# Solving the Minimisation of the Open Stacks Problem

Joshua Davis
Monash University, Clayton,
Victoria, Australia, 3168

Supervisor: Maria Garcia de la Banda
School of Computer Science
and Software Engineering

Monash University, Clayton,
Victoria, Australia, 3168

2006

# Contents

# 1 Introduction

## 1.1 The Problem

A manufacturer makes a variety of products that are ordered by a number of customers in many combinations. However, the factory only has one machine which is capable of producing a single batch of a specific product at a time. Once the first product of a customer's order is produced, a stack is opened to which subsequent products in that order are stored. Following the completion of the last product, the stack is closed and products are shipped to the customer. Due to limited space in the factory, care must be taken to minimise the maximum number of simultaneous orders, hence to minimise the maximum number of open stacks.

Our ability to minimise the number of open stacks comes from finding an optimal schedule in which the products are to be produced. For example, if customers can choose from 10 products, it would yield 10! possible schedules. Our aim is to find the minimum number of open stacks for a given problem using the maximum of 100 products and 100 customers by employing a dynamic programming approach.

## 1.2 Previous Work

This project was largly based on out project leaders paper[1], where advanced algorithms implemented were discussed.

# 2 Methodology

## 2.1 Notation and Modelling

We will use the example depicted in table 1 to introduce the notation used throughout the report and to formalise our model. Let $P$ be a set of products where $p \in P$ and $C$ be a set of customers where $c \in C$. Let $c(p)$ be a function which returns a set of clients (column $p$ from the table), ordering product $p$, we can extend this to where $P' \subseteq P$ we can now define $c(P')$ as

$c(P') = \bigcup_{p \in P'} c(p)$

We will need to evaluate a given schedule to implement simple search algorithms and to check that the solutions found by advanced ( $A\star$, stepwise, etc) search algorithms are correct. To help us do this will define an array, $schedule[i]$, which maps each point in time, $1 \leq i \leq |P|$, to an unique product $p$, where $p \in P$. Then it is possible to iterate over a schedule, however we need to evaluate how many stacks are active at time $i$ to determine the minimun number of open stacks. Let $active(p, A)$ determine the number of active customers at each position $i$ (i.e. $p = schedule[i]$) while retaining the maximum each possible time $i$. Let $A \subset P$ denote the products scheduled after $p = schedule[i]$, the set of active customers can now be defined as

$active(p, A) = c(p) \cup \{c(P - A - \{p\}) \cap c(A)\}$, where $p = schedule[i]$

| Schedule | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Product | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
| $C_1$ | 1 | * | * | * | * | 1 |
| $C_2$ | 1 | 1 | * | * | * | 0 |
| $C_3$ | 0 | 1 | * | * | * | 1 |
| $C_4$ | 0 | 0 | 1 | 1 | 1 | 0 |
| $C_5$ | 0 | 0 | 1 | * | 1 | 0 |

Table 1: The stars indicate already open stacks. At $schedule[3]$, $P_3$ alone has two active customers $C_4$ and $C_5$ as well as $C_1$, $C_2$ and $C_4$ active yielding five open stacks. The current search space is $6!$ (720 possible schedule)

Now the problem may be formalised as,

$\min_{eachschedule} \max_{1 \leq i \leq |P|} \{active(p, A) | p = schedule[i]\}$

## 2.2 Preprocessing

Several methods can be used to simplify the problem and thus reduce the amount of search space that needs exploring to find the optimal solution. Table 1. will be used to describe two methods, subsumption and partitioning which were implemented in the project. When these methods are successful it has been proven to dramatically reduce the time taken to find an optimal solution.

Subsumption eliminates scheduling products which have no impact on the optimal solution, iff $c(p') \subseteq c(p'')$. For example, if $p'$ is a pen and $p''$ is a pen lid, customers ordering only $p'$, will also require $p''$. Therefore $p''$ can be removed from the schedule and later reinserted directly after $p'$ without affecting the optimal solution. Below is subsumed products removed from table 1, resulting in table 2.

| Schedule | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Product | $P_1$ | $P_2$ | $P_3$ | $P_6$ |
| $C_1$ | 1 | * | * | 1 |
| $C_2$ | 1 | 1 | * | 0 |
| $C_3$ | 0 | 1 | * | 1 |
| $C_4$ | 0 | 0 | 1 | 0 |
| $C_5$ | 0 | 0 | 1 | 0 |

Table 2: $P_4$ and $P_5$ have been revmoved, leaving a search space of $4!$ (24 possible schedules)

Partitioning the problem into sub problems, is possible if $P$ can be partitioned into two sets $P'$ and $P''$, such that $c(P') \cap c(P'') = \emptyset$. Hence there are no customers requiring a product from $P'$ and $P''$. Then $P'$ and $P''$ are independent problems and can be solved separately, where the he solutions can be concatenated together. This dramatically reduces the search space from $|P|!$ to $|P'|! + |P''|!$, for example if $|P| = 10$, and $|P'| = 5$, the search space is reduced from millions to hundreds of possible schedules. Below table 3 has been partitioned.

| Schedule | 1 | 2 | 3 | 1 |
|---|---|---|---|---|
| Product | $P_1$ | $P_2$ | $P_6$ | $P_3$ |
| $C_1$ | 1 | * | 1 | 0 |
| $C_2$ | 1 | 1 | 0 | 0 |
| $C_3$ | 0 | 1 | 1 | 0 |
| $C_4$ | 0 | 0 | 0 | 1 |
| $C_5$ | 0 | 0 | 0 | 1 |

Table 3: The double vertical lines indicates the partition in the problem, where the first partition has a search space of $3!$ (6 schedules) and the second has $1!$ totaling 6 possible schedules from 24 in table 2.

## 2.3 Bounds

The optimal solutions must exist within a domain, which can be trivially stated as the $\max_{p \in P} c(p)$ for the lower bound and $|C|$ for the upper bound. However the bound may be improved by various techniques such as examing the problem as a graph where the nodes are customers and edges products to find the maximum weighted clique, and taking that as the new upper bound. As the be tighten, part of the solution space may be avoided allowing the search to complete faster.

## 2.4 Search Algorithms

After preprocessing, the reduced problem is passed to a search algorithm where the optimal solution may be found. We will explore exhaustive, branch and bound, $A\star$ tree search algorithms along with satisfiability variants stepwise, binarychop and backwards which expand on $A\star$. Ultimately, we will be able to apply knowledge about the stucture of the probem to reduce the amount of time searching.

### 2.4.1 Exhaustive

This algorithm exhaustively explores the entire search space, which is slow as every possible solution must be found and evaluated. This is not possible even with moderately sized $|P|$, as the computation required to find a solution explodes beyond the capabilies of current computers. However. it does have benefits over small values of $|P|$. Firstly, it reinforces the correctness of advanced algorithms as optimal solutions may be compared. Secondly, it provides a good metric to determine the efficiency of primitive methods by comparing differing implementations.

