}
TC &= X + C_{il}d_{kl} + C_{i1}d_{k1} + \ldots + C_{im}d_{km} && (3.4)\\
&= X + C_{il}d_{kl} + \sum_{r=1, r\neq k,l}^{m} C_{ir}d_{kr} && (3.5)
\end{aligned}
$$

where $X$ represents the transshipment costs of vessels other than $v_i$. After the move, the total transshipment cost

$$
TC = X + C_{ik}d_{lk} + \sum_{r=1, r\neq k,l}^{m} C_{ir}d_{lr} \qquad (3.6)
$$

Then the gain of moving $v_i$ from $P_k$ to $P_l$ is

$$
\begin{aligned}
g_{il} &= \text{old cost (3.5)} - \text{new cost (3.6)} && (3.7)\\
&= C_{il}d_{kl} - C_{ik}d_{lk} + \sum_{r=1, r\neq k,l}^{m} C_{ir}d_{kr} - \sum_{r=1, r\neq k,l}^{m} C_{ir}d_{lr} && (3.8)\\
&= C_{il}(d_{kl} - d_{ll}) + C_{ik}(d_{kk} - d_{kl}) + \sum_{r=1, r\neq k,l}^{m} C_{ir}(d_{kr} - d_{lr}) && (3.9)\\
&= \sum_{r=1}^{m} C_{ir}(d_{kr} - d_{lr}) && (3.10)
\end{aligned}
$$

Note that we subtract the new cost from the old cost because we defined the gain to be the decrease in transshipment cost.

### 3.2.2 Gain Update

The above method of gain computation takes $O(degree(v_i))$, but because each $v_i$ can potentially be moved into each of the $m$ parts, we need to compute $m$ different gain values for each vessel to reflect each of the $m$ possible moves. To do so will take $O(degree(v_i) \times m)$ for each $v_i$. The total running time to compute the gains for all vessels is $O(|V|+|E|\times m)$. This is fine for the initial gain computation which is only done once. However, we have to update the node gains after each move, and recomputing the gain from scratch is too costly since the algorithm makes many moves in one pass. A faster method is to perform incremental gain update after each move.

To work out the formula for incremental gain update, we consider two cases:

1. The effect of the move on $v_i$ itself and
2. The effect of the move on the neighbouring vessels to $v_i$.

**1. Effect of move on $v_i$ itself**

Consider moving $v_i$ from $P_k$ to $P_l$ and $g_{iq}$ where $q \neq k, q \neq l$. Before the move, the gain is

$$g_{iq} = \sum_{r=1}^{m} C_{ir}(d_{kr} - d_{qr}) \tag{3.11}$$

After the move, the new gain is

$$g_{iq} = \sum_{r=1}^{m} C_{ir}(d_{lr} - d_{qr}) \tag{3.12}$$

The change in the gain of $v_i$ is therefore

$$\begin{aligned} \Delta g &= \text{new gain (3.12)} - \text{old gain (3.11)} && (3.13) \\ &= \sum_{r=1}^{m} C_{ir}(d_{lr} - d_{qr}) - \sum_{r=1}^{m} C_{ir}(d_{kr} - d_{qr}) && (3.14) \\ &= \sum_{r=1}^{m} C_{ir}(d_{lr} - d_{kr}) && (3.15) \end{aligned}$$

This update takes $O(degree(v_i))$ time. To update all $m$ gains associated with $v_i$ takes $O(degree(v_i) \times m)$ time.

**2. Effect of move on neighbouring vessels to $v_i$**

Figure 3.5 illustrates the effect that moving $v_i$ from $P_k$ to $P_l$ has on the gain $g_{jy}$ of a neighbouring vessel $v_j \in P_x$. Before moving $v_i$,

$$
\begin{aligned}
g_{jy} &= \sum_{r=1}^{m} C_{jr}(d_{xr} - d_{yr}) && (3.16)\\
&= (C_{jk} + c_{ij})(d_{xk} - d_{yk}) + C_{jl}(d_{xl} - d_{yl}) + \sum_{r=1, r \neq k,l}^{m} C_{jr}(d_{xr} - d_{yr}) && (3.17)
\end{aligned}
$$

After $v_i$ has been moved to $P_l$,

$$
\begin{aligned}
g_{jy} &= \sum_{r=1}^{m} C_{jr}(d_{xr} - d_{yr}) && (3.18)\\
&= C_{jk}(d_{xk} - d_{yk}) + (C_{jl} + c_{ij})(d_{xl} - d_{yl}) + \sum_{r=1, r \neq k,l}^{m} C_{jr}(d_{xr} - d_{yr}) && (3.19)
\end{aligned}
$$

