International Journal of Emerging Engineering Research and Technology
Volume 2, Issue 4, July 2014, PP 383-391
ISSN 2349-4395 (Print) & ISSN 2349-4409 (Online)

Application of Parallel Algorithm for Knight Tour Problem to Avoid Backtracking

S. Vijayakumar
MCA Department
Priyadarshini Engineering College, Vaniyambadi
Tamilnadu, India

Dr. Vidyaathulasiraman
Department of Computer Science
Government Arts & Science College (W), Bargur
Tamilnadu, India

Abstract: *Parallel processing issues become engrained into a variety of applications. This paper concentrates on exploiting parallelism at machine level and data level for processing in AI systems. A review had been conducted and the proposed algorithm is applied over the Knight Tour problem specified under several scenarios. The result obtained from the scenarios prove that the proposed technique is advantageous over the existing technique as there is a significant level of reduction in the machine response time, and also it eliminates the need for extra memory, as it avoids backtracking. The proposed algorithm when applied over an enhanced machine whose architecture is based on machine level parallelism results in an AI system, whose response time would still be reduced.*

Keywords: *Predicate calculus, Unification, Parallel Algorithms, Production System, Pattern_Search Algorithm, Recognize-act cycle, Conflict set.*

1. Introduction

1.1 Parallel Processing

In computers, parallel processing is the processing of program instructions by dividing them among multiple processors with the objective of running a program in less time.

1.2 Production System

The production system is a model of computation that has proved particularly important in AI, both for implementing search algorithms and for modeling human problem solving. A production system provides pattern-directed control of a problem-solving process and consists of a set of production rules, a working memory, and a recognize-act control cycle.

1.3 Knight Tour Problem

The use of predicate calculus with a general controller to solve problems is illustrated through an example: a reduced version of the *knight's tour problem.* In the game of chess, a knight can move two squares either horizontally or vertically followed by one square in an orthogonal direction as long as it does not move off the board. There are thus at most eight possible moves that the knight may make.

As traditionally defined, the knight's tour problem attempt to find a series of legal moves in which the knight lands on each square of the chessboard exactly once. This problem has been a mainstay in the development and presentation of search algorithms. The example we use in this paper is a simplified version of the knight's tour problem. It asks whether there is a series of legal moves that will take the knight from one square to another on a reduced-size (3x3) chessboard.

A (3x3) chessboard with each square labeled with integers 1to 9 is shown in Figure 1. This labeling scheme is used instead of the more general approach of giving each space a row and column number in order to further simplify the example.

In view of the reduced size of the problem, we simply enumerate the alternative moves rather than developing a general move operator. The legal moves on the board are than described in predicate calculus using a predicate called **move**, whose parameters are the starting and ending squares of a legal move.

For example, move (1, 8) takes the knight from the upper left-hand corner to the middle of the bottom row. The predicates of figure 1 enumerate all possible moves for the 3x3 chessboard.

1	**move(1,8)**	**9**	**move(6,1)**
2	**move(1,6)**	**10**	**move(6,7)**
3	**move(2,9)**	**11**	**move(7,2)**
4	**move(2,7)**	**12**	**move(7,6)**
5	**move(3,4)**	**13**	**move(8,3)**
6	**move(3,8)**	**14**	**move(8,1)**
7	**move(4,9)**	**15**	**move(9,2)**
8	**move(4,3)**	**16**	**move(9,4)**

1	**2**	**3**
4	**5**	**6**
7	**8**	**9**

Fig1. *A 3X3 chessboard with move rules for the simplified Knight Tour Problem*

These predicates form the knowledge base for the knight tour problem. As an example of how unification is used to access this knowledge base , we test for the existence of various moves on the board . To determine whether there is a move from 1 to 8, call **pattern_search(move(1,8))**. Since this goal unifies with **move(1,8)** in the knowledge base, the result is **success**, with no variable substitutions required.