**Algorithm 1** Exhaustive Search

**procedure** $exhaustive(S, Sched, U)$
  **if** $S = \emptyset$ **then**
    **return** $maxOpen(Sched)$
  **end if**
  $min := U$
  $T := S$
  **while** $T \neq \emptyset$ **do**
    $p :=$ any $p \in T$
    $T := T - \{p\}$
    $sp := exhaustive(s - \{p\}, Sched :: p, U)$
    **if** $sp < min$ **then**
      $min := sp$
    **end if**
  **end while**
  **return** $min$
**end procedure**

The algorithm can be thought of doing this; it begins by chopping the search space into smaller sub regions which the algorithm repeats recursively to form a tree. The algorithm is not concerned with narrowing the search space, therefore useful information is ignored at branches while the tree is grown, which will be utilized by later algorithms (branch and bound, etc) to speed up the search. The while loop selects a $p \in T$, which grows the tree horizontally, whereas the recursive function call, with $p$ removed from $S$ expands the tree vertically. When $S = \emptyset$, we are at a leaf of the tree with one possible $|P|!$ schedules. We evaluate the schedule, and the number of open stacks is passed up the tree being compared to other similarly obtained solutions at each leaves. This is done after the recursive call with the minimum (optimal soultion) retained. Below is an example of a generated tree.

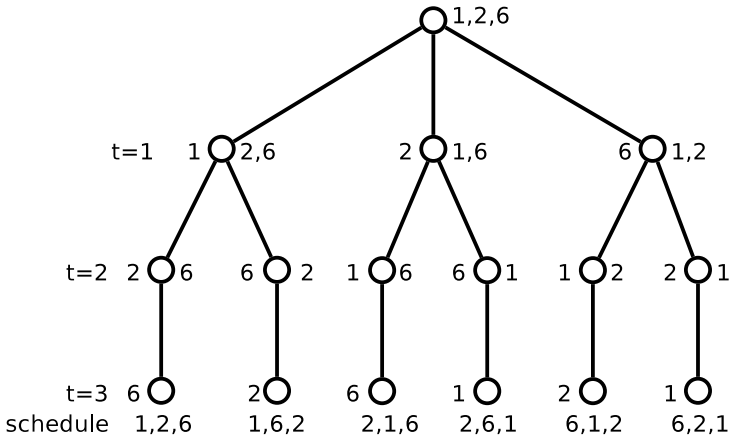


Figure 1: Exhaustive search tree of data in table 3

### 2.4.2 Branch and Bound

As $|P|$ increases the search space exponentially increases, so a system must be put in place to reduce the size of raw search performed. We will build on the idea of traversing a tree which was introduced in the above algorithm (branching). Unlike the exhaustive search algorithm, useful information contained at the branch is not simply ignored but is used to narraow the search space. This algorithm considers the information and decides whether the optimal solution resides further down that branch by comparing it to a bound which is continually updated as the tree is traversed (bounding). This avoids exploring superfluous search space which cannot improve on the current optimal solution (pruning).

**Algorithm 2** Branch and Bound Search

**procedure** $bb(S, Sched, L, U)$
  **if** $S = \emptyset$ **then**
    **return** $0$
  **end if**
  $min := U$
  $T := S$
  **while** $min > L$ **and** $T \neq \emptyset$ **do**
    $p := index \min\{active(p, S - \{p\}) | p \in T\}$
    $T := T - \{p\}$
    **if** $active(p, S - \{p\}) \geq min$ **then**
      **break**
    **end if**
    $sp := \max(active(p, S - \{p\}), bb(S - \{p\}, Sched :: p, L, min))$
    **if** $sp < min$ **then**
      $min := sp$
    **end if**
  **end while**
  **return** $min$
**end procedure**

The algorithm starts by checking if $S = \emptyset$, if so, no stacks are required and 0 is returned. Otherwise, it enters the while loop (branches) where a simple heuristic is used to determine the order of visiting each branch by picking $p \in T$ with the least active customers first. Hopefully this heuristic leads us to explore the optimal branch and subsequent iterations are avoided as the remaining p may have a greater or equal number of active customers compared to the current $min$ (bounding). If this is true the while loop terminates as no better solution may be obtained at this branch (pruning). The maximum of the current active value compared to the value passing back up the tree is returned the optimal solution.

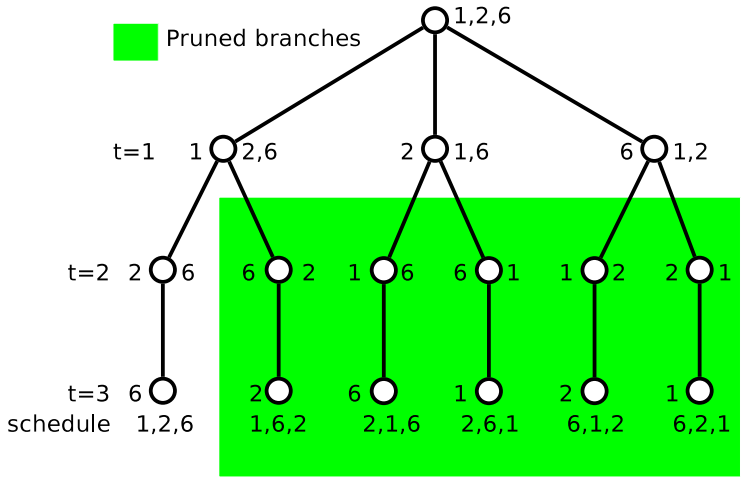


Figure 2: Branch and bound tree of data in table 3

### 2.4.3 $A\star$

Our current tree may have up to $|P|!$ branches explored in an attempt to find the optimal solution. After careful examination of the search tree it becomes evident that many branches throughout the tree contains pool of products where the next product to be selected is not unique (shown in figure 3). The active function used in the previous algorithm does not depend on the order of the products scheduled before and after the current branch. This implies the maximun number of open stacks at each of these repeated branches is the same. Therefore, the future does not depend on the past and now our problem is amendable to dynamic

programming. After evaluating a given branch the solution is stored and can be retrieved at branches with the same pool of products to be scheduled. This moves the search space down to $2^{|P|}$ as each of the unique branches make up the super set of all elements in $P$.

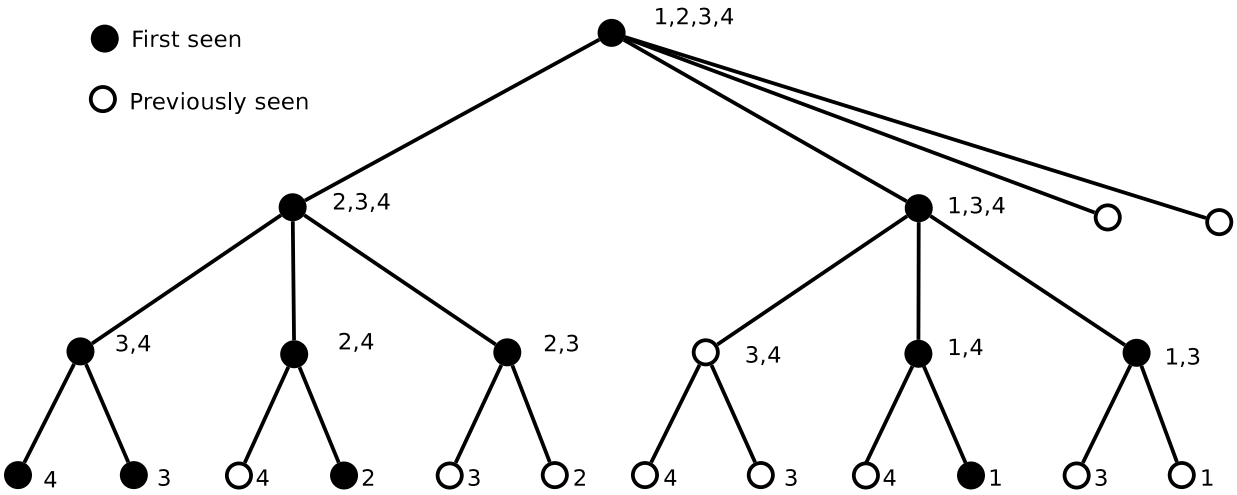


Figure 3: An example of a truncated tree with 4 products, only half the tree shon

The following is the algorithm.

