Chapter 4

Searching for Optimal Coalition Structure

There are a number of challenges in any NP-Hard problem. The first one is the size of search space is large and an exhaustive search algorithm can never guarantee (near) optimality in reasonable time. The second one is the terrain of the search space is complicated, i.e., there are numerous portions of the search space that seem to lead the algorithm to optimality but it is hard to choose. Furthermore, the algorithm can be misled easily to search on portions that do not lead to optimality.

In this section, we propose an A* algorithm to compute optimal coalition structure. This algorithm is an improvement of [27]. While [27] branches and bounds its search at the lower nodes which slows its termination times in some environments, this work seeks to improve this setback by applying the breadth-first search strategy (although the time to reach optimality of the algorithm has always been good.) For the sake of completion, we briefly review the algorithm here again. Each agent will execute the algorithm distributedly and exchange information about each other progress when the new solution is found. An agent will terminate the execution when it knows that the best possible answer in its search space will never be more than the present solution. There will be a solution at any point in time across the whole system. The optimal solution will be achieved when all agents terminate execution.

Note that at this point, each agent has its own search space to deal with. The input data identifying the agents' search space is in the form of cardinalities from which coalition structures are to be generated.

4.1 Data Structure

Firstly, we will discuss about the data structure used in the algorithm, which will be carried out by each agent individually.

4.1.1 Storing Coalitions and Values

Since searching for optimal coalition structure is an NP-Hard problem, the number of coalition structure grows exponentially with the number of agents. As previously discussed, crisply separating coalitions among agents would be problematic to mutual-exclusively cover the whole search space. We consider the attempt to do so is not worth it because each agent still has to store most coalitions. More importantly, the environment and the objective of the search is different as we have already discussed. We then assign all the coalitions and their values to be stored in each agent. During the generation process, the coalition values need to be accessed rapidly for each coalition structure. It is common in the literature (and in practice) that the coalition values are stored in memory during the execution of the algorithm. Using other types of storage, e.g., database, do not seem to provide instantaneous access to coalition values. Therefore, we need an ordered list, denoted by $\mathcal{C}$, to store coalitions for each cardinality. The coalition S of cardinality $|S|$ is denoted by $\mathcal{C}_{|S|,S}$. An easy way to store coalitions and values that allows for instantaneous access is to use two dimensional array as in [27]. Another option is to use linked list. Regardless of the data structure, the coalitions are to be sorted by their values anyway. For the sake of execution speed, representing coalition (members) and its value at the binary level can provide rapid operation for the validation of mutual exclusive condition of the coalition structures as well as saving memory space.

4.1.2 Mutual Exclusive Validator

According to the mutual exclusive condition of coalition structure, we also need a data structure to check for the remaining agents yet to be placed (as part of a coalition) in the coalition structure. This operation must be executed every time a new coalition is placed in the coalition structure template. The set of remaining agents is denoted by $\mathcal{R}$. Once there is no agents left in this data structure, it is guaranteed that the new coalition structure is generated. For the sake of execution speed, representing the remaining at binary level provide rapid operation for the mutual exclusive validation.

4.1.3 Coalition Structure Generation Tracer

The most complicated part of coalition structure generation is tracing how the process is progressed. Since the problem is combinatorial, it is very complicated to trace and guarantee that the algorithm is complete, i.e., all the coalition structures are generated, and, more importantly, systematic, i.e., each coalition structure is generated once and only once. The algorithm needs to keep track of each level of the coalition structure what coalition in each cardinality it has reached and use as the candidate of that cardinality, and what is the next candidate coalition of the cardinality once its candidate is the best one and is placed in the coalition structure template. For a set of n agents, we need to keep track of candidates in n cardinalities for n level. Therefore, we need a $n \times n$ matrix to keep track of the coalition structure generation. We denote this matrix by $\mathcal{T}$. The elements of the matrix are pointers to the candidate coalitions in $\mathcal{C}$. The candidate coalition of cardinality $|S|$ at level l is denoted by $\mathcal{T}_{|S|,l}$ (whose value a is the pointer to coalition $\mathcal{C}_{|S|,a}$.

4.1.4 Coalition Structure Template

The coalition structure being generated will kept in a template, which stores the best candidate coalitions found so far during the generation. This template is denoted by $\mathcal{CS}$. We denote the l-th coalition of this template by $\mathcal{CS}_l$, where $1 \leq l \leq n$.

The sample data of 6 agents are shown in figure 4.1.4.

