MCA Part III

Paper: 21 (Artificial Intelligence)

Topic: STATE SPACE REPRESENTATION AND SEARCH TECHNIQUES

Prepared by: Dr. Kiran Pandey

(School of Computer Science)

Email-Id: kiranpandey.nou@gmail.com

INTRODUCTION

Searching is the universal technique of problem solving in AI. There are some single-player games such as tile games, Sudoku, crossword, etc. The search algorithms help you to search for a particular position in such games. We will also discuss the concept of problem reduction and constraint satisfaction in this unit. This will give an insight in various searching techniques and use of AI in doing so.

TYPES OF SEARCHES

There are two types of search, based on whether they use information about the goal.

(i) Uninformed search (ii) Informed Search

Un-informed Search

This type of search does not use any domain knowledge. This means that it does not use any information that helps it reach the goal, like closeness or location of the goal. The strategies or algorithms, using this form of search, ignore where they are going until they find a goal and report success.

Uninformed search, also called blind search, is a class of general purpose search algorithms that operate in **a brute-force way**. These algorithms can be applied to a variety of search problems, but since they don't take into account the target problem. They are most simple, as they do not need any domain-specific knowledge. They work fine with small number of possible states.

Requirements of the search:

- State description
- A set of valid operators
- Initial state
- Goal state description

The basic uninformed search strategies are:

- **BFS (Breadth First Search):** It expands the shallowest node (node having lowest depth) first.
- **DFS (Depth First Search**): It expands deepest node first.
- **DLS (Depth Limited Search)**: It is DFS with a limit on depth.
- **IDS(Iterative Deepening Search)**: It is DFS with increasing limit
- **UCS (Uniform Cost Search):** It expands the node with least cost (Cost for expanding the node).
- **Bidirectional Search.**

Breadth-First Search (BFS)

It is a simple strategy in which the root node is expanded first, then all the successor nodes are expanded next, then their successors and so on. Thus all the nodes are expanded at the given depth first in the search tree before going to the next level of node. It generates one tree at a time until the solution is found. This method uses First-In-First-Out (FIFO) queue assuring that the nodes that are visited first will be expanded first.

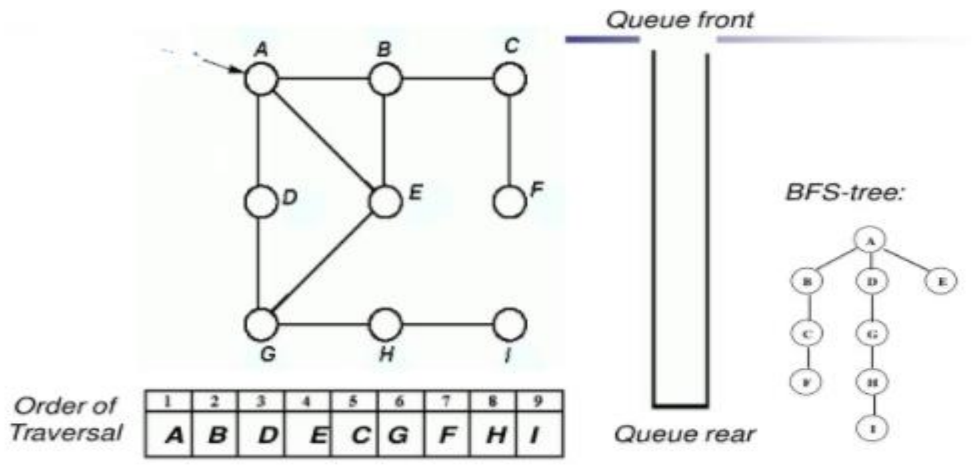


Figure 1: Breath First Search using a queue.

Advantage: Simple to implement and provides shortest path to the solution.

Disadvantage: Memory requirement is high. Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential. For example, if your problem has a solution of depth 12, then it will take 35 years for breadth first search to find it.

If **branching factor** (average number of child nodes for a given node) = b and depth = d, then number of nodes at level d = b^d. The total no of nodes created in worst case is $b + b^2 + b^3 + \ldots + b^d$. Its complexity depends on the number of nodes. It can check duplicate nodes.

Depth-First Search (DFS)

It always expands the deepest node in the current-fringe of the search tree. It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as Breadth-First method, only in the different order.