The change in the gain of $v_j$ is therefore

$$
\begin{aligned}
\Delta g &= \text{new gain (3.19)} - \text{old gain (3.17)}\\
&= C_{jk}(d_{xk} - d_{yk}) + (C_{jl} + c_{ij})(d_{xl} - d_{yl}) - (C_{jk} + c_{ij})(d_{xk} - d_{yk}) + C_{jl}(d_{xl} - d_{yl})\\
&= c_{ij}(d_{xl} - d_{yl} + d_{yk} - d_{xk})
\end{aligned}
$$

This update can be done in $O(1)$ time and the total time taken for updating the $m$ gains of one neighbouring vessel is $O(m)$. Updating all the neighbouring vessels of $v_i$ takes $O(m \times degree(v_i))$ time.

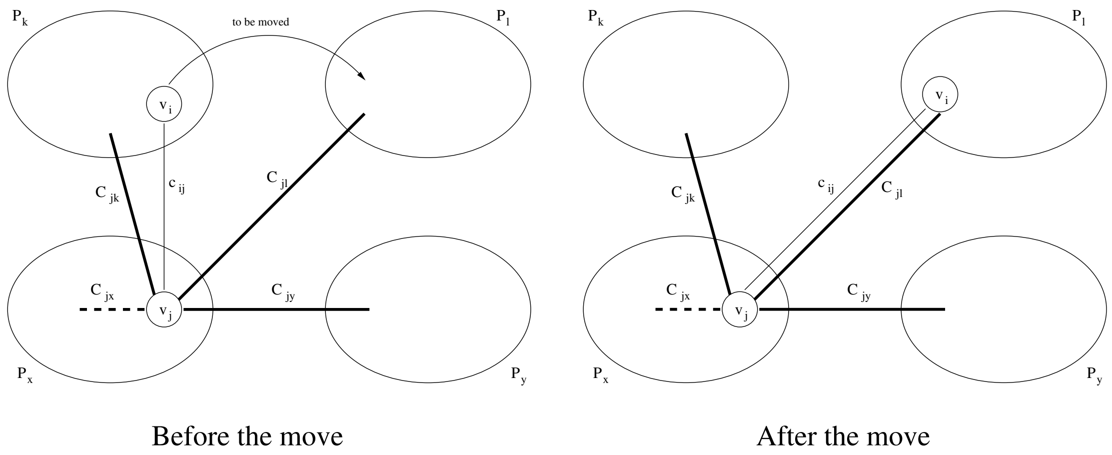


Figure 3.5: Gain Update on Neighbouring Vessel

### 3.2.3 Data Structures

As mentioned, each vessel can be moved into each of the $m$ parts and there are $n$ vessels altogether. So there are potentially $n \times m$ possible moves and selecting the move that yields the maximum gain out of these set of moves can be a time-consuming task. We want this selection to be carried out quickly as it is performed many times in one pass of the algorithm.

The FM heuristic uses a bucket data structure to support selection of the maximum gain move in $O(1)$ time and the total gain updates performed in one pass is amortised to $O(p)$ where $p$ is the total number of pins on all the nodes (nodes are known as cells in the VLSI context) to be partitioned ([FiMa82]).

Figure 3.6 shows the implementation of the FM bucket structure. At the bottom is a cell array containing pointers to cell nodes. Each cell node contains information