**Algorithm 3** $A\star$ Search

**procedure** $stacks(S, stacks, L, U)$
  **if** $S = \emptyset$ **then**
    **return** $0$
  **end if**
  **if** $stacks[S]$ **then**
    return $stacks[S]$
  **end if**
  $min := U$
  $T := S$
  **while** $min > L$ **and** $T \neq \emptyset$ **do**
    $p := index \min\{active(p, S - \{p\}) \mid p \in T\}$
    $T := T - \{p\}$
    **if** $active(p, S - \{p\}) \geq min$ **then**
      **break**
    **end if**
    $sp := \max(active(p, S - \{p\}), bb(S - \{p\}, Sched :: p, L, min))$
    **if** $sp < min$ **then**
      $min := sp$
    **end if**
  **end while**
  $stacks[S] := min$
  **if** $min > U$ **then**
    $FAIL := FAIL \cup \{S\}$
  **end if**
  **return** $min$
**end procedure**

This algorithm is identical to branch and bound except solutions for each computed set are stored in a lookup table after the while, which can be later reused avoiding unnesscessary recomputing by skipping the while loop.

### 2.4.4 Stepwise

The following three algorithms exploit that fact that $A\star$ is very effective when the lower bound equals the upper bound. Stepwise, backwards and binarychop all turn our problem into a series of satisfiability problems by letting $U = L = n$ when calling $stacks$. If $n$ is returned, $n$ is the minimum number of stacks, otherwise stacks needs to be called again in hope to find the optimal solution. The following three algorithms step through the domain $[L, U]$ in different ways attempting to find the optimal solution. However, the lookup table after a non-successful call to $stacks$, the loopup table may contain viable solutions which can be reused in if stepping forwards. As we need to be able record what solutions are incorrect, $A\star$ is slightly modified, after the while loop sets with values $> U$ are recorded, which are simply removed form the lookup table before the next call to stacks. The following is added after the while loop,

```
if min > U then
  FAIL := FAIL ∪ {S}
end if
```

which is already included in algorithm 3 above.

**Algorithm 4** Stepwise

```
procedure stepwise(P, L, U)
  for try := L to U do
    FAIL := ∅
    min := stacks(P, try, try)
    if min = try then
      return min
    end if
    for S ∈ FAIL do
      stack[S] := 0
    end for
  end for
  return U + 1
end procedure
```

Stepwise, tries each value within the domain $L \leq try \leq U$ starting at the lower bound calling $stacks(P, try, try)$ with $try$. Where $try$ is incremented and failed sets removed from the table with each failure, until an optimal solution is found.

### 2.4.5 Backwards

**Algorithm 5** Backwards

```
procedure backwards(P, L, U)
  gmin := U + 1
  try := U
  while try ≥ L do
    min := stacks(P, try, try)
    if min > try then
      return gmin
    end if
    gmin := min
    try := min − 1
    stack[S] := ∅
  end while
end procedure
```

Backwards moves from $U$ towards $L$, and has the ability to jump a number of values. A valid solution is found if we step one solution back too far and obtain an invalid solution. Unlike stepwise no solutions in the lookup table may be re-used.

### 2.4.6 Binarychop

**Algorithm 6** Binarychop

**procedure** $binarychop(P, L, U)$
  $gmin := U + 1$
  **while** $L \leq U$ **do**
    $FAIL := \emptyset$
    $try := (U + L)$ div 2
    $min := stacks(P, try, try)$
    **if** $min \leq try$ **then**
      $gmin := min$
      $U := min - 1$
      $stack[S] := \emptyset$
    **else**
      $L := try + 1$
      **for** $S \in FAIL$ **do**
        $stack[S] := 0$
      **end for**
    **end if**
  **end while**
  **return** $gmin$
**end procedure**

Binarychop is a variant of the two algorithms above, using both benefits of retaining solutions in the lookup table and the possibility of jumping a number of values. It is especially effective when the solution lays close to middle of the domain, as a binary search technique is used.

Unlike exhaustive and branch and bound at the completion of these algorithms we do not have a schedule. However, there is enough information in the lookup table to reproduce the schedule in polynomial time.

## 2.5 Non Opening Products

This is a technique used to reduce the search space in the previously explained algorithms (not including exhaustive), which returns a selected product before entering the while loop. It has been proven that products which do not open a new stack, if explored first does not affect the optimality of the solution. The proof is complicated and out of the scope of this project. This functionality can be added by the following lines before the while loop.

**if** $\exists p \in S.c(p) \supseteq \{c(P - S) \cap c(S)\}$ **then**
  $alg(S - p, L, U)$
**end if**

# 3 Implementation

Here is a simple chart showing the program execution.

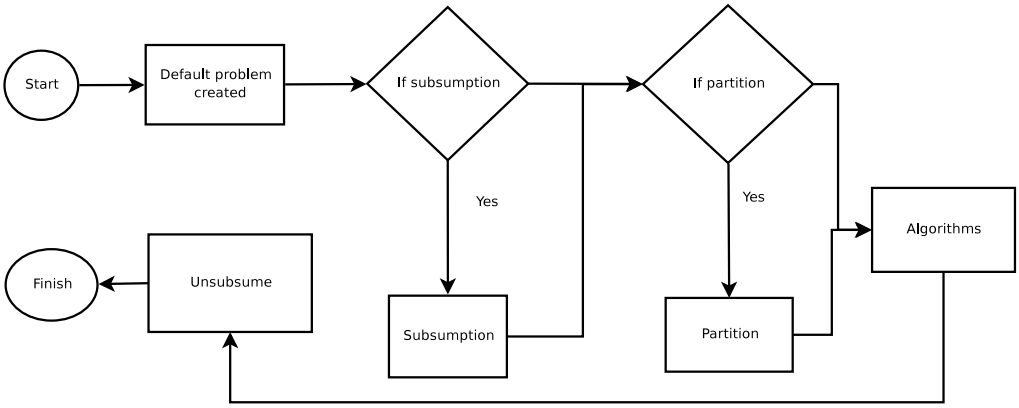


Figure 4: Program execution path

## 3.1 Data types

Implementing the above algorithms required many data types which may be done in many different ways, with varying strengths and weaknesses. Therefore, it was important that the most compatible implementation was matched with a suitable task to ensure maximum efficiency. If no thought was given to how the data type was used, the efficiency and perhaps effectiveness of obtaining a solution may have been lacking.