4.2 Supportive Function

In this section, we will review the underpinning algorithms, derived from the previous work in value-oriented algorithm for optimal coalition structure search. These algorithm include the algorithm for extending new nodes to the tree, altering existing nodes in the tree, shrinking nodes in the tree, choosing the best candidate at each level and locating the next candidate in each cardinality.

4.2.1 Pre-Process Preparation

Before the real execution of the algorithm for generating coalition structures, there are a few steps for preparing the data. It is very important that all

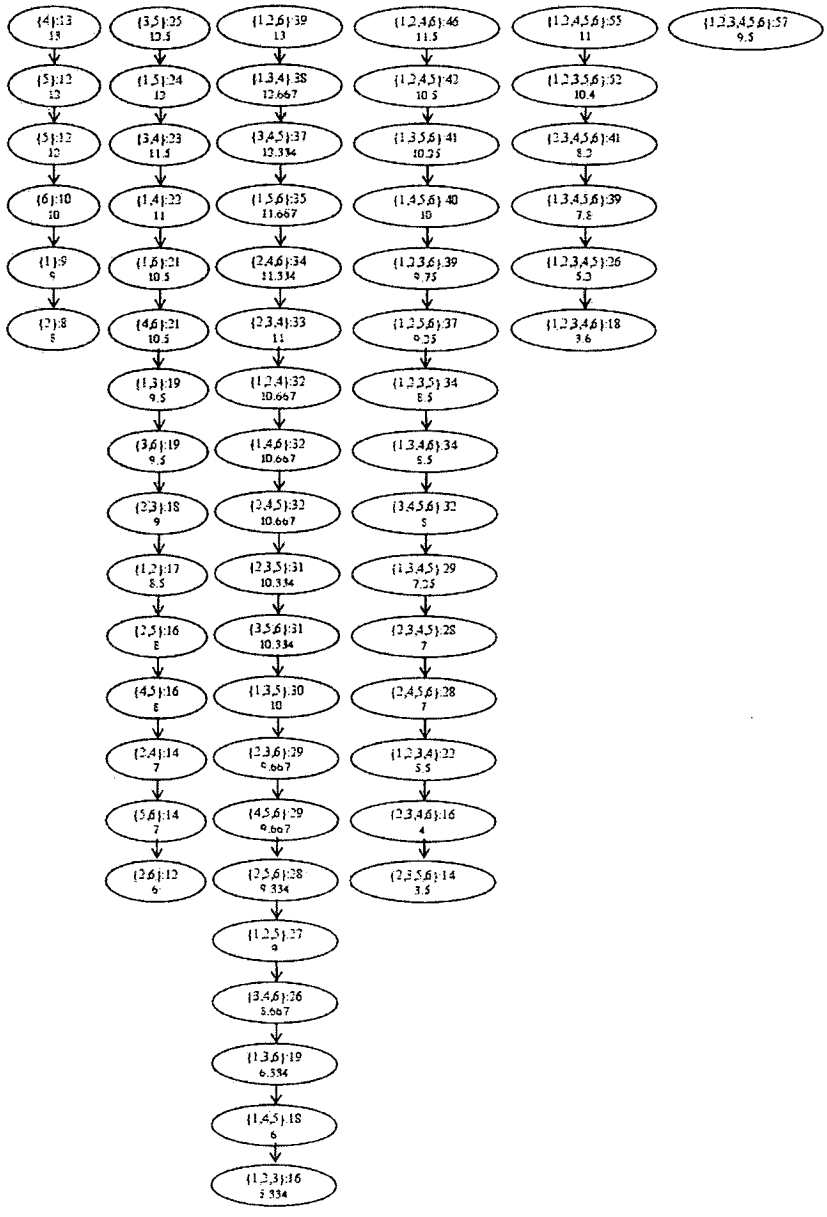

Figure 4.1: This table presents the coalitions sorted by their values in each cardinality (in the form $S : V(S)$. The lexicographical order is maintained for coalitions of the same values.