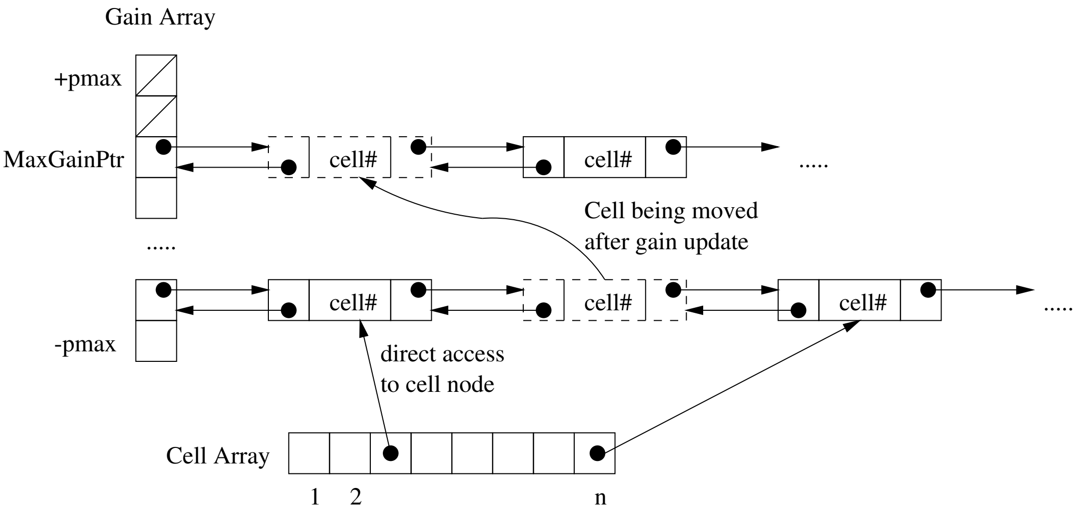


Figure 3.6: FM Bucket Structure

about the cell to be moved. The cell array is used to provide $O(1)$ access to cell nodes. On the left is a gain array containing pointers to a doubly linked list of cell nodes. All nodes in the same linked list have the same gain. The gain is indicated by the array index. In VLSI, it is possible to bound the gain to $+pmax$ and $-pmax$ since the number of pins $p$ is finite (the value of $pmax$ is dependent on the actual graph being partitioned). There is a *MaxGainPtr* which points to the array index yielding the maximum gain. It is possible for the gain array to contain unused indices, meaning there are no cells with these gain values.

Selection of the maximum gain cell to be moved is done in $O(1)$ by simply following the MaxGainPtr and selecting the first node in the linked list. After a cell has been moved, the gains of its neighbours need to be updated. The information on the cell's neighbours have been pre-computed and stored. Each neighbouring cell is then accessed via the cell array and its gain updated. After updating the gain, the cell node has to be moved to join the linked list pointed to by the appropriate array index. An example is shown in Figure 3.6 where the dashed-outline cell has its gain updated and its new gain is equal to the index value pointed to by the MaxGainPtr. This cell is removed from its original linked list and inserted in the front of the linked list pointed to by MaxGainPtr. This operation takes $O(1)$ time.

The data structure used for BAP-GP is shown in Figure 3.7. It uses similar ideas behind the FM bucket structure and is augmented with the necessary information in order for us to perform efficient gain updates. The main differences between our BAP-GP data structure and FM bucket structure are

1. The cell array is replaced by $n \times m$ look-up table because each of the $n$ vessels can be assigned to each of the $m$ sections.

2. The gain array is replaced by a doubly linked gain list. The gain list serves the same purpose as the gain array except that it reduces memory storage requirements by eliminating the unused indices. This is important because in BAP-Part, the upper and lower bounds of the gain are very large, potentially in the range of several tens of millions. Using an array results in many unused indices, wasting memory. With a linked list, we save on memory by not keeping unused index nodes while sacrificing some speed. But the empirical results from our experiments in Chapter 5 shows that the loss in