Initial analysis of the processes involved to obtain a solution showed that any operations performed during a search algorithm (staggering $O(|P|!)$) needs to be fast, but in operations performed in preprocessing (perhaps $O(n^3)$) is not as critical. This crucial observation lead to chosing implementations which favoured the operations preformed in the search algorithms. This lead to the following data types to be implemented in this project.

### 3.1.1 Set Data Type

The universal set is finite with up to 100 products and 100 clients, so using computer memory as bit string where the number of bits used is equal to the number of elements in the universal set (100 elements). A given set is then represented by a bit string in which corresponding elements in that set are 1 and all other bits are 0.

Set operations using a bitwise implementation are fast and simple in comparison to list operations, when the sets are of a small size as in this project. A desirable side effect of using bit strings is that bitwise operations $\& \mid \wedge$ can be used to implement the majority of operations. Furthermore, bitwise operations are fast, in most cases in constant time and complex procedures which may have been seen if using a list set representation are abolished. This may reduce bugs and lead to a more reliable code. Below is an example of an 8 bit segment of memory, where x can be 1 representing an element in the set or 0, the element is not in the set.

x x x x x x x x Memory
7 6 5 4 3 2 1 0 Element

Here are two simple sets, in which the | operation computes the union.

1 0 0 1 0 0 0 1 $= \{0, 4, 7\} = A$
0 1 0 1 0 0 1 0 $= \{1, 4, 6\} = B$

$A \mid B \equiv A \cup B$
1 1 0 1 0 0 1 1 = 0,2,4,6,7

### 3.1.2 List Data Type

Analysis of the search algorithms showed that the following list operations had to be fast, evident in while loop and functions calls from the body of exhaustive, and within the branch and bound algorithms. But the more advanced algorithms ($A\star$ and backwards) lacked list operations all together, so less time was spent optimising the implementation. Choosing the list implementation proved difficult, below is a table comparing performance of differing implementations.

| Operations | Array (end) | Array (Body) | Link List (end) | Link List (body) |
|---|---|---|---|---|
| Indexing | $O(1)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Inserting | $O(1)$ | $O(n)$ | $O(1)$ | $O(n)$ |
| Deleting | $O(1)$ | $O(n)$ | $O(1)$ | $O(n)$ |

Table 4: Array vs. linked list implementations

Given that obtaining values via an index was more frequent than deleting and the majority of the operations were at the end of a list, the array implementation seemed like the best choice. Only limited analysis lead to this decision and analysis of the frequencies and localality of data lookups vs. list deletes was not explored. However, our chosen implementation only suffers slight performance loss using $maxOpen$ which is used in the exhaustive and branch and bound algorithms. All the other algorithms used $O(1)$ operations.

### 3.1.3 Lookup Table Data Type

The obvious choice was to use a hash table with chaining. Not only are hash tables fast with all operations $O(1)$, given that collisions are avoided (or minimised). The hash table was used to mimic a linked list in recording failed sets in $A\star$. This is possible if the hash table is initialised with 1 bucket.

## 3.2 Data Structures

In order to successfully implement preprocessing and search algorithms all information required must be together in a structure. To further complicate matters, partitioning breaks the problem up into further sub problems, which alone has independent information. So the following structures were used.

```
typedef struct {
  char *n;        /* instance name */
  int cn;         /* customer no. */
  int pn;         /* product no. */
  int lb;         /* lower bound */
  int ub;         /* upper bound */
  int maxIterations;
  int *provenOptimal;
  int t;

  List *sp;       /* subsumed product list */
  Set **cust;     /* customer array N.B c(p) for fast look up */

  Segment *s;     /* contains all all MOSP's partitions and info */
  int sn;         /* contains the number of partitions */

  int flags;
```

```
  Stats stats;

  int **m;        /* matrix */
} Mosp;
```

In which all information is recorded. And,

```
/* contains everything an algorithm needs to complete */
typedef struct {
  int u;          /* contains the [u]pper bound */
  int l;          /* contains the [l]ower bound */
  int pn;         /* [p]roduct [n]umber */
  int cn;         /* [c]ustomer [n]umber */
  int *maxIterations;
  int *provenOptimal;
  int *t;

  List *p;        /* contains the [p]roduction */
  Set *up;  /* contains all elements in production */
  Set *uc;
  Set **cust;     /* points to m->cust */
  Stats *stats;

  int flags;      /* launch time flags */
} Segment;
```

which contains information only relevant to sub-problems.

The following data structure was used to record all statistics.

## 3.3 Preprocessing

Reference to the code may help to to understand the following.

Determining what customers ordered a given product was frequent and expensive with an $O(|C|)$ method, so a lookup table was implemented. An array, $cust[i]$ returning a set of clients in constant time for product $i$.

Subsumption was implemented using two loops iterating over the product list. The outer loop initially starts with the first product in the list, $cur$ where the inner loop always starts with the product at the front of current the list, $cmp$. The inner loop cycles to the last product before giving control back to the outer loop, where $cur$ is moved the the next product in the list. This lets every product get compared with every other product. If the $cust[cur] \subseteq cust[cmp]$, cmp is deleted from the list and the loop continues until all subsumed products have been removed from the list.

The implementation of partitioning the problem into sub-problems is similar in that two loops are used to compare every product together. However, the outer loop is not a product, but a set of clients $cmp$, which is initialised to $cust[prodA]$ before the inner loop begins, also a new segment is made. If in the inner loop $cmp$ is found to intersect with $cust[prodB]$, then $cmp = cmp \cup cust[prodB]$ and $prodB$ is then deleted from that list and added to the new segment and the inner loop starts again at the beginning of the list. The inner loop continues until no more products intersect, then $cmp$ is set to the next product in the list and a new segment is made.

## 3.4 Search Algorithms

Reference to the code may help to to understand the following.

The exhaustive search algorithm described in methodology was efficient from an 1 to 1 implementation stand point. The only problems were extracting an optimal schedule and determining where variables should exist in the computer memory (stack vs heap). Where all sets have been allocated to the stack and lists to the heap (purely due to the way list were implemented, they need to go on the heap) . Using a structure to contain the optimal schedule and the minimum value appeared to be the most elegant solution, but was not used in favour of retaining the best schedule in a global variable (memory leak when using a structure to retain both prevented this method). At the base case the current $Sched$ is placed in the global variable if it is better then the current best schedule in the global variable as well as the result from $maxOpen(sched)$ being returned and passed back up the tree with the best retained.

The implementation of branch and bound was not clear like above, as the algorithm was not efficient with a lot of recomputing of active value within the while loop taking place. The sensible thing to do was to remove the active function call and to generate the $indexMin$ data structure before the while loop. As $indexMin$ contains the active value for each product, the multiple calls to active inside the while was no longer necessary. Retaining the optimal solution uses the same technique as in exhaustive search, but retaining the minimum was done in the while loop the same way as the algorithm in methodology.

Implementing $A\star$ was particularly easy as all the code was the same as branch and bound expect for the hash function calls which needed adding before and after the while loops. Due to lack of time, I did not experient with special hash functions or hash tables sized with prime numbers.

Implementing stepwise, backwards and binary chop was also straight forward, with a direct translation from the algorithms started in methodology. Recording off successful sets was removed as it was redundant.

# 4 Guide to Using Software

The mosp utility runs both preprocessing and the search algorithm on a given problem instance and reports, statistics and the solution if obtained.

