# On Solutions to a Seating Problem *

Fei Dai, Mohit Dhawan, Changfu Wu and Jie Wu
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431

**Abstract**

The seating problem, proposed by Wu [2], deals with the following assignment of $n$ couples in a row: $n$ couples are seated in one row. Each person, except two seated at the two ends, has two neighbors. It is required that each neighbor of person $k$ should either have the same gender or be the spouse of $k$. In this paper, we consider several solutions to the problem, both direct and indirect. Simulations have been conducted to compare these solutions.

**Keywords**: *Seating problem, algorithm, recursion tree, recursive function, simulation.*

## 1 Introduction

The *seating problem* [2] is a combinatorial problem defined as follows: $n$ couples $\{(h_k, w_k)\}$, where $h_k$ and $w_k$ are husband and wife of couple $k$, go to a movie theater and are seated in one row. Each person, except two seated at the two ends, has two neighbors. It is required that each neighbor of a person $k$ should either have the same gender as $k$ or be the spouse of $k$. A *reasonable pattern* is an assignment of $n$ couples without indices that meet the above requirement. For example, when $n = 3$ all reasonable patterns are as follows:

$$\begin{array}{cccc} wwwhhh & whhwwh & whhhww & hhwwwh \\ hhhwww & hwwhhw & hwwwhh & wwhhhw \end{array}$$

while pattern "$whwwhh$" is *unreasonable* because the first $h$ has two neighboring $w$'s, only one of which could be his wife.

A classical, but different, seating problem is the following [1]: $n$ couples are seated around a table in such a way that two neighbors of each person $k$ are the different gender of $k$ and they are not the spouse of $k$. An *indirect solution* enumerates all patterns, and then eliminates all unreasonable (not reasonable) patterns by printing out reasonable patterns only. A *direct solution* systematically generates all reasonable patterns directly. In this paper, we consider four possible solutions to the seating problem. Each solution is discussed by using both direct and indirect approaches. In each solution, bit "0" represents a "wife" and "1" represents a "husband". A reasonable pattern is a 0-1 string that satisfies the following two constraints: (a) exact $n$ "0"s and $n$ "1"s. (b) no isolated 0 or 1 bit, except at two ends.

Four methods are considered in this paper. In the *binary tree method*, a binary tree structure is used to generate all possible seating patterns. Each node in the tree is assigned either 0 or 1. The *state machine method* can be considered as a special case of the binary tree method, where the last bit of a partially generated string is used as state information to decide the next one or two bits to generate reasonable patterns. In the *shifting method*, a special reasonable pattern is generated first, where all 1's are assigned to the right hand side of all 0's. Then 1's are systematically shifted from left to right. The last reasonable pattern generated is all 1's assigned to the left hand side of all 0's. The last method is based on *segments*. A 0-segment (1-segment) is a sequence of all "0"s (all "1"s) separated by "1"s ("0"s), and a pattern is a sequence of alternating 0-segments and 1-segments. In a reasonable pattern, the length of each segment is greater than or equal to 2, except for the first and last segments.

## 2 Preliminaries

Given $n$ couples $\{(h_k, w_k)\}$, a *seating* is a permutation of $n$ couples $\{h_k, w_k\}$. A *reasonable seating* is a seating with all its 2-bit-substrings in the form: (1) $h_i h_j$, (2) $w_i w_j$, (3) $h_i w_i$, or (4) $w_j h_j$, where $1 \leq i, j \leq n$. A *valid pattern* is a 0-1 string of length $2n$ with exactly $n$ "1"s and $n$ "0"s, where each "1" represents a husband, and each "0" represents a wife. Two seatings are equivalent if they correspond to the same permutation of $n$ couples without considering indices. Therefore, each seating can be represented by a pattern of 0-1 string of length $2n$:

*This work was supported in part by NSF grants CCR 9900646 and ANI 0073736 and a grant from Motorola Inc. Email: {fdai,mdhawan}@cse.fau.edu, cwu02588@fau.edu, jie@cse.fau.edu.