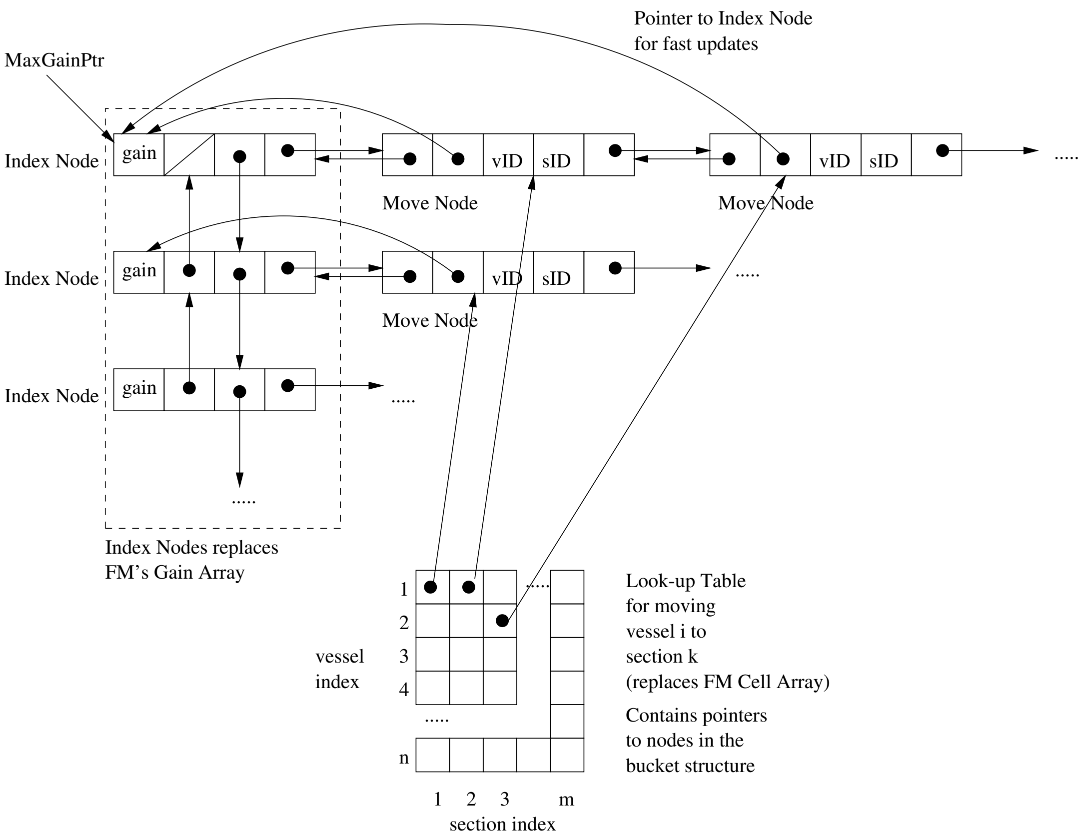


Figure 3.7: BAP-GP Bucket Structure

speed is negligible.

3. The move node (cell node in the FM bucket) is augmented with an additional pointer that points to the index node to facilitate maintenance of the data structure during gain updates.

4. The MaxGainPtr now always points to the top index node of the gain list since unused index nodes are deleted from the linked list. We can view the MaxGainPtr as the head pointer of the list.

Selection of the maximum gain move for BAP-GP is similar to that in FM, where we just follow the MaxGainPtr and select the first node in the linked list. Updating of the nodes is done via the formulae given in Section 3.2.2 and the updated move node is removed from the current linked list and added to the front of the appropriate new linked list. The index nodes in the gain list are created on demand and deleted when they point to an empty linked list of move nodes.

While our actual implementation used the doubly linked list data structure to implement the gain list, we note that the running time for the search operation may be improved by replacing the doubly linked list with a priority tree.

### 3.2.4 Maintaining Part Size

As mentioned in Section 3.1 the part size is calculated based on the density of all the vessels assigned to that part. Instead of re-computing the density of a part each time we try to assign a vessel to it, we use an incremental procedure to keep track of the free capacities of all sections. If assigning a vessel to a section causes its capacity to fall below 0, then the section is unable to accommodate that vessel.

We first find all the time zones $TZ$ as mentioned in Section 3.1.1. Since time zones are non-overlapping, we can sort them according to their starting times and assign to each of them a unique ID from 1 to $|TZ|$. Initially, each section $s$ is associated with an array $SC$ keeping track of its capacity as follows:

**Algorithms for Maintaining Part Size**

1. Initialise array of section capacities $SC[|TZ|]$.

   For $i = 1$ to $|TZ|$ do

   $SC[i] = length(s)$

To check if we can assign a vessel $v$ to $s$:

1. $Z = \{$ time zones spanned by $v$ $\}$

2. $\forall t \in Z$ do

   If $SC[t] < length(v)$, then return false.

3. Return true.

Upon the actual assignment of $v$ to $s$, we update the capacity of $s$ as follows:

1. $Z = \{$ time zones spanned by $v$ $\}$

2. $\forall t \in Z$ do

   $SC[t] = SC[t] - length(v)$

If $v$ is moved out of $s$, we simply do the reverse and add $length(v)$ to all affected $SC[t]$. The checking and updating routines take $O(|Z|)$ time each. Since each vessel spans a finite number of time zones, $|Z|$ is a small constant and the part sizes can be maintained quickly.