Another request might be to find where the knight can move from a particular location, such as square **2** . The goal **move(2,X)** unifies with two different predicates in the knowledge base, with the substitutions of **{7/X}** and **{9/X}**. Given the goal **move(2,3)** exists in the knowledge base . The goal query **move(5,Y)** also fails because no assertions exist that define a move from square 5.

The next task in constructing a search algorithm is to devise a general definition for a path of successive moves around the board, this is done through the use of predicate calculus implications, These are added to the knowledge base as *rules* for creating paths of successive moves . To emphasize the goal-directed use of these rules, we have reversed the direction of the implication arrow,i.e., the rules are written as **conclusion← premise**

For example, a two –move path could be formulated as:

$$\forall\ X, Y\ [\text{path2}(X, Y) \leftarrow \exists\ Z\ [\text{move}(X, Y)] \wedge \text{move}\ (Z,Y)]]$$

This rule says that for all locations X and Y, a two –move path exists between them if there exists a location **Z** such that the knight can move from **X** to **Z** and then move from **Z** to **Y**.

The general **path2** rule can be applied in a number of ways. First, it may be used to determine whether there is a two-move path from one location to another. If **pattern-search** is called with the goal **path2 (1, 3)**, it matches the goal with the consequence of the rule **path2(X,Y)**, and the substitutions are made in the rule's premise; the result is a specific rule that defines the conditions required for the path:

$$\text{Path}\ (2(1, 3) \leftarrow \exists\ Z\ [\text{move}\ (1, Z)\ \wedge \text{move}\ (Z, 3)]$$

Pattern_search then calls itself on this premise. Since this is a conjunction of two expressions, pattern_search will attempt to solve each subgoal separately. This requires not only that both subgoals succeed but also that any variable bindings be consistent across subgoals, substituting 8 to Z allows both subgoal to succeed.

Another request might be to find all locations that can be reached in two moves from location **2**. This is accomplished by giving **pattern_search** the goal **path2(2,Y)**.Through a similar process, a number of such substitutions may be found, including **{6/Y}** and **{2/Y}** (with intermediate **Z** being **7**) and **{2/Y}** and **{4/Y}** (with intermediate location **9**). Further requests could be to find a two-move path from a number to itself, from any number to **5**, and so on. We notice here .once of the advantages of pattern__ driven control: a variety of queries may be taken s the initial goal.

Similarly, a three –move path is defined as including two intermediate locations that are part of the path from initial to the goal. This is defined by:

$$\forall\ X, Y\ [\text{path3}(x,y) \leftarrow \exists\ Z,W[\text{move}(X,Z) \wedge \text{move}\ (Z,W) \wedge \text{move}(W,Y)]]$$

This clause can solve such goals as **path3 (1,2)**, **path 3(1,X)**,or even **path3(X,Y)**.

It soon becomes evident that the path moves are the same for a path of any length, simply requiring the proper number of intermediate places to "land". It is also evident that the path moves could be stated in terms of each other, such as:

$$. \forall X,Y([path3(X,Z) \leftarrow \exists Z [move(X,Z) \wedge path2(Z,Y)]]$$

This suggests the single, general recursive rule:

$$\forall X,Y [path(X,Y) \leftarrow \exists Z [move(X,Z) \wedge path (Z,Y)]]$$

The last path can be used to determine whether a path of any length exists. The rule may be stated as "to find a path from one square to another, first make a move from the starting square to an intermediate location and then find a path from the intermediate to the final square."

This recursive "path" rule is incomplete in that it includes no terminating condition. Any attempt to solve a goal involving the **path** predicate would fail to halt because each attempt to solve the premise would lead to another recursive call on **path (Z, Y)**. There is no test in the rule to determine whether the desire goal state is ever reached. This can be remedied by adding the clause path(X,X)to the knowledge base. Since **path (X, X)** will unify only with predicates such as path (3,3) or path (5,5), it defines the desired terminating condition. The general recursive path definition is then given by two predicate calculus formulas:

$$\forall X \; path (X, X)$$

$$\forall X,Y \leftarrow \exists [path(X,Y) [move(X,Z) \wedge path(Z,Y)]]$$

We observe again the elegance and simplicity of the recursive formulation. When combined with the recursive control provided by **pattern _search**, these rules will search the space of possible paths in the knight's tour problem. Combined with the move rules, this yields the complete problem description (or knowledge base)

move(1,8)	**move(1,6)**	**move(2,9)**	**move(2,7)**
move(3,4)	**move(3,8)**	**move(4,9)**	**move(4,3)**
move(6,1)	**move(6,7)**	**move(7,2)**	**move(7,6)**
move(8,3)	**move(8,1)**	**move(9,2)**	**move(9,4)**