**Definition 1 (Reasonable Pattern)** *Given $n$ couples, a reasonable pattern is a valid pattern that corresponds to a reasonable seating.*

**Theorem 1** *A valid pattern $P$ is a reasonable pattern if and only if it has no 3-bit-substring "101" or "010".*

**Proof**: The necessity is obvious: it is impossible to find a corresponding substring in a reasonable seating corresponding "101" or "010". For the sufficiency, we show that $P$ corresponds to a reasonable seating. Because pattern $P$ has no 3-bit-substring "101", each "0" either has no neighboring "1" or is adjacent to a unique "1". Similarly, each "1" either has no neighboring "0" or is dedicated to a unique "0". Suppose the number of "0"s with neighboring "1"s (same as the number of "1"s with neighboring "0"s) is $m$, for each matching pair of "0" and "1", we assign a couple $(h_i, w_i)$, $1 \leq i \leq m$. then we assign $h_j$ $(m < j \leq n)$ to the rest $n - m$ "1"s, assign $w_j$ $(m < j \leq n)$ to the rest $n - m$ "0"s, and get a reasonable seating. □

In a 0-1 string, a *1-segment* is a sequence of "1"s separated by "0"s, and a *0-segment* is a sequence of "0"s separated by "1"s. For example, pattern "110001" can be separated into 3 segments: "11", "000", and "1". The first segment and the last segment of a 0-1 string are *end segments*, the other segments are *internal segments*. In the above example, "11" and "1" are end segments and "000" is an internal segment.

**Theorem 2** *A valid pattern is a reasonable pattern if and only if all its internal segments are of length greater than or equal to 2.*

**Proof**: An internal segment of length 1 implies a substring "101" or "010". On the other hand, if all internal segments have length greater than or equal to 2, there is no such a substring. □

We use $e'$ to denote the *complement element* of $e \in \{0, 1\}$. Specifically, $1' = 0$ and $0' = 1$. The complement of a 0-1 string $P = p_1 p_2 \ldots p_n$ is defined as $P' = p'_1 p'_2 \ldots p'_n$. Apparently, the complement operation is symmetric; that is, $(P')' = P$.

**Theorem 3** *$P$ is a reasonable pattern if and only if its complement $P'$ is a reasonable pattern.*

**Proof**: If $P$ is a reasonable pattern, it has no substring "101" or "010", which implies that $P'$ has no substring "010" or "101", and therefore, is a reasonable pattern. If $P'$ is a reasonable pattern, $P = (P')'$ is also a reasonable pattern. □

Given $n$ couples, the seating problem is to print out all the reasonable patterns. From Theorem 3, we only need to enumerate every reasonable pattern starting with "1" and then print it out together with its complement pattern. Algorithms to solve the seating problem can be either direct or indirect. The indirect algorithms enumerate all valid patterns by traversing a searching tree in the problem space. It generates every valid pattern and print it out when it is a reasonable pattern, as in the following procedure CHECK ($P$), where $P$ is a valid pattern and PRINT ($P$) prints out both $P$ and $P'$.

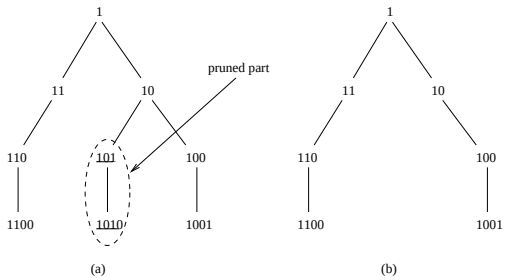



Figure 1: The binary tree method: (a) The indirect approach generates all valid pattern prefixes. (b) The direct approach generates only reasonable pattern prefixes.

CHECK($P$)
1: **if** $P$ has no substring "101" or "010" **then**
2: PRINT($P$)

In the direct solutions, the searching tree is much smaller because all the branches that cannot lead to reasonable patterns are *pruned*. It is also not necessary to check the patterns generated by the direct solutions because only reasonable patterns can be generated.