It has many command line arguments, based on a command line interface:

mosp flags problemFile

If no flags are specified or only -h is used, mosp will print flags which turn on different search algorithms, among other features.

```
-p find independent sub-problems
-d find redundant products, and consume them
-e solve using recursize exhaustive algorithm
-i <integer> stop after a number of iterations
-l <integer> use this number as the lower bound
-u <integer> use this number as the upper bound
-y solve using binarychop algorithm
-w solve using the backwards algorithm
-t show stats
-s solve using the stepwise algorithm
-f solve using initerative exhaustive algorithm
-b solving using branch and bound
-c use opening products first
```

The following example represents using subsumption and partitioning to solve a problem instance using the stepwise algorithm

mosp -dps problem

# 5 Testing

The testing was performed according to the following schemes. Depending on flags used during compilation, debugging, assertion and verbose output may be enabled. I am fairly confident that all the data type functions are correct.

All data type methods were given strict behaviour on all inputs, that is boundary cases were identified and dealt with elegantly (some instances the program terminates with an assertion). Unfortunately, in some complex functions the program terminates with an assertion due to speed considerations and shows the line number and source file where the program failed. This is not enough information in many cases to identify what caused the failure so extra information is included in the source code before the assertion describing why the assertion should be true.

All data type methods were extensively checked, with each boundary cases tested. The set library went under even more extensive tests, using randomly generated sets and testing relational, algebraic and special cases as this is the most used DT.

Due to lack of time, many higher order function lacked specific testing functions, but as these methods are built upon the data type methods, we can reason that is the logic is sound the behaviour of the function must be correct.

# 6 Results

The above implementation was tested with:

- Processor: Pentium IV 3.0Ghz
- Memory: 2.5GB
- Linux Kernel 2.6.9-34.ELsmp SMP
- Compiler: gcc 3.4.4
- Location: Monash University, student remote access terminal

The following tables are results obtained by running mosp with different search algorithms. Tables displaying full titles in each column were too big. Therefor shortened the following shortened titles stand for:

Instance, Instance Name. Optimal, Proven optimal. Parts, Partitions where each number is a partition and the number of products in it. Subs, Subsumptions. Open, Non-Opening Products counter. Fail, Fail set counter. Steps, Number of steps taken.

If all the values in a row are 0, the program could not complete due to exceeding 12mb. Only evident in $A\star$.

Also the filenames were slightly changed to alow the tables to fit, GPN is GP_NWRS. $War_1$ is Warwick_1023. $War_2$ is Warwick_1718. $War_3$ is Warwick_1851. $Ware_1$ is Warwick_0003. $Ware_2$ is Warwick_0008.

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 0 | 50 | 50 | 0 | 42 | 8 | 1000001 | 0 | 0 | 0 | 293.143 |
| $GP_2$ | 0 | 50 | 50 | 0 | 48 | 2 | 1000001 | 0 | 0 | 0 | 296.136 |
| $GP_3$ | 0 | 50 | 50 | 0 | 50 | 0 | 1000001 | 0 | 0 | 0 | 298.412 |
| $GP_4$ | 0 | 50 | 50 | 0 | 37 | 13 | 1000001 | 0 | 0 | 0 | 287.889 |
| $GP_5$ | 0 | 100 | 100 | 0 | 100 | 0 | 1000001 | 0 | 0 | 0 | 297.535 |
| $GP_6$ | 0 | 100 | 100 | 0 | 100 | 0 | 1000001 | 0 | 0 | 0 | 296.205 |
| $GP_7$ | 0 | 100 | 100 | 0 | 99 | 1 | 1000001 | 0 | 0 | 0 | 310.19 |
| $GP_8$ | 0 | 100 | 100 | 0 | 96 | 4 | 1000001 | 0 | 0 | 0 | 298.352 |
| $GPN_5$ | 0 | 20 | 30 | 0 | 20 | 10 | 1000001 | 0 | 0 | 0 | 4122.9 |
| $GPN_6$ | 0 | 20 | 30 | 0 | 23 | 7 | 1000001 | 0 | 0 | 0 | 5024.497 |
| $GPN_7$ | 0 | 25 | 60 | 0 | 32 | 28 | 1000001 | 0 | 0 | 0 | 296.558 |
| $GPN_8$ | 0 | 25 | 60 | 0 | 40 | 20 | 1000001 | 0 | 0 | 0 | 283.604 |
| $Shaw_5$ | 0 | 20 | 20 | 0 | 18 | 2 | 1000001 | 0 | 0 | 0 | 3557.243 |
| $Shaw_6$ | 0 | 20 | 20 | 0 | 19 | 1 | 1000001 | 0 | 0 | 0 | 3795.025 |
| $Shaw_7$ | 0 | 20 | 20 | 0 | 18 | 2 | 1000001 | 0 | 0 | 0 | 3512.881 |
| $Shaw_8$ | 0 | 20 | 20 | 0 | 19 | 1 | 1000001 | 0 | 0 | 0 | 3794.88 |
| $Shaw_9$ | 0 | 20 | 20 | 0 | 18 | 2 | 1000001 | 0 | 0 | 0 | 3494.539 |
| $SP_1$ | 0 | 25 | 25 | 0 | 16 | 9 | 1000001 | 0 | 0 | 0 | 2992.044 |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000001 | 0 | 0 | 0 | 309.396 |
| $War_1$ | 0 | 20 | 20 | 0 | 16 1 1 1 | 1 | 1000001 | 0 | 0 | 0 | 2971.776 |
| $War_2$ | 0 | 30 | 30 | 0 | 23 3 1 1 1 | 1 | 1000001 | 0 | 0 | 0 | 5028.988 |
| $War_3$ | 0 | 30 | 30 | 0 | 12 1 3 1 | 13 | 1000001 | 0 | 0 | 0 | 2008.417 |
| $War_e$ | 0 | 10 | 10 | 0 | 10 | 0 | 1000001 | 0 | 0 | 0 | 1610.626 |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 13714 | 0 | 0 | 0 | 17.348 |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 1971 | 0 | 0 | 0 | 4.333 |

