Learning Guide Unit 4

Site:	University of the People	Printed by:	Ryohei Hayashi
Course:	CS 4408-01 Artificial Intelligence - AY2025-T3	Date:	Thursday, 30 January 2025, 3:43 PM
Book:	Learning Guide Unit 4		

Description

Learning Guide Unit 4

Table of contents

Overview

Unit 4: Features and Constraints

Topics:

- Variables and Constraints
- Constraint Satisfaction Problem (CSP)
- Search Optimization Techniques
    - Domain Splitting
    - Variable Elimination
    - Local Search
    - Iterative Best Improvement
    - Randomized Algorithms
    - Stochastic Local Search
    - Beam Search and Stochastic Beam Search
    - Minimax Algorithm in Game Theory

Learning Objectives:

By the end of this Unit, you will be able to:

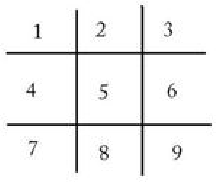
1. Examine how CSPs (constraint satisfaction problems) solve real-world problems.
2. Solve graph coloring problem using CSPs backtracking algorithm.
3. Define various search and optimization techniques appropriate to the problem.
4. Analyze how a Minimax backtracking algorithm used in decision making to find the optimal solution.

Tasks:

- Read the Learning Guide and Reading Assignments
- Participate in the Discussion Assignment (post, comment, and rate in the Discussion Forum)
- Complete and submit the Programming Assignment
- Complete an entry in the Learning Journal
- Take the Self-Quiz

Introduction

In the previous unit, we talked about structuring all of the possible moves in a tic-tac-toe game as paths in a search tree. The tic-tac-toe game board has nine positions as shown in the following diagram. As we discussed in the previous unit each of these nine positions has the potential of 3 different states as it can have a blank, an x, or an o. Nine positions each with three states is 93 or 729 states!



Tic-tac-toe is a simple game but even with such a simple game, we have 729 states. As the number of elements in the state space increases the number of potential states that must be represented increases exponentially. What should be clear is that attempting to manage every state in a complex system individually is simply not practical as the number of states increases.

Instead of attempting to manage states, we typically try to define a set of features that describes the state space. In our example above, each of the 9 locations has 3 states. Rather than keeping track of each possible state (combinations of x's, o's, and blanks), we could define variables for each feature and the range of the variable would be inclusive of the potential states that the feature could occupy.

For example, location 1 might be a feature that could have values in the range of {x,o,_} where _ is a blank. This reduced our 729 states down to 9 features that can be represented by variables. We can use algebraic variables to represent features within our model. By translating features into variables, which have a range of possible values, we can dramatically reduce the information process associated with states.

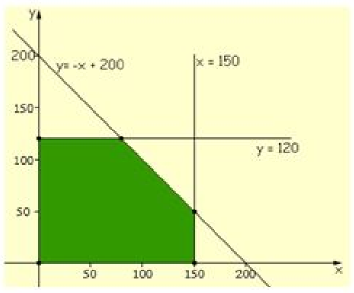
Another example of how this can reduce complexity can be seen in graphic images. A graphic image basically is a data structure where each pixel in the image has an associated data value. Imagine a greyscale picture. Each pixel can contain a discrete variable but one that has many values. In systems that use 24 bits of color depth, each pixel has the potential for 16,777,216 different values or states. Of course, if the picture is a medium-size photo with 1024 x 760 pixels it would contain 778,240 pixels each with 16,777,216 values or states which means that the state space of the picture would be approximately 13,000,000,000,000 states. Obviously a number that is too big to deal with. By treating the picture as a set of features it is far more manageable.

By identifying features and defining them with variables we have a way of reasoning about the state space of any particular problem. Of course within the set of features we often also have a series of constraints that must be met.

Constraint Satisfaction Problems

Constraints are limitations to action. We have been introduced to the concept of constraints in the past and of course the need to be able to find solutions that can accomplish goals within given constraints.

Some of you might recall the linear programming problems that you had to do in CS 3404 Analysis of Algorithms. You were given a goal such as 'maximize profits' and a set of constraints such as the fact that factory could produce some quantity of product A and some other quantity of product B and there were limited raw materials and the goal was to determine how much of each product to produce to maximize profitability. If you recall, there was a graphical view of the problem where there were only a couple of variables and constraints. You learned to use linear programming to find the solution for more complex problems.