## 3 Binary Tree Method

The most straightforward way of enumerating all possible valid patterns is to explicitly build a searching tree. At each step of the searching process, at most two decisions are feasible: assigning next seat to male ("1") or female ("0"). Therefore, the resultant searching tree is a binary tree.

**Indirect approach**. Figure 1 (a) shows a sample binary tree generated by the indirect approach for the seating problem with 2 couples. Each node of the binary tree is a triple ($v$, $l$, $r$), where $v$ represents both node id and its pattern prefix and $l$ ($r$) is the left (right) child. This binary tree only contains nodes with *valid prefixes*; that is, for $n$ couples, the number of "1"s (also "0"s) in a pattern prefix is at most $n$. A pattern prefix with $n$ "1"s and $n$ "0"s is a leaf node of the binary tree and forms a valid pattern. For example, when $n = 2$, "1", "11", and "101" are valid prefixes; "1100" and "1010" are valid patterns; "111" is not a valid prefix and does not appear in the binary tree.

The core of the following indirect algorithm is the procedure VISIT, which recursively builds the binary tree from root "1". VISIT takes three parameters: $v$ is the node to grow new branches,

$h$ is the number of "1"s (husbands) left to form a valid pattern, and $w$ is the number of "0"s (wives) left. Three conditions are tested at each node: If the current valid prefix is a valid pattern, this pattern is checked and, if it is reasonable, printed. If it is possible to create a left (right) child with valid prefix, the left (right) child is created and visited in a depth-first manner. Here we suppose function NEWNODE ($v$) will create and return a new node ($v$, NIL, NIL). $v1$ represents a concatenation of $v$ and 1, i.e., $v||1$.

TREE ($n$)
1: $root \leftarrow$ NEWNODE ("1")
2: VISIT ($root$, $n$, $n$)

VISIT ($v$, $h$, $w$)
1: **if** $h = w = 0$ **then**
2: CHECK ($v$)
3: **if** $h > 0$ **then**
4: $l(v) \leftarrow$ NEWNODE ($v1$)
5: VISIT ($l(v)$, $h-1$, $w$)
6: **if** $w > 0$ **then**
7: $r(v) \leftarrow$ NEWNODE ($v0$)
8: VISIT ($r(v)$, $h$, $w-1$)

**Direct approach**. The indirect approach is straightforward but not efficient. For example, in Figure 1 (a), branch "101"-"1010" does not contribute in generating reasonable patterns and can be safely pruned. We call a pattern prefix with subsequence "101" or "010" an *unreasonable prefix* and otherwise a *reasonable prefix*. Clearly, a reasonable pattern cannot be generated from a unreasonable prefix. Figure 1 (b) shows the binary tree built by the direct approach. If we create only nodes with reasonable prefixes, we will generate all reasonable patterns and reasonable patterns only.

The direct approach of the binary tree solution is very similar to the indirect approach, except the recursive procedure VISIT is modified as the following VISIT*. The revised procedure also checks three conditions, but with extra constraints to prevent it from creating unreasonable prefixes. Since the new procedure always generates reasonable patterns, the patterns are printed directly without checking. Function RIGHT($v$, 2) returns the 2 rightmost bits of $v$.

VISIT* ($v$, $h$, $w$)
1: **if** $h = w = 0$ **then**
2: PRINT ($v$)
3: **if** $h > 0$ **and** RIGHT($v$, 2) $\neq$ "10" **then**
4: $l(v) \leftarrow$ NEWNODE ($v1$)
5: VISIT* ($l(v)$, $h-1$, $w$)
6: **if** $w > 0$ **and** RIGHT($v$, 2) $\neq$ "01" **then**
7: $r(v) \leftarrow$ NEWNODE ($v0$)
8: VISIT* ($r(v)$, $h$, $w-1$)

# 4 State Machine Method

The state machine method is more efficient than the binary tree method because it searches the solution space without actually generating a binary tree. It also condenses several steps into one step to reduce the number of nodes in the recursion tree.