$\forall X \; path (X,X)$

$\forall X, Y [path(X,Y) \leftarrow \exists [move(X,Y) \wedge path (Z,Y)]]$

It is important to note that the solution to the problem is implemented through both the logical descriptions that define the state space and the use of **pattern_search** to control search of that space. Although the **path** rule is a satisfactory definition of a path, it does not tell us how to find that path. Indeed, many undesirable or meaningless paths around the chessboard also fit this definition .For example, without some way to prevent loops, the goal **path (1, 3)** could lead to a path from **1**to **8** to **3**, both the loop and the correct path are logical consequences of the knowledge base. Similarly, if the recursive rule is tried before the terminating condition, the fact that **path (3, 3)** should terminate the search could be overlooked, allowing the search to continue meaninglessly.

2. EXISTING PRODUCTION SYSTEM EXECUTION WITH SINGLE PROCESSOR

The 3x3 knight's tour problem presented in section 1.3 may be solved using a production system approach. Here each move would be represented as a rule whose condition is the location of the knight on a particular square and whose action moves the knight to another square. Sixteen productions represent all possible moves of the knight.

Working memory contains both the board state and goal state. The control regime applies rules until the current state equals the goal state and then halts. A simple conflict resolution scheme would fire the first rule that did not cause the search to loop.

Since the search may lead to dead ends (from which every possible move leads to a previously visited state and, consequently, a loop), the control regime should also allow backtracking; an execution of this production system that determines whether a path exists from square square 1 to 2 is shown in Table 1.

RULE	**CONDITION**		**ACTION**
1	knight on square 1	→	move knight to square 8
2	knight on square 1	→	move knight to square 6
3	knight on square 2	→	move knight to square 9
4	knight on square 2	→	move knight to square 7
5	knight on square 3	→	move knight to square 4
6	knight on square 3	→	move knight to square 8
7	knight on square 4	→	move knight to square 9
8	knight on square 4	→	move knight to square 3
9	knight on square 6	→	move knight to square 1
10	knight on square 6	→	move knight to square 7
11	knight on square 7	→	move knight to square 2
12	knight on square 7	→	move knight to square 6
13	knight on square 8	→	move knight to square 3
14	knight on square 8	→	move knight to square 1
15	knight on square 9	→	move knight to square 2
16	knight on square 9	→	move knight to square 4

Fig2. *A 3X3 chessboard with Possible moves for the simplified Knight Tour Problem.*

Table1. *A path from square 1 to square 2*

Iteration #	Working memory Current square	 Goal square	Conflict Set	Fire rule
0	1	2	1,2	1
1	8	2	13,14	13
2	3	2	5,6	5
3	4	2	7,8	7
4	9	2	15,16	15
5	2	2		Halt

It is interesting to note that in implementing that **path** predicate in the knight's tour example of section 1.3 , we have actually implemented this production system solution. From this point of view, **pattern_search** is simply an interpreter, with the actual search implemented by the **path** definition. The production are the **move** facts, with the first parameter specifying the condition (the square the piece must be on to make the move)and the second parameter, the action (the square to which it can move). The recognize-act cycle is implemented by the recursive **path** predicate. Working memory contains the current state and the desired goal state and is represented as the parameters of the **path** predicate. On a given iteration, the conflict set is all of the move expressions that will unify with the goal **move (x,z).** This program uses the simple conflict resolution strategy of selecting and firing the first move predicate encountered in the knowledge base that does not lead to a repeated state. The controller also backtracks from dead-end state. This characterization of the **path** definition as a production system is given in figure 2.