$\|S\| = 1$	$\|S\| = 2$	$\|S\| = 3$	$\|S\| = 4$	$\|S\| = 5$	$\|S\| = 6$
{1}:9	{1,2}:17	{1,2,3}:16	{1,2,3,4}:22	{1,2,3,4,5}:26	{1,2,3,4,5,6}:57
{2}:8	{1,3}:19	{1,2,4}:32	{1,2,3,5}:34	{1,2,3,4,6}:18	
{3}:11	{1,4}:22	{1,2,5}:27	{1,2,3,6}:39	{1,2,3,5,6}:52	
{4}:13	{1,5}:24	{1,2,6}:39	{1,2,4,5}:42	{1,2,4,5,6}:55	
{5}:12	{1,6}:21	{1,3,4}:38	{1,2,4,6}:46	{1,3,4,5,6}:39	
{6}:10	{2,3}:18	{1,3,5}:30	{1,2,5,6}:37	{2,3,4,5,6}:41	
	{2,4}:14	{1,3,6}:19	{1,3,4,5}:29		
	{2,5}:16	{1,4,5}:18	{1,3,4,6}:34		
	{2,6}:12	{1,4,6}:32	{1,3,5,6}:41		
	{3,4}:23	{1,5,6}:35	{1,4,5,6}:40		
	{3,5}:25	{2,3,4}:33	{2,3,4,5}:28		
	{3,6}:19	{2,3,5}:31	{2,3,4,6}:16		
	{4,5}:16	{2,3,6}:29	{2,3,5,6}:14		
	{4,6}:21	{2,4,5}:32	{2,4,5,6}:28		
	{5,6}:14	{2,4,6}:34	{3,4,5,6}:32		
		{2,5,6}:28			
		{3,4,5}:37			
		{3,4,6}:26			
		{3,5,6}:31			
		{4,5,6}:29			

Table 4.1: This table presents all the coalitions and their values (in the form $S : V(S)$) ordered lexicographically in each cardinality.

coalitions in each cardinality must be sorted by their values in descending order. We do this in order to bring in some degree of monotonicity into the search space. This will lead the algorithm to reach optimality quickly. All the preparation steps are discussed as followed:

- Each agent scans (computes) the coalitions and their values in that search space, and memorizes them in $\mathcal{C}$.
- Each agent sorts the coalitions of each cardinality by their values in descending order. Any robust algorithm for sorting can be use. A merge sort algorithm is a good option because it is one of the fastest algorithms. Its time complexity is 2 $log\ n$ while the space required is $2n$.
- All the elements in $\mathcal{T}_{i,1}$ where $1 \leq i \leq n$ are set to 1, while other elements in $\mathcal{T}$ are set to 0. This is to initialize the candidate coalitions for each $\mathcal{C}_i$ to 1.
- All the element in template $\mathcal{CS}$ are set to $\emptyset$.
- The remaining agents in $\mathcal{R}$ are set to A.

Given all the coalitions in table 4.1.4, the sorted coalitions are shown in Figure 4.1.

4.2.2 Best Candidate Coalition

While other algorithms for generating optimal coalition structure are lexicographic, our algorithm works differently. Our algorithm looks for the best candidate coalition and places it in the coalition structure template until the mutual exclusive condition is satisfied. The best candidate coalition is the one with the highest *average agent contribution value*:

$$\bar{S} = \frac{V(s)}{|S|}. \tag{4.1}$$

4.2.3 Function Extend()

This function extends $\mathcal{CS}$ to the next level. The algorithm examines whether there will be a new coalition to be inserted into layer $l+1$ of, $\mathcal{CS}$. The template $\mathcal{CS}$ can be extended if the condition $l < n$ holds. Obviously, the cardinality of the candidates for this must not be larger than the number of the remaining agents, i.e., $1 \leq i \leq |\mathcal{R}|$. Another condition is that there