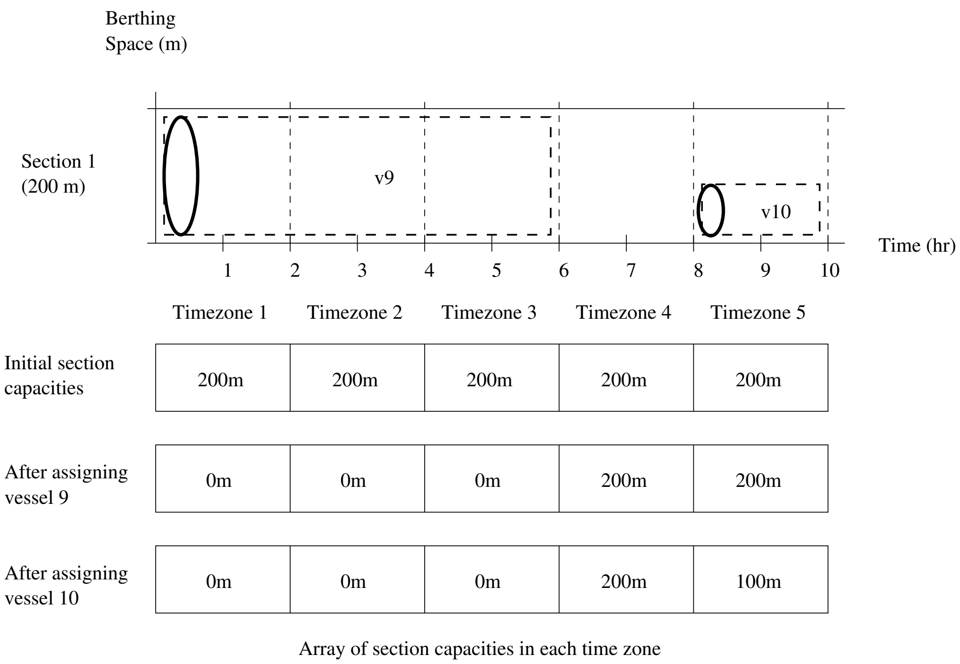


Figure 3.8: Example on Maintaining Part Size

Figure 3.8 illustrates the process. Prior to any vessel assignment, section 1 has capacity of $200m$ in all 5 time zones. After assigning $v_9$ to $s_1$, its capacities in time zones 1 to 3 drop to $0m$ while capacities in time zones 4 and 5 remain at $200m$. There is enough capacity to accommodate $v_{10}$, after which the capacity in time zone 5 drops to $100m$.

## 3.3 Extending the BAP-Part Model

Our BAP-Part model is easily extended to handle two problem extensions:

1. Modelling imports and exports

2. Penalty function for infeasible solutions

We explain both extensions in detail below.

### 3.3.1 Modelling Import and Export

In the BAP, vessels not only carry transshipment for other vessels, they may also carry containers for import into the country or export containers out of the country. These import and export operations also affect the turnaround time of the vessel — the containers have to be transported from the vessel past the gate to the storage area or vice versa.

To handle imports and exports in our BAP-Part model, we make use of a pseudo-section 0 and a pseudo-vessel 0. The gate is modelled as section 0. The distance of the gate to the section $s_k$ where the vessel is berthed is then given by $d_{0k}$ or the distance between $s_0$ and $s_k$, and $d_{0k} = d_{k0}$. The port is modelled as vessel 0. Then import is modelled as a transshipment $c_{i0}$ from $v_i$ to $v_0$ and export is modelled as a transshipment $c_{0i}$ from $v_0$ to $v_i$. The transshipment cost due to import and export for $v_i$ berthed in $s_k$ is then given by $c_{i0}d_{0k} + c_{0i}d_{0k}$.

### 3.3.2 Penalty Function

During the execution of the BAP-GP algorithm, the current solution could sometimes be infeasible. Figure 3.9 shows an example: in the left diagram, $v_2$ (150m)

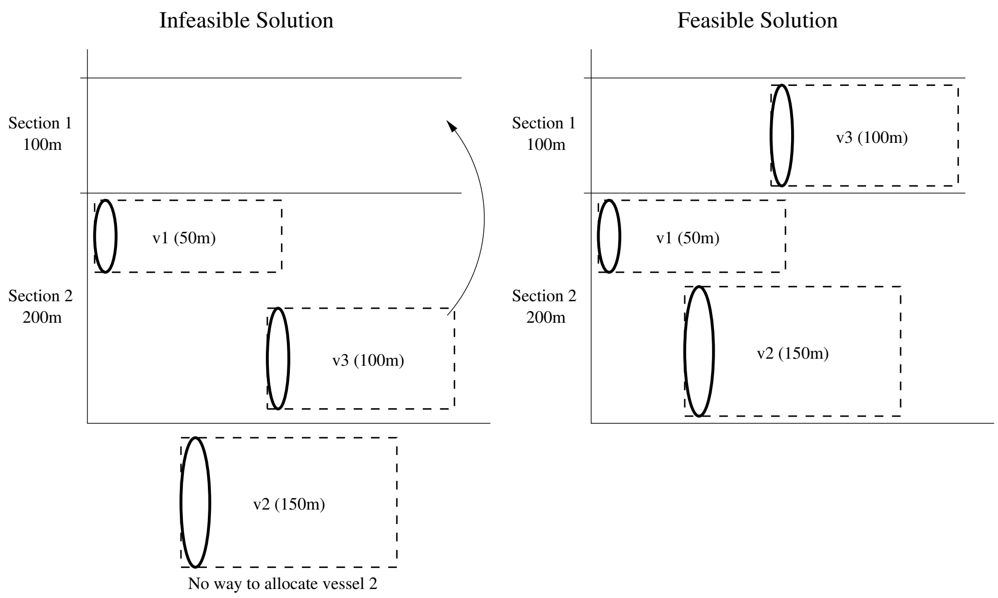


Figure 3.9: Resolving an Infeasible Solution

has to be assigned. Currently section 1 (100m) is too small to accommodate $v_2$ and section 2 (200m) has only 50m of free capacity, so it is also unable to accommodate $v_2$. However, by moving $v_3$ out of $s_2$ to $s_1$, we free up enough space in $s_2$ for $v_2$ to be assigned to it, as shown in the right diagram. We want BAP-GP to be able to find this series of moves.