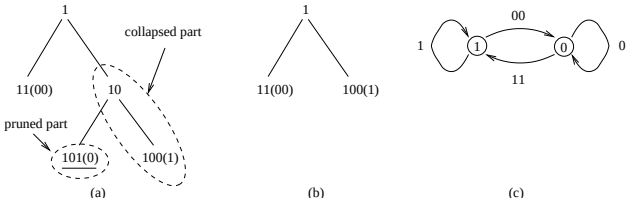



Figure 2: The state machine method: (a) In the indirect approach, single-thread segments collapse into single nodes. (b) In the direct approach, pruning produces a new single-thread segment, which is also collapsed. (c) The state machine used by the direct approach.

**Indirect approach**. The binary tree method is inefficient for two reasons: the explicit binary tree and single-thread segments. Explicitly building a binary tree consumes enormous amount of memory as well as CPU time, and is also unnecessary. In the recursive procedure VISIT ($v$, $h$, $w$), all tasks are based on $v$, $h$, and $w$, and have nothing to do with rest of the binary tree. In the following three methods, there will be no explicit tree. Instead, the recursive procedures can traverse the virtual searching trees, navigating via three parameters $v$ (prefix), $h$ ("1"s left), and $w$ ("0"s left). For example, in Figure 2 (a), root "1" corresponds to the first level invocation FIRE ("1", 1, 2), which first spawns node "11", corresponding to the second level invocation FIRE ("11", 0, 2). After the search in this branch ends, the control goes back to root, and goes down to another node "10". After the algorithm completes, all invocations to FIRE form a binary recursion tree.

A *single-thread segment* is a part of the recursion tree consisting of a parent, its only child, its child's only child, etc. For example, in Figure 1, nodes "11", "110", and "1100' form a single-thread segment. A single-thread segment can be *collapsed* into one node, because the status of the last node in the segment is already determined at the first node. For example, considering two couples, from prefix "11" comes only one valid pattern "1100" (no more "1"s available). By collapsing this segment into one node ("11(00)" in Figure 2 (a)), we reduce the number of invocations to the recursive procedure FIRE.

The core of the following indirect approach is the procedure FIRE. Similar to VISIT in the previous method, it visits the left and right children in a depth-first manner. It collapses single-thread segments in this way: If there are only "0"s left, they are all appended to the current prefix to form a valid pattern. If there are only "1"s left, they are treated in the same way. Notation "$1^{(x)}$" ("$0^{(x)}$") represents $x$ continuous "1"s ("0"s).

STATE ($n$)
1: FIRE ("1", $n-1$, $n$)

FIRE ($v$, $h$, $w$)
1: **if** $h = 0$ **then**
2: CHECK ($v0^{(w)}$)
3: **else if** $w = 0$ **then**
4: CHECK ($v1^{(h)}$)
5: **else**

6: FIRE ($v1$, $h-1$, $w$)
7: FIRE ($v0$, $h$, $w-1$)

**Direct approach**. As shown in Figure 2, when an unreasonable branch is pruned from the recursion tree, the newly created single-thread segment can be further collapsed. That is, because node "10" has only one child with reasonable prefix "100(1)", they collapse into one after pruning.

The direct approach uses a *state machine* to achieve both pruning and collapsing. In the following recursive procedure FIRE*, new events are *fired* based on current *state* (i.e., the last bit of current prefix), number of "1"s left ($h$), and number of "0"s left ($w$). As shown in Figure 2 (c), if the last bit is "1", the following bits are either "1" or "00". "01" is forbidden, because it will produce an unreasonable prefix. If the last bit is "0", the following bits are either "0" or "11". The other part (collapse with only "0"s or "1"s left) is exactly the same as in FIRE.