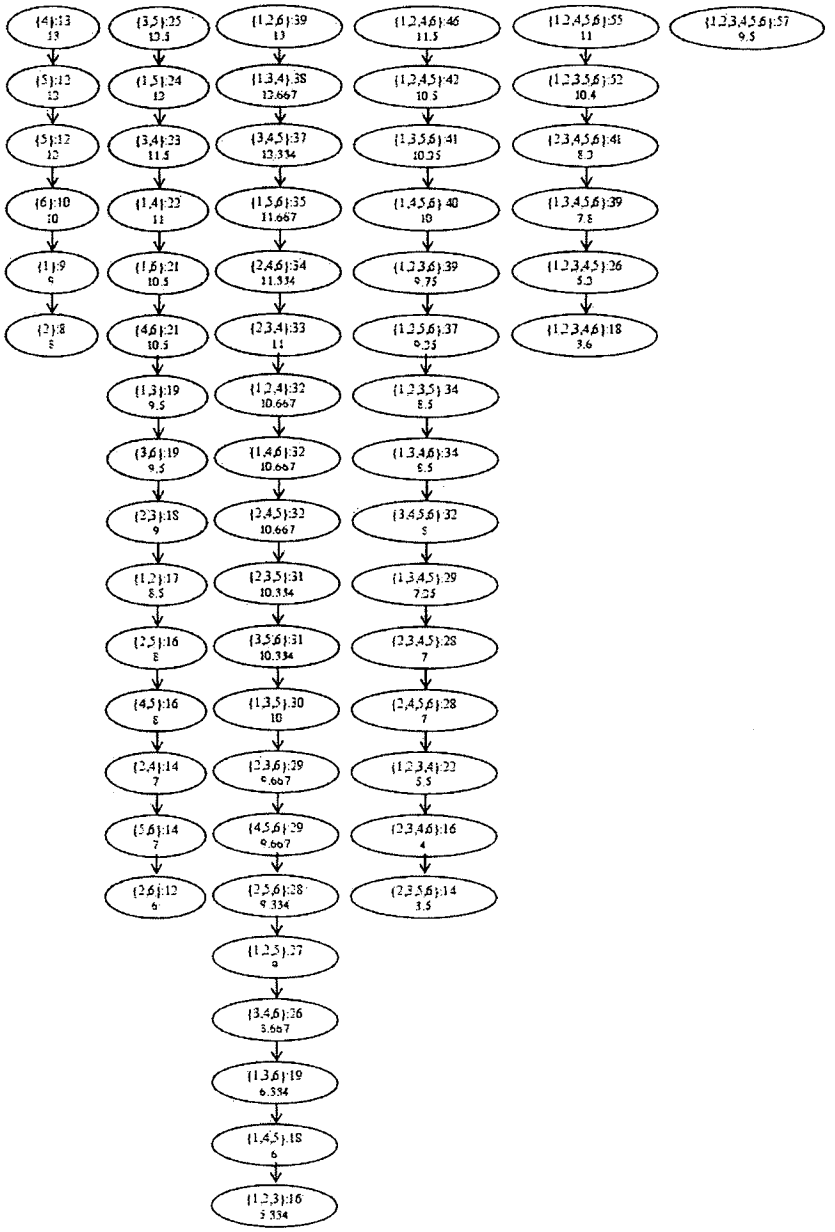

Figure 4.1: This table presents the coalitions sorted by their values in each cardinality (in the form $S : V(S)$. The lexicographical order is maintained for coalitions of the same values.

must be candidate coalitions for the next level. The candidate coalition of each cardinality must be set at tracer $\mathcal{T}_i$, where $1 \leq i \leq \mathcal{R}$. Before the algorithm finds for the next candidates, it sets the beginning position for each $\mathcal{C}_i$ to the position of the present position of current candidates, except for the cardinality of $|\mathcal{CS}_l|$, which begins at the next position. The algorithm calls function $NextS$ to locate the next candidate for each valid cardinality and updates the position in $\mathcal{T}_{i,l+1}$. At this point, the candidate coalitions at level $l+1$ are identified. The algorithm calls function $BestS$ to locate the best candidate coalition and assigns it to $\mathcal{S}^*$. The algorithm then verify if the returned value is not $\emptyset$. If the condition holds, the algorithm increases the value of l by 1, which means that $\mathcal{CS}$ can be extended to the next level. The algorithm will reach the end of the main loop and go back to the beginning of the loop. If the condition does not holds, then the algorithm tries the next option.

4.2.4 Function Alter()

In [27], this function alters the last coalition at $\mathcal{CS}_l$. This seems to be a set back of the algorithm because it takes place at the lowest level. In this work, the alteration takes place at the pivot position, starting from the head of the portion. The coalition at the pivot position will be altered with its successor, which is the next best available candidate. It discards the element $\mathcal{CS}_l$ and return its members to $\mathcal{R}$. The beginning position for finding the next candidate is updated. The algorithm locates the next candidate for $\mathcal{C}_{|\mathcal{CS}_l|}$ by calling function $NextS$. The returned value is assigned to $\mathcal{T}_{|\mathcal{CS}_l|,l}$. The algorithm calls function $BestS$ to locate the best candidate coalition $\mathcal{S}^*$. If the returned value is not null, the algorithm reaches the end of the loop and goes back to the beginning of the loop. If the returned value is null, the algorithm tries with the last option.