As the nodes on the single path are stored in each iteration from root to leaf node, the space requirement to store nodes is linear. With branching factor ***b*** and depth as ***m***, the storage space is ***bm****.*

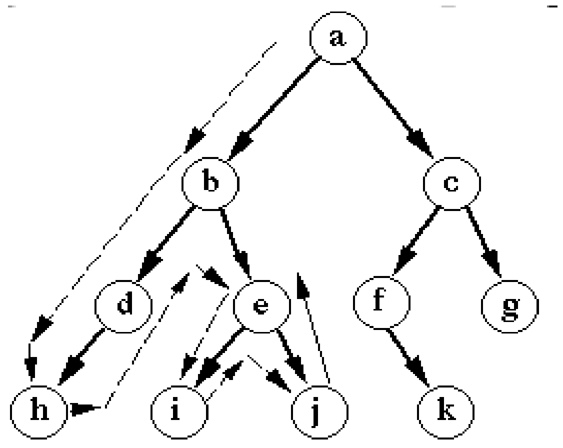


Figure 2: Depth first Search

Advantages: DFS search has a very modest memory requirements. It needs only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

Disadvantage − This algorithm may not terminate and go on infinitely on one path. The solution to this issue is to choose a cut-off depth. If the ideal cut-off is **d**, and if chosen cut-off is lesser than **d**, then this algorithm may fail. If chosen cut-off is more than **d**, then execution time increases. Its complexity depends on the number of paths. It cannot check duplicate nodes.

Depth Limited Search (DLS)

It is DFS with a limit on depth. That is at depth ***l*** nodes are treated as if they have no successors. It solves the infinite-path problem. DFS can be viewed as a special case of depth-limited search with $l = \infty$.

DLs also introduces an additional source of incompleteness if we choose $l<d$, that is, the shallowest goal is beyond the depth limit. DLS will also be non-optimal if we choose $l>d$. Its time complexity is is **$O(b^l)$ and space complexity is $O(bl)$.**

Iterative Deepening Search (IDS)

It is a general strategy often used in combination with DFS that finds the best depth limit. It is DFS with increasing limit. It combines the benefits of DFS and BFS. Like DFS its memory requirements are very modest that is **$O(b^d)$** but like BFS it is complete when the branching factor is finite. It is optimal when the path cost is a non-decreasing function of the depth of the node.

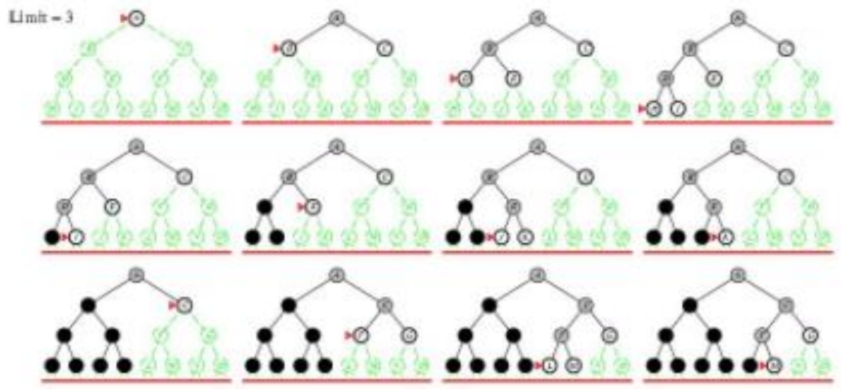


Figure 3: Iterative Deepening Search

It performs depth-first search to level 1, starts over, executes a complete depth-first search to level 2, and continues in such way till the solution is found. It never creates a node until all lower nodes are generated. It only saves a stack of nodes. The algorithm ends when it finds a solution at depth *d*. The number of nodes created at depth *d* is b^d and at depth *d-1* is $b^{d-1.}$

Uniform Cost Search (UCS)

It expands the node with least cost. It is identical to BFS. UCS does not care about the number of steps a path has, but only about their total cost. Therefore it will be stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state. We can guarantee completeness provided the cost of every step is greater than or equal to some small positive constant c. This condition is sufficient to ensure optimality. It means that the cost of the path always increases as we go along the path. UCS is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of **b** and **d**.

Disadvantage − There can be multiple long paths with the cost ≤ C*. Uniform Cost search must explore them all.