FIRE* ($v$, $h$, $w$)
1: **if** $h = 0$ **then**
2: PRINT ($v0^{(w)}$)
3: **else if** $w = 0$ **then**
4: PRINT ($v1^{(h)}$)
5: **else if** RIGHT($v$, 1) = "1" **then**
6: FIRE* ($v1$, $h-1$, $w$)
7: **if** $w \geq 2$ **then**
8: FIRE* ($v00$, $h$, $w-2$)
9: **else**
10: FIRE* ($v0$, $h$, $w-1$)
11: **if** $h \geq 2$ **then**
12: FIRE* ($v11$, $h-2$, $w$)

# 5 Shifting Method

Unlike the other three methods, the shifting method does not use pattern prefixes to synthesize patterns. The indirect approach starts from a valid pattern and switches to other valid patterns by shifting the position of "1"s in the original pattern. The direct approach starts from a reasonable pattern and switches to other reasonable patterns in a similar but restricted way. Therefore, in the direct approach, the number of nodes in the recursion tree is exactly the number of reasonable patterns.

**Indirect approach**. Figure 3 (a) shows the recursion tree of the indirect shifting method. The original pattern is the one with all "0"s in the left side and all "1"s in the right side; that is, "0011". However, because we only need to generate those patterns starting with "1", the first "1" is shifted to the first slot to generate the root node, "1001". Then the second "1" can be shifted left to the second and third slots to generate "1100" and "1010", respectively.

The core of the following indirect algorithm is the recursive procedure MOVE ($v$, $h$, $w$). The three parameters represent a valid pattern: the first part of the pattern is $v$, the second part is $w$ "0"s, and the third part is $h$ "1"s ($v0^{(w)}1^{(h)}$). For example, the corresponding pattern for MOVE ("1001", 3, 3) is "1001000111". When invoked, MOVE first checks the current pattern and prints

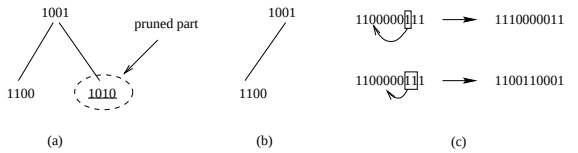



Figure 3: The shifting method: (a) the indirect approach, (b) the direct approach, and (c) two legal shiftings in the direct approach.

it if reasonable. Then it shifts the first "1" in the third part of the pattern, and recursively processes every newly generated pattern.

SHIFT($n$)
1: MOVE("1", $n-1$, $n$)

MOVE($v$, $h$, $w$)
1: CHECK ($v0^{(w)}1^{(h)}$)
2: **if** $h \neq 0$ **then**
3: **for** $k \leftarrow 0$ **to** $w-1$ **do**
4: MOVE ($v0^{(k)}1$, $h-1$, $w-k$)

Note that the recursion tree of MOVE is not necessarily a binary tree. In pattern $v0^{(w)}1^{(h)}$, the first "1" in the third part can be shifted to $w$ positions, and therefore, may has $w$ children.

**Direct approach**. In the following revised procedure, MOVE*, only two categories of shifting are allowed, as shown in Figure 3 (c): (1) moving one "1" and putting it right behind $v$, and (2) moving two "1"s together and leaving at least two "0"s between them and $v$. Note that shifting category (2) is the result of pruning and collapsing regarding to the indirect approach. Figure 3 (b) shows the example of two couples. From "1001", only shifting (1) is feasible because $h < 2$.

MOVE*($v$, $h$, $w$)
1: PRINT ($v0^{(w)}1^{(h)}$)
2: **if** $h \geq 1$ **then**
3: MOVE* ($v1$, $h-1$, $w$)
4: **else if** $h \geq 2$ **and** $w \geq 2$ **then**
5: **for** $k \leftarrow 2$ **to** $w-1$ **do**
6: MOVE* ($v0^{(k)}11$, $h-2$, $w-k$)

The shifting method is slightly more efficient than the state machine method because it has fewer nodes in the recursion tree. Its recursion tree is also *flatter* than that of the state machine method; that is, each node has more children and it takes fewer steps to generate a reasonable pattern. For example, in order to get pattern "110001", the shifting method takes 2 steps: "100011" → "110001"; while the state machine method takes 4 steps: "1" → "11" → "1100" → "11000(1)".