4.2.5 Function BestS()

At this point, there are a number of candidate coalitions pointered by elements in $\mathcal{T}_l$. Choosing the best candidate coalition is very simple. The algorithm scans for a coalition with highest $\hat{S}$ over all cardinalities $1 \leq i \leq$. Firstly, the best coalition S^* is set to empty as well as its $\hat{S}^*$ is set to 0. The algorithm then goes through each candidate coalition in descending order by their cardinalities and compares its $\hat{S}$ against $\hat{S}^*$. Only if $\hat{S} > \hat{S}^*$ then the candidate coalition is set to the best coalition. Therefore, $\mathcal{S}^*$ is also the largest among ones with highest $\hat{S}$.

4.2.6 Function NextS()

At any layer l, the algorithm needs to prepare the candidate coalition from $\mathcal{C}_i$ in each cardinality $1 \leq i \leq |\mathcal{R}|$. The process of finding the next candidate is simple but very important. Starting from the given position, the algorithm looks down the list $\mathcal{C}_i$ for the first coalition $\mathcal{S} \subseteq \mathcal{R}$. If such a coalition is found, the algorithm returns its position. Otherwise it returns 0. However, exploring every branch (adjacent nodes) can lead to the poor performance of the algorithm. In order to increase the speed of the search, unnecessary search space must be eliminated. This can be achieved by applying branch and bound.

Branch and Bound

Branch and bound is a well known technique in computer science to reduce execution time by ignoring some search space which is useless, i.e. we will never find a better solution in that portion of the whole search space. Here, we use a branch and bound mechanism in order to increase the performance of the algorithm, i.e. the algorithm can converge to the optimality quickly.

At the first level, we always have $\bar{CS} \geq \bar{CS}^*$. When proceeding, the algorithm keeps adding the best coalition to $\mathcal{CS}$. Let $\mathcal{CS}_B$, i.e. $\mathcal{CS}[l]$, be a coalition structure template at a point in time. Its value, $V(\mathcal{CS}_B)$, so far is the sum of the values of candidate coalitions chosen up to the point. Its average agent contribution is

$$\bar{CS}.$$

Let $\mathcal{S}$ be the next candidate of the given cardinality. When added to $\mathcal{CS}$, there are 2 possible consequences:

- if $\bar{S} = \bar{CS}$, this leaves $V(\mathcal{CS})$ unchanged.
- if $\bar{S} < \bar{CS}$, this definitely reduces the $\bar{CS}$.

The question here is how much the reduction can be while leaving a chance to improve $\mathcal{CS}^*$? In other words, what is the minimum value for $V(\mathcal{S})$ such that the condition $\bar{CS} > \bar{CS}^*$ holds.

Let $\mathcal{S}$ be the best candidate, whose average agent contribution is $\bar{S}^*$. The minimal value that will keep the opportunity for improving the solution has to satisfy the following condition:

$$\frac{\bar{CS} + \bar{S}}{2} > \bar{CS}^*$$

Therefore, this is the lower bound for locating the next candidate in NextS function. Implementing this is very simple. So we neglect the detailed discussion on this issue to the implementor of the algorithm.

4.3 Search Algorithms

In this work we apply the breadth-first search strategy in searching for optimal coalition structure in the main algorithm. The aforementioned functions will be used as the nuts and bolts in the main algorithm. With the parallel search adopted, the time to reach optimality is expected to be shorter.

4.3.1 Main Algorithm

Before the commence of the main algorithm, the pre-process preparation must be completed. After this step, the coalitions are sorted by their values in descending order and are ready for the search. Note that we apply branch and bound strategy for pruning the unnecessary search space and reduce the execution time. Therefore, the algorithm will generate only the coalition structure which has higher value that the latest one. We refer to the new coalition structure as the solution at that time. The last solution found before termination of the algorithm is then the optimal coalition structure. Furthermore, the agents stores all coalitions as per requested by the motivation as well as the need for internal parallel search in order to speed up the algorithm.

There are 4 main steps in the main algorithm:

- **Generate the first coalition structure.** This step is and easy task, simply keep calling function extend until the first coalition structure is generated. Within function *Extend*, the new candidate for each cardinality will be updated by calling function *Nexts*, and the best candidates can be chosen from the available ones by calling function *BestS*.