Of course, these are approaches to solving a class of problems known as constraint satisfaction problems or CSPs. Constraints are common when developing agents that operate in a larger environment. Constraint satisfaction is an important element of agent reasoning and is often factored into the problem search trees that are generated for agent-based problems.

Generate and Test Algorithms

One approach to solving a finite CSP (a finite constraint satisfaction problem is one in which the number of potential solutions that must be searched to find a workable solution is finite and can be searched in a measurable period of time) is through an exhaustive generate-and-search algorithm.

This approach is a brute force approach. The idea is that every possible condition is generated in advance and then each solution is tested in order. When a solution is found that fits the constraints it is returned and the search for a solution ends.

This is obviously a simple approach that is easy to implement, however, it is also inefficient and the amount of time that it takes to find a solution could be excessively long.

Solving CSP through Search

An alternative to this brute force approach would be to construct a search space of potential solutions and then use the search strategies (and optimization strategies) covered in previous units to make searching for a solution more efficient.

Local Search

The basic idea around local search is different from what we have been studying up to this point.   We have been focusing on the idea of a comprehensive search to find problems. Algorithms can make the process of searching methodically for solutions more efficient, but the techniques have all embodied the idea that we have defined a set of search paths that represent all of the options. We search through all of the options using a methodical and essentially serial process until a solution is found.

In local search, we are conducting a search in a very different way. Imagine that we had a constraint optimization problem where we had four variables and a series of constraints on each variable, such as the following:

Variables A, B, C, and D where A $A >= B$, B is not = 0, $C < D$, and $B+A > C$ given

$A \in$ {set of all integers}, $B \in \{4,5,6,7,8\}$, $C \in$ {set of all integers}

This is a relatively complex constraint problem. In local search, the idea would be to assign some value to each of the variables perhaps just random numbers, and then ‘walk’ through incremental or neighboring values for each of these variable assignments until a fit is found that meets the criteria. This process is repeated for each of the constraint variables.

Local search algorithms will typically have some form of a ‘stop’ variable or value that will limit the number of iterations that will be checked for a variable.   If a solution is not found within the limited-stop window then another random ‘try’ will be executed.

One implementation of a local search is known as Hill Climbing and it basically is designed to find a solution that fits the constraints but then will check ‘neighboring’ values to see if the solution can be optimized. If a more optimal solution is found then it becomes the new candidate solution and the optimization continues.

Hill Climbing can get trapped at local maxima, to avoid this use a random search that occasionally accepts changes that decrease object function. The Hill Climbing is combined with the ability to determine the local selection of a neighbor using a randomization routine. Such a combination is known as Stochastic local search and it features random selection in two areas. First, is the initial random assignment of local variables as part of the ‘try’ and second is the selection of the neighbors to test.

Optimization

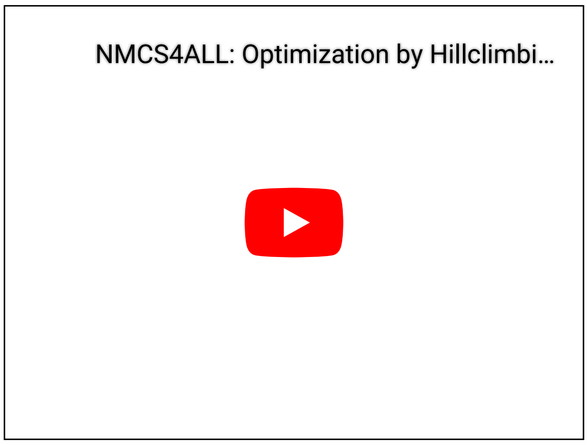
There are a number of procedures and processes that can be employed to optimize CSPs to improve processing time and to ensure that more optimum solutions have been discovered which are covered in detail throughout Chapter 4 in our text.

Some of these optimization techniques include:

- Iterative best improvement or Hill Climbing
- Stochastic local search which introduces random moves
- Simulated annealing which uses random moves and the idea of a ‘temperature’ gauge which counts down to zero. This temperature is used to slowly reduce the randomization of the algorithm which provides a balance between greedy upward moves that can get caught in local maximums and randomized moves which can allow the algorithm to ‘break out’ of local maximum traps.

Watch the following video by Dave Ackley from the University of New Mexico as he demonstrates some of these optimization techniques.