Production systems are capable of generating infinite loops when searching a state space graph. These loops are particularly difficult to spot in a production system because the rule can fire in any order. That is, looping may appear in the execution of the system, but it cannot easily be found from a syntactic inspection of the rule set. For example , with the "move" rule of the knight's tour problem ordered as in section 5.2 and a conflict resolution strategy of selecting the first match, the pattern **move(2,X)** would match with **move(2,9),** indication a move to square 9. On the next iteration, the pattern **move(9,X)** would match with **move(9,2),** taking the search back to square **2**, causing a loop.

To prevent looping, **pattern_search** checks a global list (**closed)** of visited state. The actual conflict resolution strategy was therefore: select the first matching move that leads to *unvisited state.*

In a production system, the proper place for recording such case-specific data as a list of previously visited states is not a global closed list but the working memory itself. We can alter the **path** predicate to use working memory for loop detection.

Let us assume that **pattern_search** does not maintain a global closed list or otherwise perform loop detection. Let us assume that our predicate calculus language is augmented by the addition of a special construct, **assert(X)** , which causes its argument X to be entered into the working memory. A**ssert** is not an ordinary predicate but an action that is performed; hence, it always succeeds.

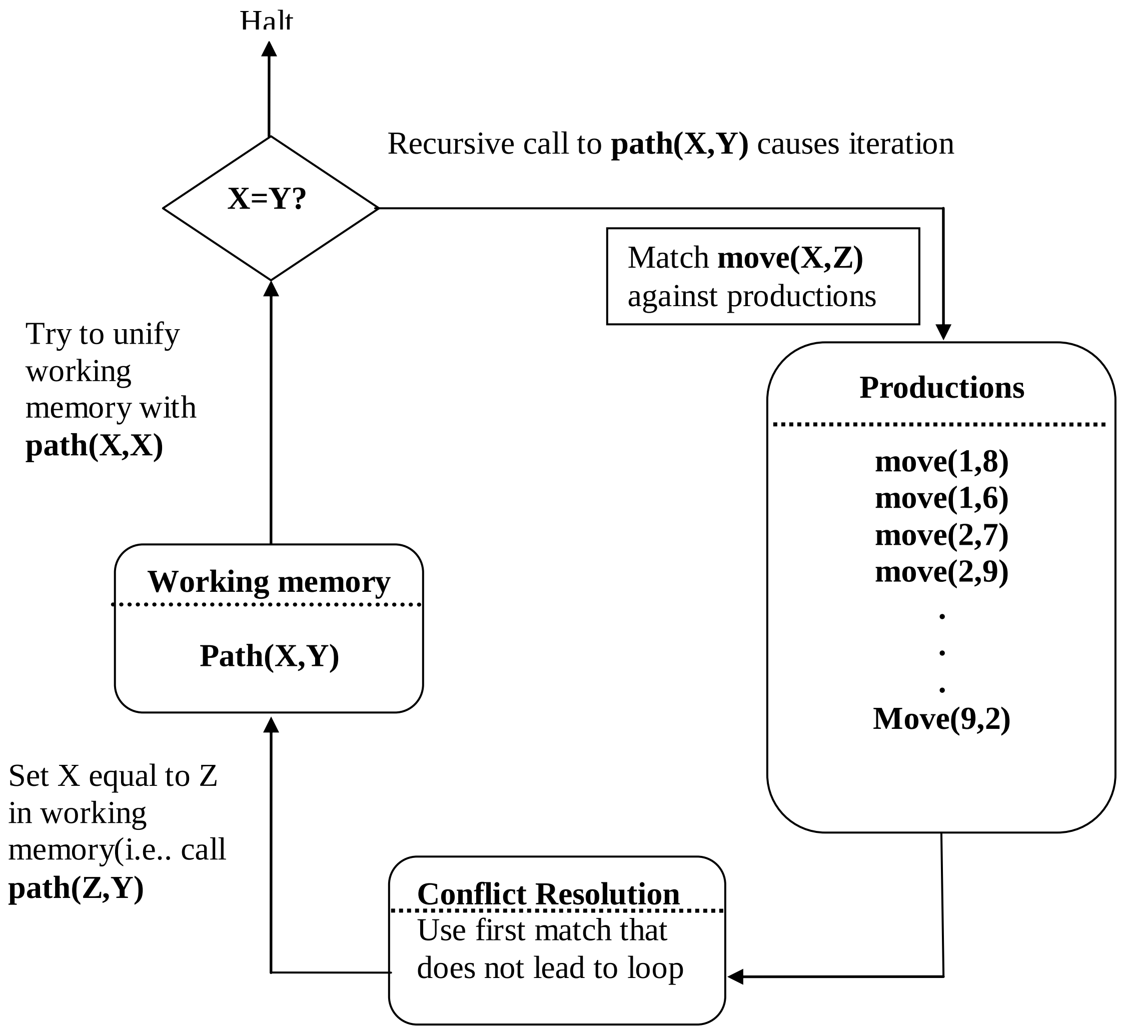
Halt
X=Y?
Recursive call to path(X,Y) causes iteration
Match move(X,Z) against productions
Try to unify working memory with path(X,X)
Productions
move(1,8)
move(1,6)
move(2,7)
move(2,9)
.
.
.
Move(9,2)
Working memory
Path(X,Y)
Set X equal to Z in working memory(i.e.. call path(Z,Y)
Conflict Resolution
Use first match that does not lead to loop