- **Locate the lower bound for the root of the tree.** The algorithm will compute the average agent contribution in the first coalition structure. This value will be used as the lower bound of the search. A direct implication of this is that the root of the optimal coalition structure will have the average agent contribution value not less this value. Further implication is that the minimum value of the average agent contribution is equal to that of the first solution, which directly implies that if there were another coalition structure, the minimum value

of average agent contribution of the root and other coalitions in that coalition structure is as much as that of the first coalition structure. The stricter implication is that if the first coalition structure is not the optimal coalition structure then the value of the average agent contribution of the first coalition of the optimal coalition structure must be more than the value of the average agent contribution. The coalition which satisfy this condition is the lower bound of the search.

- **Locate the head of the search space.** Following the steps presented in the the previous chapter, the head of the search space portions can be located. Each of these search space portions are designated by the coalitions as the root nodes of the head and the tail. The algorithm can choose a strategy to divide the search space as small (many) as appropriate. This depends on the nature of the data distribution and the preference of the agents. However, the strategy to be applied must be agreed up on by all the agents so that the search space portions are identical among agents.

- **Search for optimal coalition structures.** The algorithm will proceed the search for optimal coalition structure within the main loop. In there, the algorithm will follow the breadth-first search strategy in order to find the optimal coalition structure. Note that

4.3.2 Breadth First Search Strategy

The most important part of the algorithm is the application of the breadth first search strategy. Starting from the root node, the principle of the breadth first search is to examine all the adjacent nodes of the root at once. If the answer is found, the algorithm terminates. Otherwise, the adjacent nodes of these nodes examined. The process will be repeated until the solution/answer is found. In an implementation, these nodes will be put in a queue when it is found that they are not the result. The algorithm extract the node one by one from the queue, examine its adjacent nodes, and so on. The algorithm terminates when the solution is found or when the queue is empty. Therefore, the progress of the algorithm proceeds from one level to the next level.

In this work, the concept of breadth first search strategy has to be applied a little because in the basic breadth first search algorithm, only a single node is examined while in our work a single node is just part of the examination. Therefore, we cannot think of the root node of the graph just as a single coalition which is assigned as the head of each search space portion. Since the head of the portion is a part of the coalition structure, which has been

built up to the level where the head is located, this information has to be passed on as the root of the graph. In addition, we also need to know the remaining agents and the tracing information in order to keep track of available coalitions in each cardinality. Hence, the needed information include the present template CS, the tracer $\mathcal{T}$ and the remaining agents $\mathcal{R}$.

4.3.3 Main Loop

Since this work applies breadth-first search strategy, we need to follow the common approach used in the literature. There are two data structures used for such a purpose: graph and queue. Here, we have discussed to some extent in the previous Chapter about graph. However, there is additional detail we need to add into the node, i.e. all the relevant information to explore the graph further. This information includes the present $\mathcal{CS}, \mathcal{T}$ and $\mathcal{R}$. We shall refer to the node by $\mathcal{N}$. We denote by $\mathcal{N}.\mathcal{CS}$, $\mathcal{N}.\mathcal{T}$ and $\mathcal{N}.\mathcal{CS}$ the template, the tracer and the remaining agents of the node. Queue is a common data structure used in implementing breadth-first search. Here, we use queue to keep track of how the algorithm progresses. There are two functions being used here: *Enqueue* and *Dequeue*. *Dequeue* takes one input, which is the queue $\mathcal{Q}$ itself. This function simply remove the last element of the queue and return it to the caller. *Enqueue* takes two inputs, i.e. the queue $\mathcal{Q}$ and the new node $\mathcal{N}$. This function simply inserts $\mathcal{N}$ into $\mathcal{Q}$ as its last element.

The main function is supposed to be given a queue of nodes, each of which represents the head of a search portion. These nodes can be calculated using the algorithm presented in the previous chapter. The algorithm proceeds in a simple manner. It examines if the queue has more elements. If that is the case, it calls function *Dequeue* to retrieve the node. The relevant information, i.e. $\mathcal{CS}, \mathcal{R}, \mathcal{T}$ are extracted. The new $\mathcal{S}$ is retrieved by calling $BestS$, which determines the best candidate for appending $\mathcal{CS}$. After examining $\mathcal{S}$ is not null, the algorithm then enters a small loop to fulfill the $\mathcal{CS}$. Inside the loop, $\mathcal{S}$ is appended to $\mathcal{CS}$ and members of $\mathcal{S}$ are removed from $\mathcal{R}$. The algorithm then examines if there are agents left in $\mathcal{R}$. If that is the case, the present solution $\mathcal{CS}^*$ is set to $\mathcal{CS}$ and exit the loop. Otherwise, the algorithm still have to proceed. It locates the candidate coalitions in each cardinality by calling $Next$ and identifies the best candidate $\mathcal{S}$ by calling $BestS$.