# 6 Segment Method

The segment method treats a pattern as a sequence of alternating 1-segments and 0-segments. In the indirect approach, the length of each segment can be any positive number. In the direct

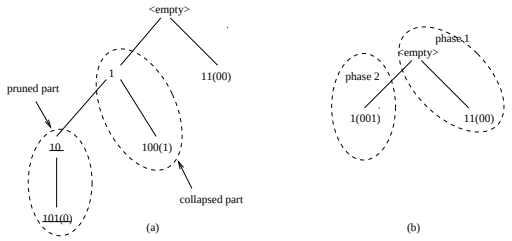



Figure 4: The segment method: (a) The indirect approach has no constraint on segment length. (b) The direct approach requires segment length of at least two.

approach, the length of every internal segment is greater than or equal to 2.

**Indirect approach**. The segment method generates patterns by allocating $n$ "1"s and $n$ "0"s to alternating segments. In the following indirect algorithm, procedure HUSBAND allocates 1-segments and WIFE allocates 0-segments. For example, in Figure 4 (a), HUSBAND has two options: "1" or "11". From "1", WIFE also has two options: "10" and "100". By searching all these options, the indirect approach can enumerate every valid pattern.

SEGMENT ($n$)
1: HUSBAND ($\emptyset$, $n$, $n$)

HUSBAND ($v$, $h$, $w$)
1: **if** $w = 0$ **then**
2: CHECK ($v1^{(h)}$)
3: **else**
4: **for** $k \leftarrow 1$ **to** $h$ **do**
5: WIFE ($v1^{(k)}$, $h-k$, $w$)

WIFE ($v$, $h$, $w$)
1: **if** $h = 0$ **then**
2: CHECK($v0^{(w)}$)
3: **else**
4: **for** $k \leftarrow 1$ **to** $w$ **do**
5: HUSBAND ($v0^{(k)}$, $h$, $w-k$)

**Direct approach**. The direct approach prunes unreasonable pattern prefixes by enforcing the following rule in the revised HUSBAND* and WIFE* procedures: the length of each segment, except for the last one, must be greater than or equal to 2. For the sake of simplicity, the first segment is also treated as an internal segment. The revised SEGMENT* takes care of the special case with a single "1" as the first segment. For example, in Figure 4 (b), "1100" is a normal allocation (in phase 1) and "1001" is treated as a special case (in phase 2).

SEGMENT* ($n$)
1: HUSBAND* ($\emptyset$, $n$, $n$)
2: WIFE* ("1", $n-1$, $n$)

HUSBAND* ($v$, $h$, $w$)
1: **if** $w = 0$ **then**
2: PRINT ($v1^{(h)}$)
3: **else if** $w = 1$ **then**
4: PRINT ($v1^{(h)}0$)
5: **else**
6: **for** $k \leftarrow 2$ **to** $h$ **do**
7: WIFE* ($v1^{(k)}$, $h-k$, $w$)

Procedure WIFE* is same as HUSBAND*, except that "1" and "0", $h$ and $w$ are interchanged. Note that the segment method has the flattest recursion tree. It takes even fewer steps to generate a reasonable pattern than the shifting method. For example, in order to get pattern "111000", the segment method only needs two steps: $< empty > \rightarrow$ "111(000)"; while the shifting method needs three steps: "100011" $\rightarrow$ "110001" $\rightarrow$ "111000".

# 7 Performance Evaluation

**Complexity analysis**. The complexity of a recursive algorithm is mainly determined by the size of its recursion tree. We use $T(\mathcal{A})$ to denote the number of nodes and $L(\mathcal{A})$ the number of leaf nodes in the recursion tree of algorithm $\mathcal{A}$; that is, the recursive function is invoked $T(\mathcal{A})$ times satisfying the normal branch and $L(\mathcal{A})$ times satisfying the terminating condition.