To achieve this in our model, $v_2$ and all unassigned vessels are considered to be put into a pseudo-section $s_{m+1}$. We impose a penalty function by defining $s_{m+1}$ to be at a distance of $\infty$ metres from all the other sections of the port. Doing so will result in a very large transshipment cost whenever the solution contains unassigned vessels. Therefore, even though moving $v_3$ to section 1 may increase the transshipment cost a little for the example in Figure 3.9, being able to assign

$v_2$ next will more than compensate for this increase in transshipment cost.

In order for our BAP-Part objective function (3.1) to handle the above two extensions, we simply sum $i, j$ from 0 to $n$ and sum $k, l$ from 0 to $m+1$. Logically, we can view our BAP-Part objective function as a sum of two components:

$$Objective\ Value = Transshipment\ Cost + Penalty \tag{3.20}$$

As a result of making this modification to our model, if a feasible solution cannot be found, BAP-GP will attempt to find the *best infeasible solution* by minimizing the sum of the transshipment cost among the vessels and the penalty from unassigned vessels. This is because in our implementation of the algorithm, the distance of $\infty$m is implemented as a very large (albeit finite) number. So a direct comparison of the objective values is possible. Finding the best infeasible solution has practical implications. In a real world situation, the unassigned vessels will have to be delayed in berthing and we can view the penalty as an indicator of the cost the port will incur as a result of this delay[2].

## 3.4 The Complete Solution Framework

The complete solution framework for BAP-Part is shown in Figure 3.10. BAP-GP forms the core algorithm in this framework, and it is enhanced with suitable pre-processor and post-processor algorithms. This framework is proposed because:

[2]PSA actually has a holding area in the sea for temporarily holding vessels which are delayed.

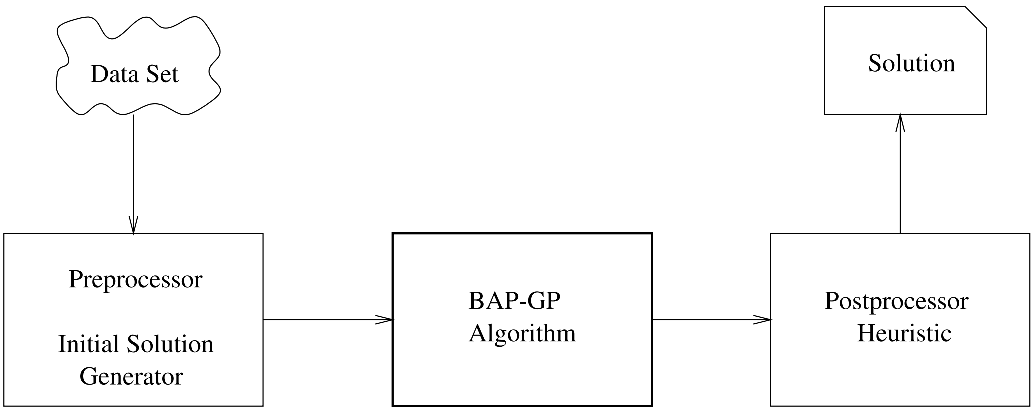


Figure 3.10: Complete BAP-Part Solution Framework

- The solution quality of the KL/FM class of heuristics is dependent on the initial solution generated and BAP-GP is no exception. A good starting solution can help BAP-GP to arrive at a good final solution while a bad one can cause the algorithm to get stuck in a bad local optimal. The pre-processor algorithm aims to generate a good starting solution for BAP-GP.

- Sometimes BAP-GP still ends up with an infeasible solution despite the implementation of the penalty function. The post-processor algorithm is there to try to resolve this infeasibility.

Both pre-processor and post-processor are heuristic-based algorithms with fast running times. We can plug-and-play different pre- and post-heuristics to try different algorithm combinations for different solutions.

### 3.4.1 Pre-Processor Initial Solution Generators

We have 7 methods to generate the initial solution, code-named *gp001, ..., gp007*. Some of them are variants of each other and hence they can be grouped under one family. The heuristics are:

- Random (gp001)

  Select a vessel at random and select a section at random to assign vessel to. If no sections can accommodate the vessel, mark it as unassigned[3] (ie. put into pseudo-section $s_{m+1}$).