After the loop, the algorithm have to set $\mathcal{CS}, \mathcal{R}$ and $\mathcal{T}$ to that of $\mathcal{N}$ once more because their values have been changed by the previous steps. The algorithm then calls function *Alter*, which alters the last coalition of $\mathcal{CS}$ with its valid successor, i.e. the first coalition down the list which satisfies the mutual exclusive condition. Note that members of $\mathcal{S}$ will be returned to $\mathcal{R}$ in function *Alter*. The algorithm then examines whether the alteration is

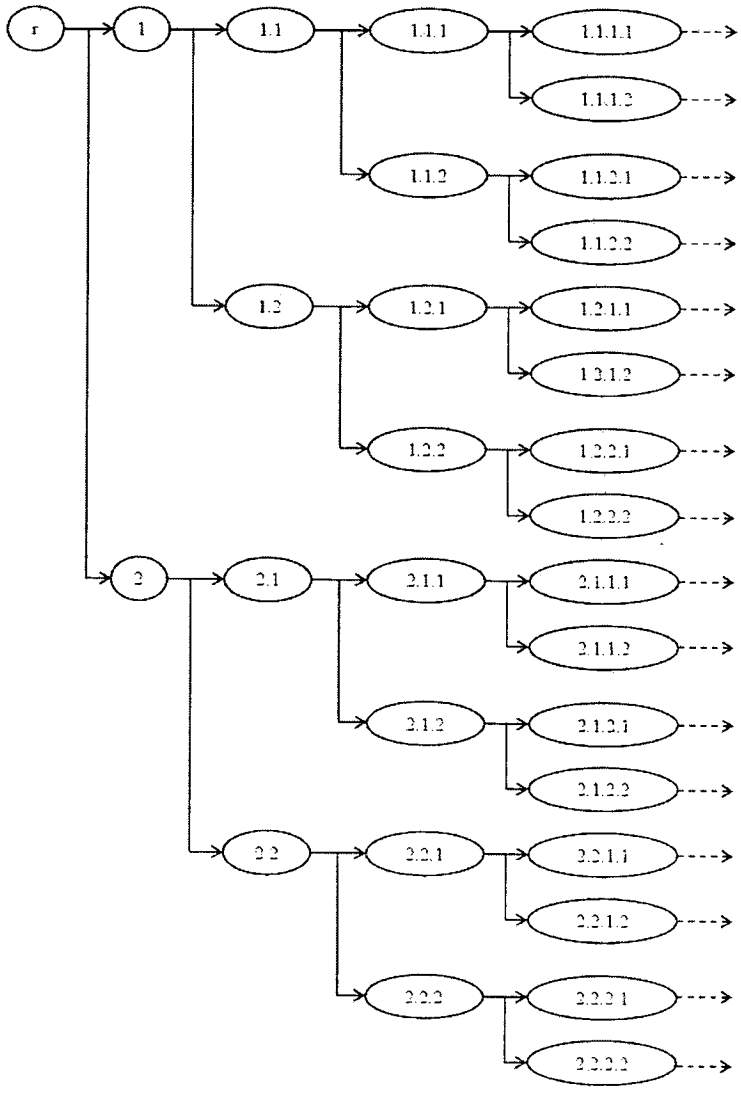

Figure 4.2: **Breadth-First Search** This Figure illustrates how the breadth-first search strategy can be applied optimal coalition structure problem.

successful, i.e. $\mathcal{S} \neq \emptyset$. If that is the case, the algorithm enters another loop in which a new node,$\mathcal{N}$, will be initialized. The template, the tracer and the validator of $\mathcal{N}$ are set to their respective values. The algorithm calls function $\mathcal{E}\backslash\amalg\sqcap]\sqcap]$ to insert $\mathcal{N}$ to its tail. The last step of the loop is trying to alter $\mathcal{CS}$ again.

Algorithm 1 The main function repeatedly examines the element of the queue whether it can lead to the new solution. There are two steps in the main loop: to fullfil $\mathcal{CS}$ and to alter $\mathcal{CS}$.