**Lemma 1** *For any algorithm $\mathcal{A}$ proposed in the paper, $T(\mathcal{A}) \leq 2nL(\mathcal{A})$, where $n$ is the number of couples in the seating problem.*

Let $V(n)$ denote the number of valid patterns with $n$ couples and $R(n)$ denote the number of reasonable patterns, the following theorem holds:

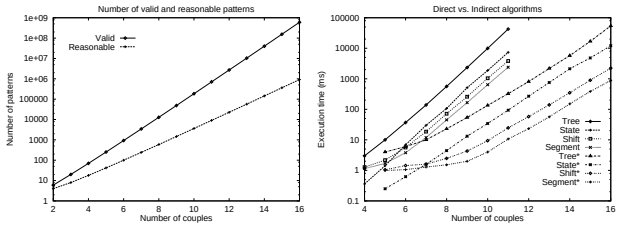
**Theorem 4** *If $\mathcal{I}$ is an indirect algorithm and $\mathcal{D}$ is the corresponding direct algorithm, then $T(\mathcal{I}) \in [V(n)/2, nV(n)]$ and $T(\mathcal{D}) \in [R(n)/2, nR(n)]$.*

**Proof**: For the shifting method, every node in the recursion tree is a valid (reasonable) pattern in the indirect (direct) approach. Because only half of the total valid (reasonable) patterns are generated, $T(\mathcal{I}) = V(n)/2$ and $T(\mathcal{D}) = R(n)/2$. For the other methods, every leaf node is a valid (reasonable) pattern in the indirect (direct) approach. From Lemma 1, $V(n)/2 \leq T(\mathcal{I}) \leq nV(n)$ and $R(n)/2 \leq T(\mathcal{D}) \leq nR(n)$. □

Apparently, $V(n) = \binom{2n}{n}$, which increases exponentially. There is no simple expression for $R(n)$. Figure 5 (a) compares $V(n)$ and $R(n)$ for $n$ from 2 to 16. We observe that (1) both $V(n)$ and $R(n)$ exhibit exponential growth, and (2) the relative ratio between $V(n)$ and $R(n)$ also increases exponentially. If these two rules still hold for larger $n$, from Theorem 4, $T(\mathcal{I})$ of any indirect algorithm $\mathcal{I}$ is larger than $T(\mathcal{D})$ of any direct algorithm $\mathcal{D}$, and the ratio $\frac{T(\mathcal{I})}{T(\mathcal{D})}$ increases exponentially with $n$. The relationship between different direct algorithms is specified in Theorem 5.

**Lemma 2** *For a recursion tree of algorithm $\mathcal{A}$, $T(n) \leq 2L(n)-1$ if every non-leaf node has at least 2 children; $T(\mathcal{A}) \geq 2L(n)-1$ if every non-leaf node has at most 2 children.*

**Theorem 5** *Let* TREE*, STATE*, SHIFT*, and* SEGMENT* *denote the direct binary tree algorithm, the direct state machine algorithm, the direct shifting algorithm, and the direct segment algorithm, respectively, then $T$(TREE*) $\geq T$(STATE*) $\geq T$(SEGMENT*) $\geq T$(SHIFT*).*




(a) (b)

Figure 5: Comparison between (a) $V(n)$ and $R(n)$, and (b) the execution time of both indirect and direct algorithms.

Table 1: Execution time (ms) of indirect algorithms.

| Couple# | TREE | STATE | SHIFT | SEGMENT |
|---|---|---|---|---|
| 6 | 37 | 6.88 | 5.60 | 3.79 |
| 7 | 139 | 30.0 | 18.6 | 11.9 |
| 8 | 562 | 107 | 71.8 | 45.1 |
| 9 | 2347 | 512 | 259 | 166 |
| 10 | 9933 | 1871 | 1042 | 642 |
| 11 | 42578 | 7366 | 3830 | 2395 |

**Proof**: Obviously, $T(\text{TREE*}) \geq T(\text{STATE*})$. $T(\text{STATE*}) \geq T(\text{SEGMENT*})$ because the recursion tree of STATE* is an incomplete binary tree, and that of SEGMENT* has no single-thread segment (that is, each non-leaf node has at least 2 children). From Lemma 2, $T(\text{STATE*}) \geq 2L(n) - 1 \geq T(\text{SEGMENT*})$. At last, $T(\text{SEGMENT*}) \geq L(n) = T(\text{SHIFT*})$. □