Ackley, D. (2013, March 1). NMCS4ALL: Optimization by Hillclimbing [Video]. YouTube.



Minimax Algorithm

In a two-player game, how can an agent develop a strategy when it has no way of knowing what moves its opponent will make? Technically it can't, therefore some assumptions must be made. With no prior knowledge of its opponent's skill or knowledge, it is reasonable to assume its opponent will play optimally. If an agent can determine what the optimal move is given the current state, it can predict what moves its opponent will make assuming its opponent plays optimally. This is an important assumption that must be kept in mind as we talk about algorithms for finding strategies.

Assuming both players play optimally, it is possible to compute what is known as minimax values for all nodes in the game tree and use these values to determine the optimal strategy. The minimax value for a node is defined as follows:

The minimax value of a node is the utility for MAX to be in the corresponding state, assuming both players play optimally from that state to the end of the game.

Based on this definition, MAX will always try to move to the state with the highest value, while MIN will always try to move to the state with the lowest minimax value. For the terminal nodes, the minimax value is simply the value of the utility function. The minimax values for the nodes above the terminal functions are always computed from the bottom up using the following rules:

1. If the node is a MIN node (i.e., it is MIN's turn), the minimax value will be the minimum of the minimax values of the node's successors.

2. If the node is a MAX node (i.e., it is MAX's turn), the minimax value will be the maximum of the minimax values of the node's successors.

Check out the following link to get a better idea of how the Minimax algorithm is used in a tic-tac-toe game.

Surma, G. (2018, November 12). Tic tac toe – Creating unbeatable AI. *Towards datascience*. https://towardsdatascience.com/tic-tac-toe-creating-unbeatable-ai-with-minimax-algorithm-8af9e52c1e7d

References

Russell, S. & Norvig, P. (1995). *Artificial intelligence: A modern approach*. Pearson Education, Inc.

Surma, G. (2018, November 12). Tic tac toe – Creating unbeatable AI. *Towards datascience*. https://towardsdatascience.com/tic-tac-toe-creating-unbeatable-ai-with-minimax-algorithm-8af9e52c1e7d

Reading Assignment

Poole, D. L., & Mackworth, A. K. (2017). *Artificial Intelligence: Foundations of computational agents*. Cambridge University Press. https://artint.info/2e/html/ArtInt2e.html

Read the following chapters:

- Chapter 4 – Reasoning with Constraints

Aradhya, A. L. (n.d.). Minimax algorithm in game theory | Set 1 (Introduction). *GeeksforGeeks*. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

Video Resources

This video is helpful for completing the programming assignment.

Iacobelli, F. (2015, June 29). *Constraint satisfaction problems* [Video]. YouTube.

Discussion Assignment

For your discussion post, explain your process for each of the following steps:

1. Define constraint satisfaction problem (CSP) in terms of variables, domains, and constraints.

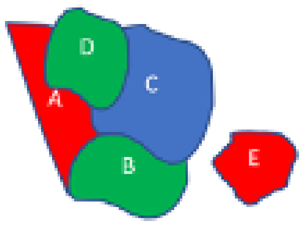
2. Then consider the N-Queen Problem (N number of queens should be placed in an N X N matrix such that no queen shares the same row, column, or diagonal). Convert this problem to CSP by creating a variable set, domain set, and constraint set.

3. Construct a search space in the form of a graph from which the search strategy can be used from the above information. Feel free to upload any hand diagram or graph.

4. Finally identify the search strategy/algorithm that you would recommend to solve this problem and explain why you selected the strategy?

Your Discussion should be at least 250 words in length, but not more than 750 words. Use APA citations and references for the textbook and any other sources used.

Programming Assignment



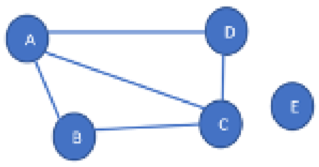


Fig 1. Map of a country with 5 districts Fig 2. Graph representation of Fig 1 map

Consider the famous map coloring problem. Your goal is to find the minimum number of colors and the color name for the entire map such that two adjacent regions of the map are not colored with the same color. For example, in Fig 1, we need a minimum of 3 colors (red, green, blue).

For this programming assignment, you need to write a program in Java or Python using the Constraint Satisfaction Problem (CSP) algorithm. Here are the detailed requirements.