Bidirectional Search

It searches forward from initial state and backward from goal state till both meet to identify a common state. The path from initial state is concatenated with the inverse path from the goal state. Each search is done only up to half of the total path. The motivation is that $\mathbf{b^{d/2} + b^{d/2}}$ is much less than $\mathbf{b^d}$. The time and space complexity of bidirectional search is $\mathbf{b^{d/2}}$**.** The algorithm is complete and optimal if both searches are BFS; other combinations may sacrifice completeness, optimality or both.

Comparison of Various Algorithms Complexities

Let us see the performance of algorithms based on various criteria –

Criterion	Breadth First	Depth First	Bidirectional	Uniform Cost	Interactive Deepening
Time	b^d	b^m	$b^{d/2}$	b^d	b^d
Space	b^d	b^m	$b^{d/2}$	b^d	b^d
Optimality	Yes	No	Yes	Yes	Yes
Completeness	Yes	No	Yes	Yes	Yes

Informed (Heuristic) Search

It uses problem-specific knowledge beyond the definition of the problem itself and can find the solution more efficiently than an uninformed strategy. This type of search uses domain knowledge. It generally uses a heuristic function that estimates how close a state is to the goal. This heuristic need not be perfect. This function is used to estimate the cost from a state to the closest goal. To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these number of moves for all tiles. A key component of these algorithms is a heuristic function denoted by h(n);

h(n) = estimated cost of the cheapest path from node to a goal node.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes. In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

The basic informed search strategies are:

- **Greedy search (best first search)** : It expands the node that appears to be closest to goal
- **A* search**: Minimize the total estimated solution cost that includes cost of reaching a state and cost of reaching goal from that state.

Greedy Best First Search

This algorithm tries to expand the node that is closest to the goal on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: f(n)= h(n). It is implemented using priority queue. Expand the node with smallest h and is similar to depth-first search. It follows single path all the way to goal, backs up when dead end.

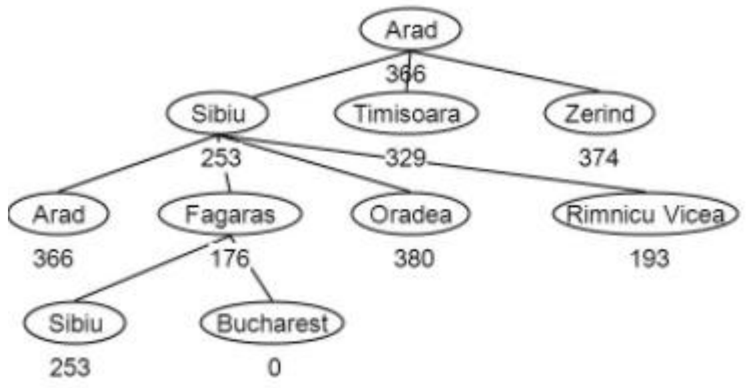


Figure 4: Greedy Best First Search

Worst case time: O(bm), m = depth of search space

Worst case memory: O(bm), needs to store all nodes in memory to see which one to expand next. It can get stuck in loops. It is not optimal.

Branch and Bound Search

It is applied to problems having graph search space where more than one alternative path exist between two nodes. This search saves all the paths (costs) from a node to all generated nodes and chooses the shortest path for further expansion. It then compares the new path lengths with all old ones and again chooses the shortest path for expansion. In this manner, any path to a goal node is certain to be minimal length path. It uses queue data structure. Always extending the lowest cost path in branch and bound search insures that a lowest cost path will be found if one exists. An example is given below:

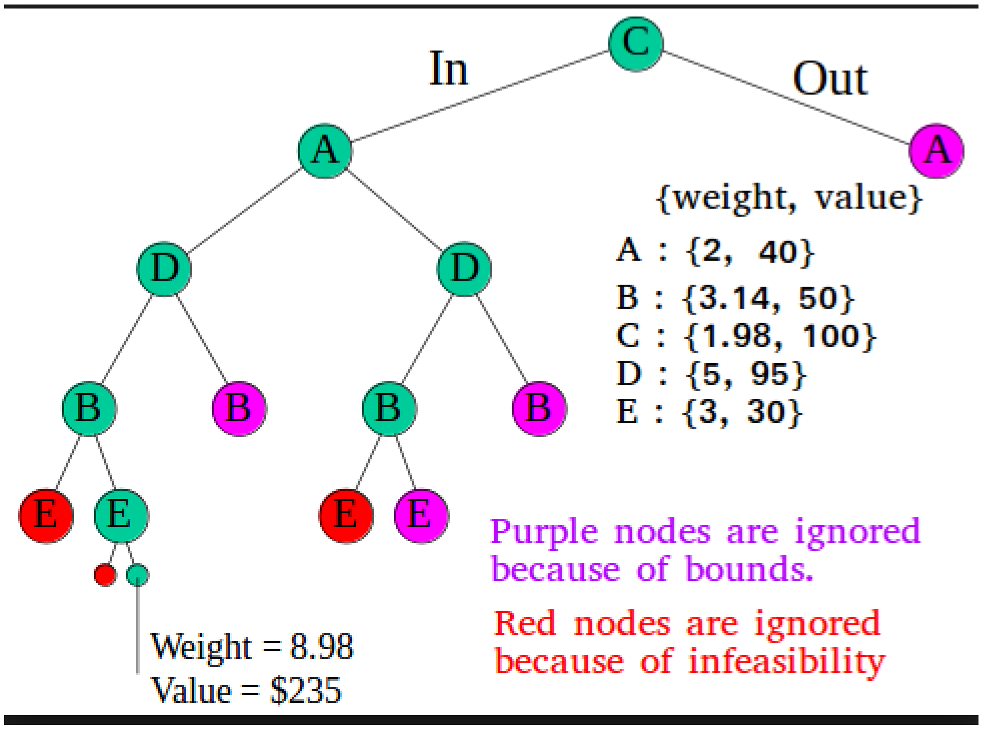


Figure 5: Branch and Bound Search

A * Search (AND graph)

The previous heuristics methods fail to describe how the shortest distance to a goal should be estimated. It is best-known form of Best First Search. It avoids expanding paths that are already expensive, but expands most promising paths first.

A* algorithm generates all successor nodes and computes an estimate of the distance (cost) from the start node to a goal node through each of the successors. It then chooses the successor with the shortest estimated distance for expansion. The successor for this node are then generated, their distance estimated, and the process continues until a goal is found or the search ends in failure.

The heuristic estimation function for A* can be given by:

f(n) = g(n) + h(n), where

- ***g(n)*** is the cost (so far) to reach the node
- ***h(n)*** is the estimated cost to get from the node to the goal
- ***f(n)*** is the estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

Now let us consider some desirable properties of heuristic search algorithms:

(i) **Admissibility condition**: Algorithm A is admissible if it is guaranteed to return an optimal solution when one exists.

(ii) **Completeness condition:** Algorithm A is complete if it always terminates with a solution when one exists.

(iii) **Dominance property:** Let $\mathbf{A_1}$ and $\mathbf{A_2}$ be admissible algorithm with heuristic estimation functions $\mathbf{h^*_1}$ and $\mathbf{h^*_2}$ respectively. A_1 is said to be more informed than $\mathbf{A_2}$, whenever $\mathbf{h^*_1(n)>h^*_2(n)}$ for all **n**. $\mathbf{A_1}$ is also said to dominate $\mathbf{A_2}$.

(iv) **Optimality Property:** Algorithm A is optimal over a class of algorithms if A dominates all members of the class.

A* algorithm is both complete and admissible. Thus, A* will always find an optimal path if one exists. The efficiency of an A* algorithm depends on how closely h* approximates h and the cost of the computing f*.