Fig3. *The recursive path algorithm: a production system*

Assert is used to place a "marker" in working memory to indicate when a state has been visited. This marker is a unary predicate, **been(X)**, which takes as its argument a square on the board **been(X)** is added to working memory when a new state X is visited. Conflict resolution may then require that **been(Z)** must not be in working memory before **move(X,Z)** can fire . For a specific value of **Z,** this can be tested by matching a pattern against working memory.

The modified recursive path definition is written:

$$\forall \textbf{ X path(X,X)}$$

$$\forall \textbf{ X path(X,Y)} \leftarrow \exists \textbf{ Z move(X,Y)} \wedge \neg\textbf{(been(z))} \wedge \textbf{assert(been(Z))} \wedge \textbf{path(Z,Y)}$$

In this definition, **move(X,Y)** succeeds on the first match with a move predicate. This binds a value to Z. If **been(Z)** matches with an entry in working memory,-(**been(Z))** will cause a failure (i.e., it will be false).

pattern_search will then backtrack and try another match for **move(X,Z)**. If square **Z** is a new state, the search will continue, with **been(Z)** asserted to the working memory to prevent future loops. The actual firing of the production takes place when the path algorithm recurs. Thus, the presence of **been** predicates in working memory implements loop detection in this production system.

We note that although predicate calculus is used as the language for both productions and working memory entries, the procedural nature of production systems requires that the goal be tested in left-to-right order in the **path** definition. Then order of interpretation is provided by **pattern_search.**

Using assert: In this method, if we want to find it a path exists from Square 7 to Square 4, iterations shown in Table 2 result.

Table2. *Loop is avoided using assert in the path from square 7 to square 4*

Iteration #	Working memory Current square	Goal square	Conflict Set	assert	Fire rule
0	7	4	11,12	11	11
1	2	4	3,4	11,3	3
2	9	4	15,16	11,3,15	15
3	2	4	4	11,3,15,4	4
4	7	4	12	11,3,15,4,12	12
5	6	4	9,10	11,3,15,4,12,9	9
6	1	4	1,2	11,3,15,4,12,9,1	1
7	8	4	13,14	11,3,15,4,12,9,1,13	13
8	3	4	5,6	11,3,15,4,12,9,1,13,5	5
9	4	4			Halt

3. PROPOSED PRODUCTION SYSTEM WITH MULTIPLE PROCESSORS

3.1. Data Parallelism

For this problem we are going to use Data Parallelism which is the use of multiple functional units to apply the same operation simultaneously to elements of a data set**.** The data set available here are the production rules. From the sixteen production rules given in the section 2, we can observe that for each square there exits exactly two possible moves.

$$\forall X \leftarrow \exists Y,Z\ [path(X,Y),\ path(X,Z)]$$

For example, from Square 1, the possible moves are **move(1,8)** and **move(1,6).** If these two moves are checked simultaneously by separate processors, one of the processor may yield the goal state. For every square, if the possible two moves are simultaneously checked, then there is no need for backtracking and two processors are more than enough for this problem.