- Based on vessel properties

  1. By length (gp002)

     Select vessel according to non-increasing vessel length to be assigned to a randomly selected section. This method attempts to assign the largest vessel first. The premise is that a large vessel is harder to assign later when sections are filled up.

  2. By transshipment volume (gp003)

     Select vessel according to non-increasing total vessel transshipment volume to be assigned to a randomly selected section. This method attempts to assign the busiest vessel first to reduce the overall transshipment cost.

[3] NB: All the other methods described later also handle unassignable vessels in this way.

- Simple clustering (gp004)

  Select pairs of vessels according to the transshipment between them, in non-increasing order. Try assigning both to the same section if possible, otherwise randomly assign them. This method tries to cluster vessels starting with the highest transshipment pair.

- Based on time zone density

  After implementing the above 4 methods, we discovered that certain data sets produced infeasible solutions due to the difficulty in assigning vessels in a busy time zone. So the next 3 methods attempt to resolve the busy time zones first by scanning the time zones in order of non-increasing density (or starting with the busiest time zone down to the least busy one) and assigning the vessels found in them in that order. For each set of vessels found in a time zone, the assignment of vessels is done as follows:

  1. By vessel length (gp005)

     Same idea as gp002 but start with the busiest time zone.

  2. By vessel transshipment volume (gp006)

     Same idea as gp003 but start with the busiest time zone.

  3. By number of time zones occupied (gp007)

     Select vessel in order of non-increasing number of time zones spanned to be assigned to a randomly selected section. The idea is to assign the vessel which impacts the most time zones first since such a vessel would be harder to assign later.

Not surprisingly, some methods outperform the others. The relative quality of these initial solution generators are reported in Section 5.2.

### 3.4.2 Post-Processor Heuristic

Our initial framework did not include the post-processor module. However, from the results of our experiments (see Chapter 5), we noticed that BAP-GP produces infeasible solutions for about 7 data sets. We took a deeper look into this problem and discovered that it was due to the difficulty of assigning certain vessels in busy time zones.

An example of this problem is shown in Figure 3.11, which is an actual solution returned by BAP-GP on the data set $d200$. In this example, the algorithm is unable to assign $v_{60}$ which spans 4 time zones — 69 to 72. The shaded areas represent the free space in the 4 sections across the 4 time zones. The reason $v_{60}$ cannot be assigned is that in time zone 69, none of the sections have enough free space to accommodate it, even though the total amount of free space in all the 4 sections exceed the length of $v_{60}$.