Table 5: Exhaustive with partioning and subsumption, total time taken 45s.231

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 45 | 50 | 50 | 1 | 42 | 8 | 408 | 0 | 0 | 0 | **21.727** |
| $GP_2$ | 40 | 50 | 50 | 1 | 48 | 2 | 24867 | 0 | 0 | 0 | 1312.662 |
| $GP_3$ | 40 | 50 | 50 | 1 | 50 | 0 | 5371 | 0 | 0 | 0 | 347.42 |
| $GP_4$ | 30 | 50 | 50 | 1 | 37 | 13 | 94 | 0 | 0 | 0 | 5.975 |
| $GP_5$ | 95 | 100 | 100 | 1 | 100 | 0 | 35196 | 0 | 0 | 0 | 8742.903 |
| $GP_6$ | 0 | 100 | 100 | 0 | 100 | 0 | 1000001 | 0 | 0 | 0 | 115323.215 |
| $GP_7$ | 0 | 100 | 100 | 0 | 99 | 1 | 1000001 | 0 | 0 | 0 | 185875.257 |
| $GP_8$ | 0 | 100 | 100 | 0 | 96 | 4 | 1000001 | 0 | 0 | 0 | 93389.832 |
| $GPN_5$ | 12 | 20 | 30 | 1 | 20 | 10 | 24799 | 0 | 0 | 0 | 198.407 |
| $GPN_6$ | 12 | 20 | 30 | 1 | 23 | 7 | 253122 | 0 | 0 | 0 | 1817.737 |
| $GPN_7$ | 0 | 25 | 60 | 0 | 32 | 28 | 1000001 | 0 | 0 | 0 | 13208.477 |
| $GPN_8$ | 0 | 25 | 60 | 0 | 40 | 20 | 1000001 | 0 | 0 | 0 | 31298.675 |
| $Shaw_5$ | 14 | 20 | 20 | 1 | 18 | 2 | 471516 | 0 | 0 | 0 | 3655.681 |
| $Shaw_6$ | 14 | 20 | 20 | 1 | 19 | 1 | 31620 | 0 | 0 | 0 | 356.013 |
| $Shaw_7$ | 13 | 20 | 20 | 1 | 18 | 2 | 29519 | 0 | 0 | 0 | 260.941 |
| $Shaw_8$ | 14 | 20 | 20 | 1 | 19 | 1 | 64613 | 0 | 0 | 0 | 716.566 |
| $Shaw_9$ | 13 | 20 | 20 | 1 | 18 | 2 | 13498 | 0 | 0 | 0 | 138.071 |
| $SP_1$ | 9 | 25 | 25 | 1 | 16 | 9 | 769657 | 0 | 0 | 0 | 4294.533 |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000001 | 0 | 0 | 0 | 20514.9 |
| $War_1$ | 3 | 20 | 20 | 1 | 16 1 1 1 | 1 | 64 | 0 | 0 | 0 | **2.474** |
| $War_2$ | 4 | 30 | 30 | 1 | 23 3 1 1 1 | 1 | 31144 | 0 | 0 | 0 | 312.156 |
| $War_3$ | 6 | 30 | 30 | 1 | 12 1 3 1 | 13 | 20 | 0 | 0 | 0 | **2.222** |
| $War_e$ | 3 | 10 | 10 | 1 | 10 | 0 | 27 | 0 | 0 | 0 | **1.859** |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 31 | 0 | 0 | 0 | **1.75** |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 15 | 0 | 0 | 0 | **1.804** |

Table 6: Branch and bound with non-opening products, partioning and subsumption, total time taken 8m1.829s

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 45 | 50 | 50 | 1 | 42 | 8 | 700 | 467 | 1982 | 0 | 70.618 |
| $GP_2$ | 40 | 50 | 50 | 1 | 48 | 2 | 46821 | 43580 | 194366 | 0 | 2621.512 |
| $GP_3$ | 40 | 50 | 50 | 1 | 50 | 0 | 38762 | 35929 | 148346 | 0 | 2097.014 |
| $GP_4$ | 30 | 50 | 50 | 1 | 37 | 13 | 1656 | 1244 | 2886 | 0 | 90.657 |
| $GP_5$ | 95 | 100 | 100 | 1 | 100 | 0 | 35485 | 33643 | 321356 | 0 | 5189.517 |
| $GP_6$ | 75 | 100 | 100 | 1 | 100 | 0 | 407044 | 396710 | 4417382 | 0 | 65854.026 |
| $GP_7$ | | | | | | | | | | | 80200.525 |
| $GP_8$ | | | | | | | | | | | 86656.389 |
| $GPN_5$ | 12 | 20 | 30 | 1 | 20 | 10 | 6124 | 4654 | 5169 | 0 | 157.643 |
| $GPN_6$ | 12 | 20 | 30 | 1 | 23 | 7 | 15981 | 13593 | 15723 | 0 | 347.484 |
| $GPN_7$ | 10 | 25 | 60 | 1 | 32 | 28 | 228697 | 183285 | 218004 | 0 | 4986.602 |
| $GPN_8$ | 0 | 25 | 60 | 0 | 40 | 20 | 1000000 | 944706 | 1560918 | 0 | 25358.762 |
| $Shaw_5$ | 14 | 20 | 20 | 1 | 18 | 2 | 7311 | 5720 | 4422 | 0 | 149.031 |
| $Shaw_6$ | 14 | 20 | 20 | 1 | 19 | 1 | 10133 | 7981 | 5475 | 0 | 191.548 |
| $Shaw_7$ | 13 | 20 | 20 | 1 | 18 | 2 | 5374 | 3901 | 3092 | 0 | 119.179 |
| $Shaw_8$ | 14 | 20 | 20 | 1 | 19 | 1 | 12207 | 9950 | 7341 | 0 | 223.549 |
| $Shaw_9$ | 13 | 20 | 20 | 1 | 18 | 2 | 4558 | 3316 | 2541 | 0 | 105.726 |
| $SP_1$ | 9 | 25 | 25 | 1 | 16 | 9 | 12678 | 8231 | 822 | 0 | 166.169 |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000000 | 628304 | 272479 | 0 | 19229.413 |
| $War_1$ | 3 | 20 | 20 | 1 | 16 1 1 1 | 1 | 621 | 245 | 132 | 0 | 74.989 |
| $War_2$ | 4 | 30 | 30 | 1 | 23 3 1 1 1 | 1 | 12362 | 5341 | 2906 | 0 | 329.45 |
| $War_3$ | 6 | 30 | 30 | 1 | 12 1 3 1 | 13 | 19 | 1 | 0 | 0 | 62.743 |
| $War_e$ | 3 | 10 | 10 | 1 | 10 | 0 | 109 | 29 | 12 | 0 | 18.812 |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 54 | 15 | 11 | 0 | 33.405 |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 30 | 7 | 3 | 0 | 32.205 |

Table 7: $A\star$ with non-opening products, partioning and subsumption, total time taken 4m54s