3.2. Working Method with Two Processors

Working memory is initialized at the beginning with problem descriptions. The current state of the problem solving is maintained as a set of patterns in working memory. These patterns are matched against the goal state. If the current state is the goal state, then halt. Otherwise with recursive call to the production rules, this produces a subset of production rules called the conflict set. The conflict set always contains two production rules. One production rule will be processed by processor 1 and another processed by processor 2. If one of the production rules unify with the working memory, then halt.

An algorithm proposed for the new model is given below

```
Step 1: Start.
Step 2: Input the Current State and Goal State.
Step 3: Initialize Processor P1 with Current State.
Step 4: P1 obtains the two possible Conflict Set from the Production Rules.
Step 5: P1 fires the Rule for First Conflict Set from the Working Memory.
        P2 fires the Rule for Second Conflict Set from the Working Memory.
Step 6: If the P1 or P2 reaches the Goal State then Halt.
Step 7: Else
        Initialize P1 as per the Fire Rule in Step 5.
Step 8: Repeat Step 4 to Step 7.
```

From the above algorithm we test if a path exists from square 1 to square 2, iterations are as shown in table 3.

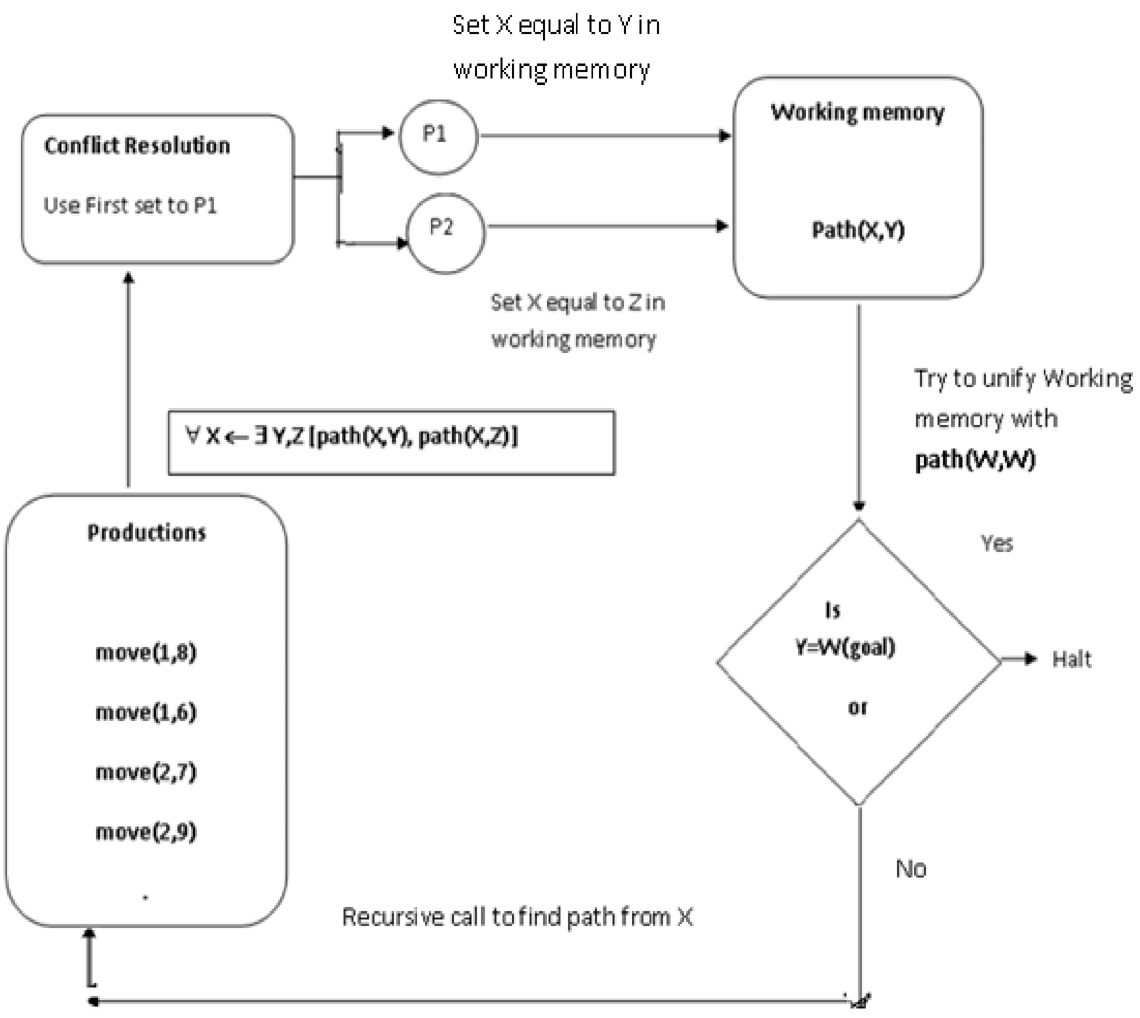
Set X equal to Y in working memory
Conflict Resolution
Use First set to P1
P1
P2
Working memory
Path(X,Y)
Set X equal to Z in working memory
Try to unify Working memory with path(W,W)
∀ X ← ∃ Y,Z [path(X,Y), path(X,Z)]
Productions
move(1,8)
move(1,6)
move(2,7)
move(2,9)
Yes
Is Y=W(goal) or
Halt
No
Recursive call to find path from X