Local Search Algorithms

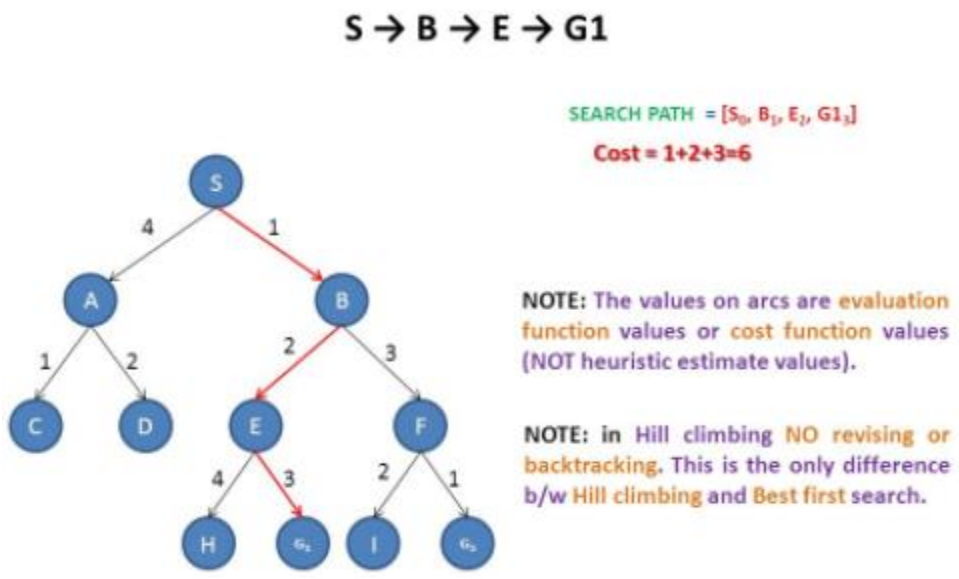
Local search algorithms operates using single current state and generally move only to the neighbors of that state. Typically the paths followed by the search are not retained. This algorithms has two key advantages:

(i) they use very little memory

(ii) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Hill-Climbing Search

It is like DFS where most promising node is selected for expansion. It is an iterative algorithm that starts with an arbitrary solution to a problem and attempts to find a better solution by changing a single element of the solution incrementally. If the change produces a better solution, an incremental change is taken as a new solution. This process is repeated until there are no further improvements. An example of hill climbing is shown below:



SEARCH PATH = [S_0, B_1, E_2, $G1_3$]

Cost = 1+2+3=6

NOTE: The values on arcs are evaluation function values or cost function values (NOT heuristic estimate values).

NOTE: in Hill climbing NO revising or backtracking. This is the only difference b/w Hill climbing and Best first search.

Figure 6: Hill-Climbing

Advantages: Hill climbing can produce substantial savings over blind searches when an informative, reliable function is available to guide the search to a global goal. It suffers from some serious drawbacks when this is not the case.

Disadvantage − this algorithm is neither complete, nor optimal. Potential problem types named after certain terrestrial anomalies are the **foothill, ridge, and plateau**.

A **foothill** trap results when local maxima or peaks are found. In this case the children all have less promising goal distances than the parent node. The search is essentially trapped at the at the local node with no try moving in some arbitrary direction a few generations in the hope that the real goal direction will become evident, backtracking to an ancestor node and trying a secondary path choice, or altering the computation procedure to expand ahead a few generations each time before choosing a path.

A **ridge** occurs when several adjoining nodes have a higher values than surrounding nodes whereas a **plateau** may occur during search when in an area all neighboring nodes will have the same value.

Local Beam Search

In this algorithm, it holds **k** number of states at any given time. At the start, these states are generated randomly. The successors of these **k** states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.

Otherwise the (initial **k** states and **k** number of successors of the states = **2k**) states are placed in a pool. The pool is then sorted numerically. The highest **k** states are selected as new initial states. This process continues until a maximum value is reached.

Travelling Salesman Problem

In this algorithm, the objective is to find a low-cost tour that starts from a city, visits all cities en-route exactly once and ends at the same starting city.

PROBLEM REDUCTION

Problem reduction search is a basic problem-solving technique of AI. It involves reducing a problem to a set of easier sub-problems whose solutions, if found, can be combined to form a solution to the hard problem.

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution. The AND-OR (AO*) graphs are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND arc may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph:

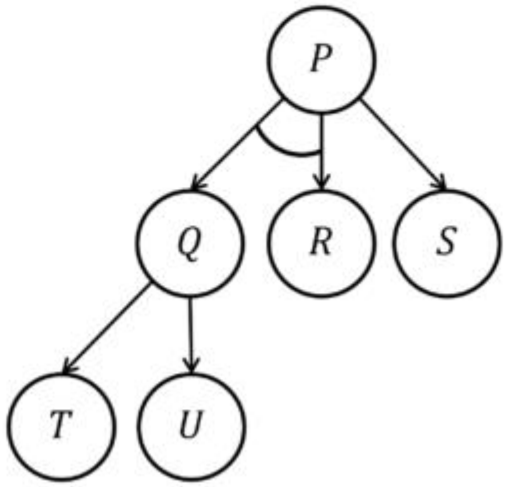


Figure 7: **AND - OR graph**

The above graph represents the search space for solving the problem **P**, using the goal-reduction methods:

(i) P if Q and R, (ii) P if S, (iii) Q if T, (iv) Q if U

The algorithm is a variation of the original given by Nilsson (1971). It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for **AND** node solutions which require solutions to all successor nodes. A solution is found when the start node is labeled as solved.