1. Your program must be complete, compile able, and produce output.
2. Your program must be generic enough to solve problems with other country maps.
3. Your program should be able to handle a minimum of 2 and a maximum of 20 vertices/districts.
   2 =< V >= 20, where V is total number of vertices.
4. The total number of colors you can use is 7. Red, Green, Blue, Yellow, Violet, Gray, Orange. Remember, your goal is to choose the minimum number of colors. For example, given in Fig 1, you only need 3 colors to satisfy the goal (no adjacent regions has the same color)
5. Your program will read the graph from a text file (example: graph.txt)
6. Follow the order of the color mentioned in number 4. For example, if you need 4th color, pick Yellow, if you need 5th color, then pick Violet and so on. This would be easier for you when you are assessing your peer's assignment by comparing it with your output.
7. You must follow the CSP backtracking algorithm.
8. If your program can't solve the problem using 7 colors, it should output "Not Possible with 7 colors."

Sample Input (graph.txt) (Based on Fig 1)

```
5
A B C D
B A C
C A B D
D A C
E
```

Input file Explanation:

The first line represents the total number of vertices/nodes/districts. In this case, the total number is 5.

After that, there will be the same number of lines representing the vertex name and their adjacent vertices. Each vertex and it's adjacent must be separated by space. For example, A B C D means vertex A has 3 adjacent vertices which are B, C, and D. The last line represents the vertex E with no adjacent.

Sample Output (Based on Fig 1)

```
A red
B green
C blue
D green
E red
```

Output Explanation:

Your output should mention each vertex name in a separate line and the appropriate color name separated by one space.

If a given map is not possible to color with the given 7 colors, simple output the following line:

Not possible with 7 colors

Sample coding on how to read the input file (graph.txt)

```
// assuming graph.txt file located in the same project folder
// if not then add the entire path
File file = new File("graph.txt");

BufferedReader br = new BufferedReader(new FileReader(file));

int total_no_vertices = Integer.parseInt(br.readLine());

for (int i = 1; i <= total_no_vertices; ++i) {
   String line = br.readLine();
   String[] vertex_and_adjacent = line.split(" ");

   for (int j = 0; j < vertex_and_adjacent.length; ++j) {
     // Assign the vertex vertex_and_adjacent[0] into your data structure
      // Assign it's adjacent vertices from vertex_and_adjacent[1] to vertex_and_adjacent[length - 1] into another data structure
   }
 }
```

You will be graded on the following:

1. Does the program compile?
2. Does the program correctly read the input file?
3. Does the data structure correctly represent the map/graph?
4. Is the Constraint Satisfaction Problem (CSP) backtracking algorithm correctly implemented?
5. Does the program produce the correct output?
6. Do you think given a scenario where 7 colors are not enough to solve the problem, the program will give the corresponding output?

Learning Journal

The Learning Journal is a tool for self-reflection on the learning process. The Learning Journal will be assessed by your instructor as part of your Final Grade.

Your learning journal entry must be a reflective statement that considers the following questions:

1. Describe what you did. This does not mean that you copy and paste from what you have posted or the assignments you have prepared. You need to describe what you did and how you did it.

2. Describe your reactions to what you did.

3. Describe any feedback you received or any specific interactions you had while participating discussion forum or the development assignment, discuss how they were helpful.

4. Describe your feelings and attitudes.

5. Describe what you learned. You can think of one or more topics from your week's lesson and explain your understanding in writings. Feel free to add any diagram or coding example if that helps you explain better.

6. Did you face any challenges while doing the discussion assignment or the development assignment? Were you able to solve it by yourself?

The Learning Journal entry should be a minimum of 400 words and not more than 750 words. Use APA citations and references if you use ideas from the readings or other sources.

Self-Quiz

The Self-Quiz gives you an opportunity to self-assess your knowledge of what you have learned so far.

The results of the Self-Quiz do not count towards your final grade, but the quiz is an important part of the University's learning process and it is expected that you will take it to ensure understanding of the materials presented. Reviewing and analyzing your results will help you perform better on future Graded Quizzes and the Final Exam.

Please access the Self-Quiz on the main course homepage; it will be listed inside the Unit.

Checklist

- Peer assess Unit 3 Development Assignment
- Read the Learning Guide and Reading Assignments
- Participate in the Discussion Assignment (post, comment, and rate in the Discussion Forum)
- Complete and submit the Programming Assignment
- Complete an entry in the Learning Journal
- Take the Self-Quiz