Fig4. *The recursive path algorithm: a production system with two processors*

Table3. *A path from square 1 to square 2 by two processor method*

Iteration #	Working Memory			Conflict set from P1	Fire Rule	
	Current State P1	P2	Goal State		P1 First	P2 Second
0	1	-	2	1,2	1	2
1	8	6	2	13,14	13	14
2	3	1	2	5,6	5	6
3	4	8	2	7,8	7	8
4	9	3	2	15,16	15	16
5	2	4	2	Halt		

If we compare Table 1 and Table 3 (exhibiting a scenario, where the Current State is 1 and the GOAL state is 2), both the algorithms reaches the goal state in five iterations.

An another Scenario, where if we want to find Whether a path exists from square 7 to square 4 , iterations are shown in Table 4.

Table4. *A path from square 7 to square 4 by two processor method*

Iteration #	Working Memory			Conflict set from P1	Fire Rule	
	Current State P1	P2	Goal State		P1 First	P2 Second
0	7	-	4	11,12	11	12
1	2	6	4	3,4	3	4
2	9	7	4	15,16	15	16
3	2	4	4	Halt		

According to the above specified example, we are able to see a drastic reduction in the number of iterations, when we compare Table 2 and Table 4, as in Single processor method, it consumes 9 iterations to reach the GOAL state, whereas in Two processor method, it consumes only 3 iterations to reach the GOAL state.

Note: The single processor method consumes extra memory for storing assert, as in Table 2, which totally avoided in two processor method, i.e., in Table 4. As a result, the two processor method does not require any extra memory.

Hence our proposed algorithm finds the path from square 7 to squar 4 in three iterations without any backtracking which is not possible in the existing algorithm.

Another Scenario is also specified in Appendix A, where the Current state is 6 and the GOAL state is 7. Here we are able to see a drastic reduction in the number of iterations, as in Single processor method, it consumes 13 iterations to reach the GOAL state, whereas in Two processor method, it consumes only 1 iterations to reach the GOAL state.