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 45 | 50 | 50 | 1 | 42 | 8 | 354 | 204 | 126 | 235 | 33.302 |
| $GP_2$ | 40 | 50 | 50 | 1 | 48 | 2 | 3924 | 2826 | 1889 | 2941 | **152.844** |
| $GP_3$ | 40 | 50 | 50 | 1 | 50 | 0 | 2545 | 1788 | 1725 | 2434 | **119.512** |
| $GP_4$ | 30 | 50 | 50 | 1 | 37 | 13 | 64 | 20 | 37 | 44 | 21.886 |
| $GP_5$ | 95 | 100 | 100 | 1 | 100 | 0 | 10500 | 8586 | 36816 | 38633 | **1041.119** |
| $GP_6$ | 75 | 100 | 100 | 1 | 100 | 0 | 1268 | 679 | 789 | 1280 | **184.649** |
| $GP_7$ | 75 | 100 | 100 | 1 | 99 | 1 | 4289 | 3209 | 4620 | 5602 | **358.734** |
| $GP_8$ | 60 | 100 | 100 | 1 | 96 | 4 | 940 | 532 | 860 | 1173 | **135.873** |
| $GPN_5$ | 12 | 20 | 30 | 1 | 20 | 10 | 891 | 535 | 188 | 527 | 35.906 |
| $GPN_6$ | 12 | 20 | 30 | 1 | 23 | 7 | 1782 | 1250 | 644 | 1156 | 56.884 |
| $GPN_7$ | 10 | 25 | 60 | 1 | 32 | 28 | 2623 | 1963 | 1264 | 1893 | 89.854 |
| $GPN_8$ | 16 | 25 | 60 | 1 | 40 | 20 | 990740 | 888232 | 739103 | 841576 | 21928.027 |
| $Shaw_5$ | 14 | 20 | 20 | 1 | 18 | 2 | 8681 | 6611 | 2766 | 4823 | 160.549 |
| $Shaw_6$ | 14 | 20 | 20 | 1 | 19 | 1 | 5901 | 4095 | 915 | 2708 | 111.87 |
| $Shaw_7$ | 13 | 20 | 20 | 1 | 18 | 2 | 3124 | 2107 | 781 | 1786 | 73.32 |
| $Shaw_8$ | 14 | 20 | 20 | 1 | 19 | 1 | 7588 | 5647 | 1940 | 3868 | 141.236 |
| $Shaw_9$ | 13 | 20 | 20 | 1 | 18 | 2 | 2807 | 1896 | 597 | 1495 | 66.308 |
| $SP_1$ | 9 | 25 | 25 | 1 | 16 | 9 | 5200 | 3153 | 230 | 2262 | 97.219 |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000000 | 770448 | 93987 | 323539 | 19408.691 |
| $War_1$ | 3 | 20 | 20 | 1 | 16 1 1 1 | 1 | 51 | 1 | 5 | 37 | 63.393 |
| $War_2$ | 4 | 30 | 30 | 1 | 23 3 1 1 1 | 1 | 747 | 197 | 68 | 591 | 98.772 |
| $War_3$ | 6 | 30 | 30 | 1 | 12 1 3 1 | 13 | 18 | 0 | 0 | 1 | 62.453 |
| $War_e$ | 3 | 10 | 10 | 1 | 10 | 0 | 28 | 0 | 1 | 20 | 17.493 |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 30 | 0 | 2 | 24 | 32.502 |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 16 | 0 | 1 | 9 | 32.518 |

Table 8: Stepwise with non-opening products, partioning and subsumption, total time taken 44.553

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 45 | 50 | 50 | 1 | 42 | 8 | 364 | 204 | 158 | 0 | 34.043 |
| $GP_2$ | 40 | 50 | 50 | 1 | 48 | 2 | 4294 | 3313 | 5344 | 0 | 207.092 |
| $GP_3$ | 40 | 50 | 50 | 1 | 50 | 0 | 3355 | 2511 | 3274 | 0 | 157.766 |
| $GP_4$ | 30 | 50 | 50 | 1 | 37 | 13 | 150 | 40 | 89 | 0 | 26.398 |
| $GP_5$ | 95 | 100 | 100 | 1 | 100 | 0 | 11648 | 10398 | 69075 | 0 | 1367.949 |
| $GP_6$ | 75 | 100 | 100 | 1 | 100 | 0 | 78149 | 73969 | 532744 | 0 | 8485.849 |
| $GP_7$ | 75 | 100 | 100 | 1 | 99 | 1 | 36519 | 32394 | 123564 | 0 | 2911.617 |
| $GP_8$ | 60 | 100 | 100 | 1 | 96 | 4 | 209025 | 193862 | 672128 | 0 | 13327.962 |
| $GPN_5$ | 12 | 20 | 30 | 1 | 20 | 10 | 529 | 317 | 131 | 0 | 28.435 |
| $GPN_6$ | 12 | 20 | 30 | 1 | 23 | 7 | 1050 | 735 | 445 | 0 | **41.208** |
| $GPN_7$ | 10 | 25 | 60 | 1 | 32 | 28 | 2644 | 1963 | 1275 | 0 | **82.711** |
| $GPN_8$ | 16 | 25 | 60 | 1 | 40 | 20 | 627095 | 568016 | 540477 | 0 | **12098.695** |
| $Shaw_5$ | 14 | 20 | 20 | 1 | 18 | 2 | 4442 | 3511 | 1821 | 0 | **93.047** |
| $Shaw_6$ | 14 | 20 | 20 | 1 | 19 | 1 | 3367 | 2441 | 678 | 0 | **69.678** |
| $Shaw_7$ | 13 | 20 | 20 | 1 | 18 | 2 | 1812 | 1283 | 553 | 0 | **50.738** |
| $Shaw_8$ | 14 | 20 | 20 | 1 | 19 | 1 | 3917 | 3046 | 1222 | 0 | **79.231** |
| $Shaw_9$ | 13 | 20 | 20 | 1 | 18 | 2 | 1555 | 1083 | 406 | 0 | **44.974** |
| $SP_1$ | 9 | 25 | 25 | 1 | 16 | 9 | 5404 | 3199 | 231 | 0 | **93.001** |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000000 | 767845 | 172841 | 0 | 16402.846 |
| $War_1$ | 3 | 20 | 20 | 1 | 16 1 1 1 | 1 | 106 | 4 | 21 | 0 | 65.836 |
| $War_2$ | 4 | 30 | 30 | 1 | 23 3 1 1 1 | 1 | 1013 | 278 | 115 | 0 | 104.376 |
| $War_3$ | 6 | 30 | 30 | 1 | 12 1 3 1 | 13 | 39 | 0 | 0 | 0 | 62.793 |
| $War_e$ | 3 | 10 | 10 | 1 | 10 | 0 | 37 | 0 | 2 | 0 | 18.043 |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 45 | 0 | 5 | 0 | 32.932 |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 24 | 0 | 2 | 0 | 32.642 |

Table 9: Backwards with non-opening products, partioning and subsumption, total time taken 55.948