This BAP-Part *infeasibility subproblem* occurs when $U = \{$ unassigned vessels

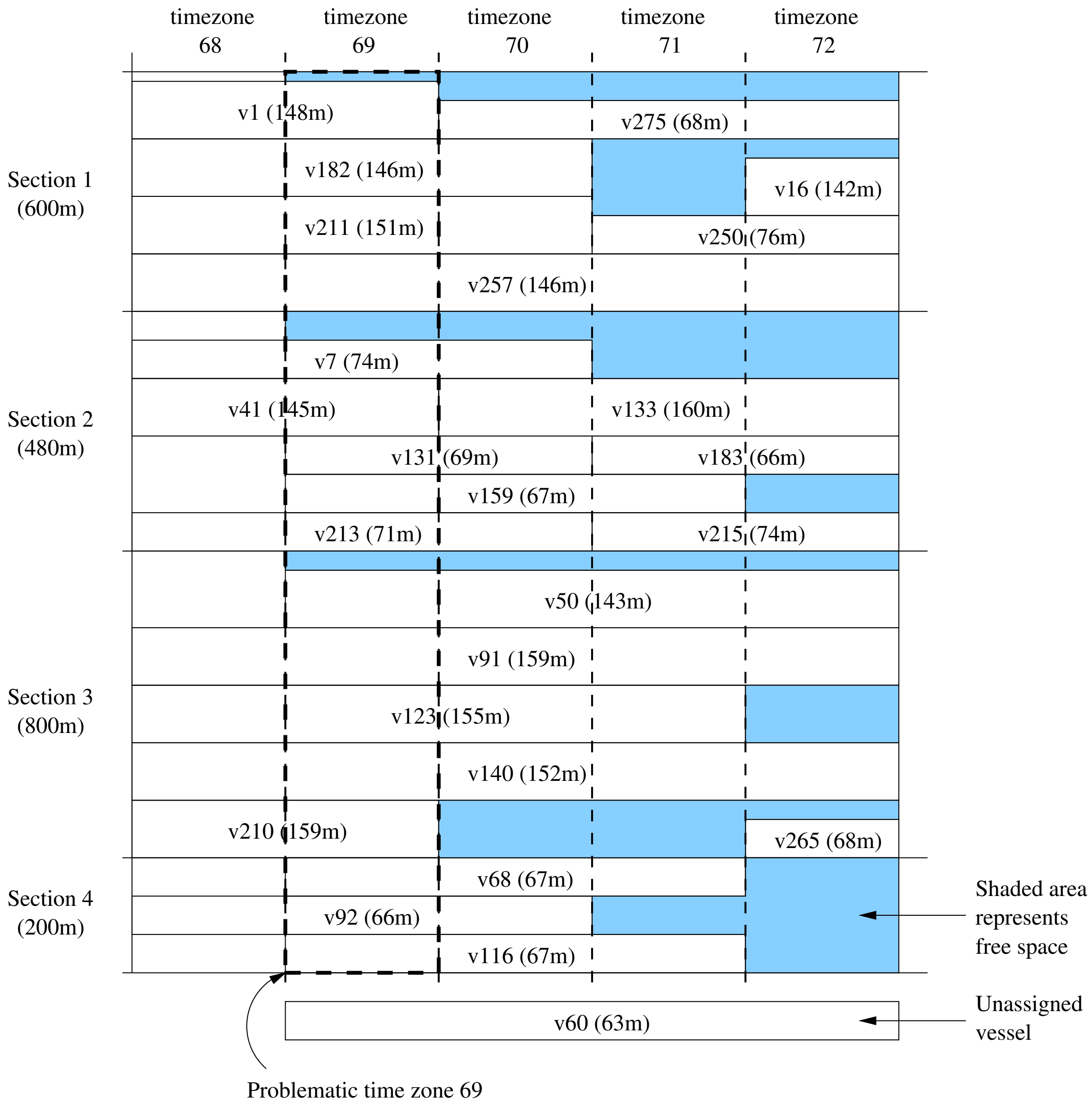


Figure 3.11: Infeasible Solution from Data Set d200

} is non-empty and

$$\forall v \in U, \qquad \sum_{k=1}^{m} free\ space(s_k) \geq length(v)$$

$$\text{but} \quad free\ space(s_k) < length(v) \qquad \forall k = 1, \ldots, m$$

This infeasibility subproblem turns out to be an instance of the 0-1 multiple knapsack problem (MKP). In the MKP, we are given a set of $n$ items, each having a certain weight and profit, and a set of $m$ knapsacks ($m \leq n$), each having a certain capacity. We want to select $m$ disjoint subsets of items so that the total profit of the selected items is maximum, and each subset can be assigned to a different knapsack whose capacity is no less that the total weight of items in that subset. The formal MKP definition can be found in the textbook on knapsack problems by Martello and Toth ([MaTo90]), which we used as part of our literature survey.

In our problem, the items are mapped to vessels and the knapsacks to sections. The item's weight is the length of the vessel and we set its profit to be 1. The knapsack capacity is the length of each section. In determining the total weight of items in a subset, we again take the density of vessels assigned to a section as the part size as mentioned in Section 3.1.

The MKP is NP-hard and [MaTo90] presented a $O(n^2)$ heuristic to solve the MKP. To resolve the infeasibility problem, we developed a post-processor heuristic based on a similar idea from [MaTo90]. The idea behind this heuristic is to locate all vessels which overlap in time zones with the unassigned vessel(s) and to re-shuffle these vessels among the sections to try to free up enough space to accommodate the unassigned vessel(s). Our heuristic is as follows:

**Algorithm To Resolve Infeasible Solutions**

$\forall v \in U$ do

1. Find $V = \{$ vessels overlapping in time zones with $v$ $\}$.
2. $\forall v_1, v_2 \in V, v_1 \in s_k, v_2 \in s_l, k \neq l$
   (a) Set *success* = false.
   (b) Unassign $v_1$ from $s_k$, $v_2$ from $s_l$.
   (c) If $s_k$ can accommodate $v_2$ and $s_l$ can accommodate $v_1$, then assign $v_2$ to $s_k$ and $v_1$ to $s_l$.
   If $s_k$ can now accommodate $v$, then assign $v$ to $s_k$ and set $success =$ true.
   (d) If $success ==$ false, then undo step 2(b) by re-assigning $v_1$ to $s_k$, $v_2$ to $s_l$, and continue with next pair of vessels.
   Else stop and continue with next unassigned vessel.

Our implementation of this algorithm runs in $O(|U| \times |V|^2)$ time, where $|U|$ is a very small constant ($< 5$ from our experiments in Chapter 5) and $|V| \ll n$ in general.