Converse path is also possible by our proposed algorithm. Suppose we want to find if a pat exists from square 4 to square 7 which is converse to table 4. The iterations are shown in Table 5.

Table5. *A path from square 4 to square 7 by two processor method*

Iteration #	Working Memory			Conflict set from P1	Fire Rule	
	Current State		Goal State			
	P1	P2			P1 First	P2 Second
0	4	-	7	7,8	7	8
1	9	3	7	15,16	15	16
2	2	4	7	3,4	3	4
3	9	7	7	Halt		

Hence, in this reduced 3X3 knight tour problem, we can find the path from any state as a current state and reach the goal state without any backtracking. If we use the backtracking, a single processor algorithm will test only one path of the current state and the goal state may be reached by many iterations. To reduce considerable amount of iterations, we use one more processor, so that both the paths of the state are checked by the two processors simultaneously. If the number of iterations are reduced, then definitely total execution time is reduced.

4. Conclusions

The result obtained from the scenarios prove that the proposed technique is advantageous over the existing technique as there is a significant level of reduction in the machine response time, and also it eliminates the need for extra memory, as it avoids backtracking. In this reduced 3X3 knight tour problem, we are having only 16 production rules. If we increase the rows and columns such as 8X8, !6X16, etc then the production rules also increases. Searching with single processor from huge number of production rules will take much time. We can manage this problem by multiple processors and parallel algorithms.

References

[1]Kai Hwang and Douglas, Degroot,. Parallel processing for super computers and artificial intelligence, Mc Graw Hills, 1989.

[2]George F, Artificial intelligence structures and strategies for complex problem solving, IV Edition , Pearson Education 2007.

[3]L.Zhuo and V.K.Prasanna, scalable and Modular Algorithms for Floating point Matrix Multiplication on Reconfigurable computing systems, IEEE Trans .on parallel and Distributed systems, Vol 18, pp 433-448, April 2007.

[4]S.Dhagl, M.M.Hayat J.E.Peoza C.Yong and D.A.Bader, “Dynamic Load Balancing in Distributed systems in the presence of Delays :,IEEE Trans .on parallel and Distributed systems , Vol 18, pp 485-497, Apr2007.

[5]K.Li, “Analysis of parallel Algorithms for Matrix chain product and Matrix power on Distributed Memory systems “,IEEE Trans .on parallel and Distributed systems Vol 18 ,pp 865-878, July 2007.

[6] S.H.Chiang and S.Vasupongayya ,"Design and potential performance of Goal-Oriented Job scheduling policies for parallel computer Worloads", IEEE Trans. On parallel and distributed systems,Vol 19, pp1642-1656, Dec 2007.

[7] Xian- He Sun and Diane T. Rover, "Scalability of Parallel Algorithms- Machine Combinations", IEEE Trans. . On parallel and distributed systems,Vol 5,No 6, pp599-613, June 1994.

[8] Norma Alias and Md. Rajibul Islam, "A Review of the Parallel Algorithms for Solving Multidimentional PDE Problems", Jounal of Applied Sciences 10(19) pp 2187-2197, 2010.

AUTHOR'S BIOGRAPHY

S.Vijaya Kumar had completed his MCA from University of Madras in the year 2001, and had completed his M.Phil in the year 2007. He had authored 1 Research paper. He is currently working as Assistant Professor (Sr) in the Department of Computer Applications, Priyadashini Engineering College, Vaniyambadi, Vellore Dt, TN. His area of interest includes Parallel Processing, Artificial Intelligence and Data Mining.

Vidyaathulasiraman had completed her MCA from University of Madras in the year 1997, and had completed her M.Phil in the year 2002 and Doctorate in the year 2010 from Mother Teresa Women's University. She had authored 16 papers and had co-authored 4 papers. She had also published two books, titled "Computer Graphics" and "Biometrics". She is currently working as Assistant Professor and Head in the Department of Computer Science, Government Arts & Science College (W), Bargur. She had guided around 45 M.Phil students and her area of interest includes Network Security Management, Artificial Intelligence and Data Mining.