| Instance | Best | $\|C\|$ | $\|P\|$ | Optimal | Parts | Subs | Steps | Reused | Opening | Fail | Time(ms) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $GP_1$ | 45 | 50 | 50 | 1 | 42 | 8 | 364 | 204 | 158 | 235 | 22.5 |
| $GP_2$ | 40 | 50 | 50 | 1 | 48 | 2 | 4542 | 3327 | 2950 | 4023 | 195.643 |
| $GP_3$ | 40 | 50 | 50 | 1 | 50 | 0 | 3664 | 2673 | 3197 | 4040 | 177.85 |
| $GP_4$ | 30 | 50 | 50 | 1 | 37 | 13 | 150 | 40 | 89 | 88 | **12.558** |
| $GP_5$ | 95 | 100 | 100 | 1 | 100 | 0 | 12876 | 11062 | 69195 | 70711 | 1893.369 |
| $GP_6$ | 75 | 100 | 100 | 1 | 100 | 0 | 13813 | 12562 | 74808 | 75660 | 1822.498 |
| $GP_7$ | 75 | 100 | 100 | 1 | 99 | 1 | 14933 | 12365 | 28863 | 31036 | 1232.798 |
| $GP_8$ | 60 | 100 | 100 | 1 | 96 | 4 | 22644 | 20199 | 62836 | 64802 | 1794.683 |
| $GPN_5$ | 12 | 20 | 30 | 1 | 20 | 10 | 798 | 478 | 180 | 462 | **22.823** |
| $GPN_6$ | 12 | 20 | 30 | 1 | 23 | 7 | 1559 | 1091 | 613 | 1014 | 44.007 |
| $GPN_7$ | 10 | 25 | 60 | 1 | 32 | 28 | 2644 | 1963 | 1275 | 1893 | 93.349 |
| $GPN_8$ | 16 | 25 | 60 | 1 | 40 | 20 | 867577 | 784261 | 712513 | 795711 | 24499.352 |
| $Shaw_5$ | 14 | 20 | 20 | 1 | 18 | 2 | 7965 | 6176 | 2719 | 4475 | 166.714 |
| $Shaw_6$ | 14 | 20 | 20 | 1 | 19 | 1 | 5122 | 3636 | 869 | 2320 | 101.298 |
| $Shaw_7$ | 13 | 20 | 20 | 1 | 18 | 2 | 2714 | 1875 | 735 | 1541 | 74.334 |
| $Shaw_8$ | 14 | 20 | 20 | 1 | 19 | 1 | 6336 | 4811 | 1749 | 3239 | 131.642 |
| $Shaw_9$ | 13 | 20 | 20 | 1 | 18 | 2 | 1879 | 1287 | 446 | 986 | 42.318 |
| $SP_1$ | 9 | 25 | 25 | 1 | 16 | 9 | 5361 | 3198 | 231 | 2347 | 107.482 |
| $SP_2$ | 0 | 50 | 50 | 0 | 44 | 6 | 1000000 | 792030 | 192651 | 400571 | 18546.65 |
| $War_1$ | 3 | 20 | 20 | 1 | 16 1 1 1 | 1 | 88 | 2 | 13 | 46 | 5.742 |
| $War_2$ | 4 | 30 | 30 | 1 | 23 3 1 1 1 | 1 | 993 | 278 | 111 | 746 | **35.381** |
| $War_3$ | 6 | 30 | 30 | 1 | 12 1 3 1 | 13 | 39 | 0 | 0 | 2 | 3.63 |
| $War_e$ | 3 | 10 | 10 | 1 | 10 | 0 | 37 | 0 | 2 | 20 | 3.194 |
| $Ware_1$ | 3 | 10 | 10 | 1 | 7 3 | 0 | 45 | 0 | 5 | 25 | 3.718 |
| $Ware_2$ | 3 | 10 | 10 | 1 | 6 3 | 1 | 24 | 0 | 2 | 9 | 2.637 |

Table 10: Binarychop with non-opening products, partioning and subsumption, total time taken 51.068

# 7 Discussion

## 7.1 Interpretation of Results

As expected the exhaustive algorithm failed to obtain an optimal solution for the majority of the tests, terminating early due to exceeding the 1 million step limit. This due to that fact that exhaustive search could only solve problems with $< 9$ products or the search space would exceed the 1 million allocated steps. To test this I choose 3 tests specifically with small search spaces $\leq 10!$, where two completed successfully, and the one equaling $10!$ failed.

Branch and bound, successfully obtained the solution to 19 of the 25 problems tested. This demonstrates that .bounding. and .pruning. is very effective at reducing the search space needing to be explored to obtain an optimal solution. It would have been interesting if I had time to implement an alternative heuristic for comparison. But it appears the choosing the product with the least active customers first was very successful, as it solved a further 17 problems on top of exhaustive with the same step limit. Also it appears the branch and bound is very efficient under some circumstances as it found the solution to $GP_1$ the fastest. I would assume $GP_1$ having a very obvious path to the solution and not requiring many past solutions needing re-use as it beat Astar and its variants.

Retaining partial solutions was the only enhancment added to branch and bound, perhaps the simplist change made to our program, yet allowed the ability to solve the majority of the selected problem isntances (perhaps all of them but $SP_2$ if more then 12MB of memory could be used, and ¿ 1 million steps allowed). Table 7 shows 21 of the 25 tests obtained optimal solutions, which is a fair improvement upon the branch and bound. This really does go to show that simple solutions are always tthe best. It is interesting that if a solution was obtained, it was done well within the 1,000,000 steps allowed. This must be due to the nature of the search tree, if a solution was not obtained then the search must not be storing many used sets. Due to the limitation on memory usage two of the tests failed, which may have completed if allowed to finish executing.

Backwards, stepwise and binarychop all performed very well but it seems that how fast a solution is obtained is directly rated to where the optimal solution lays in the domain from L to U. If close to L, stepwise is extremly fast, but also in some cases even when the best value is close to —P— stepwise still performs the best. This could be due to the fact that start at the back is harder then starting at the front retaining partial solutions as you step forwards. The opposite goes for backwards which appeared to be extremly fast when the best value was ¿ $|P|/2$ indicated by table 9, tests $GPN_6$ to $SP_1$. Binarychop is extremly good at finding solutions if the lay in the middle, indicated be a single result $GPN_5$ in table 10.

The fastest algorithm over all was stepwise, this is most likly due to its ability to continually reuse the stack loopup table. I guess the same logic can be applied from going to dynamic prgomming $A\star$ from branch and bound, in that past solutions are constantly reused reducing recomputing.

## 7.2 Optimisations

After initial profiling both O(n) methods in the set implementation, setCardinal and setNext accounted for up to 40solution. So both this operations were optimised using lookup tables. Which improved the search time by a small margin.

The bit string sets also came in handy in quickly generate hash values, as the value of the string if read as an integer gives a high probability of being unique.

Retaining successful sets in $A\star$ was ignored as backwards and binary when moving backwards, started with a new stacks table after each iteration.

The sliding window technique was used where ever possable, eg in $maxOpen$.

Avoiding excess calls to malloc and free was avoided by placing sets on the stack.

## 7.3 What I Learnt

What I learnt I have always struggled with recursion and knew I had to expose myself to it so that I could utilise it where it appeared more appropriate than iteraton. Therefore, the natural thing to do was to choose MOSP as my 3rd year project as I would have no option but to learn how to use it correctly. Now that the project is complete and I have implemented four algorithms solely based on traversing a tree generated recursively. This project has allowed me to feel more comfortable with using recursion, however, I still believe I will require examples to work from when writing more complicated recursive algorithms. It is hoped with more time and practice I will find recursion as simple as iteration.

Keeping in mind the advice, 'use pen and paper before you touch the computer', I began my project on paper. Although, I would normally use the computer to find the best implementations for the datatypes, I found that using paper allowed me to be thorough and hence choose the best implementations. This was a valuable lesson and will be used as a starting point for further projects.

I reluctantly learnt to use long variables names, to which I have no grown a liking to you, as after a few weeks of not looking at the code, I quickly forgot what I was trying to do. Variables names which convey meaning facilitate the understanding of code.

I accidently issued rm * in the source code directory 8 hours before the project was due, with my last backup 5 days old. So, I will always make frequent backups from now on.

## 7.4 Things I want to do

Unfortunately due to lack of time I was unable to implement a single heuristic of my own. Decreasing the upper bound would of been nice.

Also moving onto a 64 bit integer for the set DT would of been fun to see if it improved performance. Unfortuently I could not do this due to lack of time.

## 7.5 Did I reach my goals

I feel that I am happy with what I achieved, as always with alittle more time, or perhaps better time management I could of. But all in all, I learnt alot from this project, and Im extremly lucky to had Maria teach it me. Thanks Maria for the semester!

# 8 References

[1] M. Garcia de la Banda and P. J. Stuckey. Dynamic Programming to Minimize the Maximum Number of Open Stacks. IJCAI05 Constraint Modelling Challenge entry.