**Theorem 6** *For the seating problem with $n$ couples, the number of reasonable patterns $R(n) \geq 2^n$.*

**Proof**: $R(1) = 2 \geq 2^1$. Suppose $R(k) \geq 2^k$, by putting an extra couple before the first segment we get $R(n)$ reasonable patterns for $k+1$ couples; by putting the extra couple after the first segment we get other $R(n)$ reasonable patterns, and there is no intersection between these two groups. That is, $R(k+1) \geq 2R(k) \geq 2^{k+1}$, for all $k \geq 1$. □

**Simulation results**. The four indirect algorithms (TREE, STATE, SHIFT, and SEGMENT) and four direct algorithms (TREE*, STATE*, SHIFT*, and SEGMENT*) are implemented and tested in a SUN 4 workstation to compare their actual performances. Table 1 shows the execution time of the four indirect algorithms. Table 2 shows the execution time of the four direct algorithms. As expected, the execution time of both indirect and direct algorithms grows exponentially as the number of couples increases. The binary tree method fails for 12 couples for the indirect solution and 17 couples for the direct solution, when its enormous memory demands exceed the system limit.

Between different methods, the binary tree is by far the slowest. For the seating problem with 16 couples, the direct binary tree algorithm is 50 times slower than the direct segment algorithm. This is because of its largest recursion tree and the extra overhead of explicitly building a binary tree. The state machine method is about 5 times faster than the binary tree method and 10 times slower than the segment method. It is surprising that the shifting method is about two times slower than the segment method, although it has a smaller recursion tree. This can be explained by an optimization in the segment algorithm. In the other three methods, patterns are stored in the form of 0-1 strings. However, in the segment methods, patterns are stored as a sequence of integer numbers, each number representing a segment. These condensed patterns occupy less memory and take less time to generate.

Table 2: Execution time (ms) of direct algorithms.

| Couple# | TREE* | STATE* | SHIFT* | SEGMENT* |
|---|---|---|---|---|
| 6 | 6 | 0.62 | 1.44 | 1.06 |
| 8 | 23 | 4.45 | 2.47 | 1.52 |
| 10 | 135 | 34.3 | 9.40 | 3.99 |
| 12 | 810 | 269 | 57.9 | 23.3 |
| 14 | 5744 | 2128 | 349 | 151 |
| 16 | 54390 | 12078 | 2239 | 885 |

Figure 5 (b) compares the performance of all eight algorithms specified in this paper. In spite of the huge difference in absolute values, the execution time of four indirect algorithms has the same slope with a logarithmically scaled $Y$-axis. Similarly, the four direct algorithms also have almost the same slope. That means, these methods probably have the same complexity. The only difference is the constant coefficients determined by the different pruning and collapsing methods and various implementation issues. Another observation is that the execution time of both the indirect and direct algorithms grows exponentially. Although the direct algorithms are slightly better than the indirect algorithms, sooner or later, they reach the same limit of the CPU speed. For all these algorithms, the seating problem is only solvable for a few ($\leq 30$) couples.

## 8 Conclusion

In this paper we have introduced a seating problem and its four possible solutions, which are based on quite different ideas and implemented with different data structures and algorithms. For each solution, we also have compared the indirect approach that generates all possible seating patterns and eliminates the "bad" patterns, and the direct approach that only generates the "good" patterns. When describing these solutions, we provide an insight that the difference among them is the result of several transformations, namely pruning, collapsing, and flattening, on their corresponding recursion trees. Furthermore, by complexity analysis and performance simulation, we show that none of these solutions is a polynomial-time algorithm. An efficient algorithm can delay, but not prevent, the explosion of the computation time.

## References

[1] K. P. Bogart and P. G. Doyle. Non-sexist solution of the ménage problem. http://math.dartmouth.edu/~doyle/docs/menage/menage/menage.html, Sep 1994.

[2] J. Wu. Lecture Notes: COT6401 Analysis of Algorithms. http://www.cse.fau.edu/~jie/project.pdf, Fall 2001.