Require: $\mathcal{Q}$
1: **while** $\mathcal{Q}! = \emptyset$ **do** ▷ While queue is not empty
2: $\mathcal{N} \leftarrow Dequeue(\mathcal{Q})$
3: $\mathcal{CS} \leftarrow \mathcal{N}.\mathcal{CS}$
4: $\mathcal{T} \leftarrow \mathcal{N}.\mathcal{T}$
5: $\mathcal{R} \leftarrow \mathcal{N}.\mathcal{R}$
6: $\mathcal{S} \leftarrow BestS(\mathcal{T}, \mathcal{R})$
7: **while** $\mathcal{S} \neq \emptyset$ **do**
8: $\mathcal{CS}.last \leftarrow \mathcal{S}$
9: $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{S}$
10: **if** $\mathcal{R} \neq \emptyset$ **then**
11: $\mathcal{CS}^* \leftarrow \mathcal{CS}$
12: **end if**
13: $NextS(\mathcal{S}, \mathcal{T}, \mathcal{R})$
14: $\mathcal{S} \leftarrow BestS(\mathcal{T}, \mathcal{R})$
15: **end while**
16: $\mathcal{CS} \leftarrow \mathcal{N}.\mathcal{CS}$
17: $\mathcal{T} \leftarrow \mathcal{N}.\mathcal{T}$
18: $\mathcal{R} \leftarrow \mathcal{N}.\mathcal{R}$
19: $\mathcal{S} \leftarrow Alter(\mathcal{CS}, \mathcal{T}, \mathcal{R})$
20: **while** $\mathcal{S} \neq \emptyset$ **do**
21: $\mathcal{N} \leftarrow Initialize()$
22: $\mathcal{N}.\mathcal{S} \leftarrow \mathcal{S}$
23: $\mathcal{N}.\mathcal{R} \leftarrow \mathcal{R}$
24: $\mathcal{N}.\mathcal{T} \leftarrow \mathcal{T}$
25: $Enqueue(\mathcal{Q}, \mathcal{N})$
26: $\mathcal{S} \leftarrow Alter(\mathcal{CS}, \mathcal{T}, \mathcal{R})$
27: **end while**
28: **end while**

The example run of the algorithm is shown in Figure 4.3.

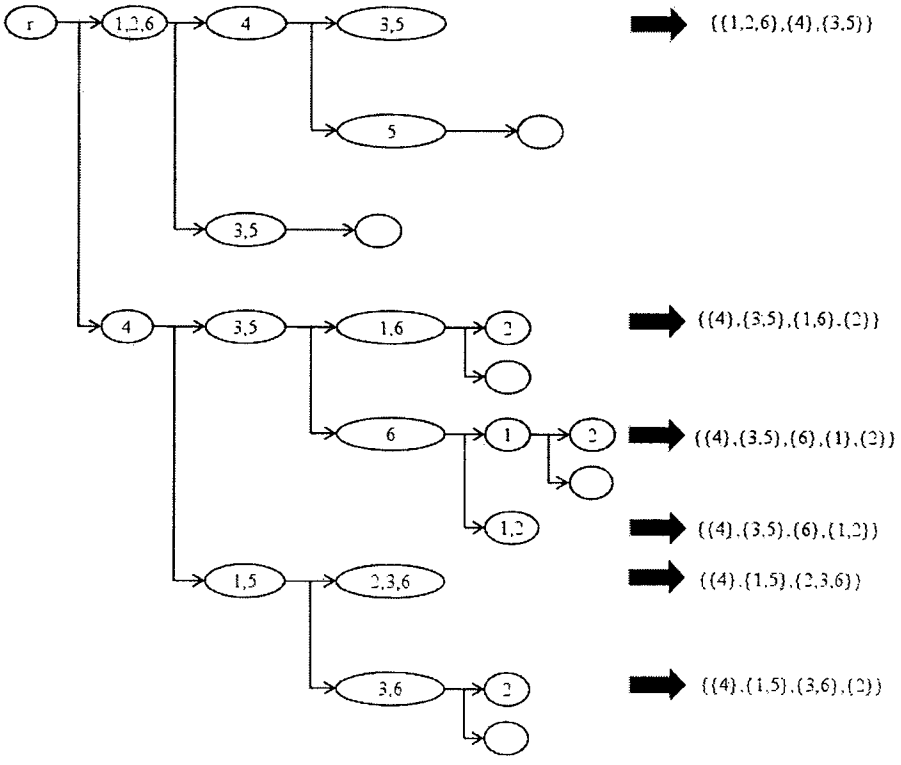

Figure 4.3: **Example Run** This figure shows how the algorithm generates coalition structures.