AO* algorithm

(i) Place the start node ***s*** on open

(ii) Using the search tree constructed thus far, compute the most promising solution tree **T_0**

(iii) Select a node n that is both on open and a part of **T_0**. Remove n from open and place it on closed.

(iv) If **n** is a terminal goal node, label n as solved. If the solution of n results in any of **n's** ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where **T_0** is the solution tree. Remove from open all nodes with a solved ancestor.

(v) If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of **n's** ancestors become unsolvable label them unsolvable as well. Remove from open all nodes with unsolvable ancestors.

(vi) Otherwise, expand node n generating all of its successors. For each such successors to give individual sub-problems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h* for each newly generated node and place all such nodes that do not yet have descendants on open. Next, recompute the values of h* at n and each ancestor of **n**.

(vii) Return to step **2**.

CONSTRAINTS SATISFACTION

Constraint satisfaction problems or CSPs are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria. A constraint is a restriction of the feasible solutions in an optimization problem. Many problems can be stated as constraints satisfaction problems.

Definition: "A **Constraint Satisfaction Problem (CSP)** is characterized by:

• **a set of variables** {x1, x2, .., xn},

• **for each variable xi a domain Di with the possible values for that variable**, and

• a **set of constraints**, i.e. relations, that are assumed to hold between the values of the variables. These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.

Representation of CSP

A CSP is usually represented as an undirected graph, called Constraint Graph where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyper arcs.

A constraint can affect any number of variables form 1 to n (n is the number of variables in the problem). If all the constraints of a CSP are binary, the variables and constraints can be represented in a constraint graph and the constraint satisfaction algorithm can exploit the graph search techniques.

The conversion of arbitrary CSP to an equivalent binary CSP is based on the idea of introducing a new variable that encapsulates the set of constrained variables. This newly introduced variable, we call it an encapsulated variable, has assigned a domain that is a Cartesian product of the domains of individual variables. Note, that if the domains of individual variables are finite than the Cartesian product of the domains, and thus the resulting domain, is still finite.

Now, arbitrary n-ary constraint can be converted to equivalent unary constraint that constrains the variable which appears as an encapsulation of the original individual variables. As we mentioned above, this unary constraint can be immediately satisfied by reducing the domain of encapsulated variable. Briefly speaking, n-ary constraint can be substituted by an encapsulated variable with the domain corresponding to the constraint.

This is interesting because any constraint of higher arity can be expressed in terms of binary constraints. Hence, binary CSPs are representative of all CSPs.

Solving CSPs

Four solution for CSPs are: (i) generate-and-Test (ii) Backtracking, (iii) Consistency Driven and (iv) Forward Checking.

1 Generate and Test: We generate one by one all possible complete variable assignments and for each we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop we test each constraint. In most situation this method is intolerably slow.

2 Backtracking: We order the variables in some fashion, trying to place first the variables that are more highly constrained or with smaller ranges. This order has a great impact on the efficiency of solution algorithms and is examined elsewhere. We start assigning values to variables. We check constraint satisfaction at the earliest possible time and extend an assignment if the constraints involving the currently bound variables are satisfied.

3 Consistency Driven Techniques: Consistency techniques effectively rule out many inconsistent labeling at a very early stage, and thus cut short the search for consistent labeling. These techniques have since proved to be effective on a wide variety of hard search problems. The consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to done. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could). In binary CSPs, various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. Thus, the potential inconsistency is detected as soon as possible.

4 Forward checking: It is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

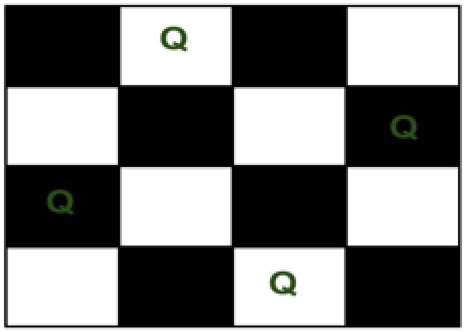
Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when each assignment is added to the current partial solution.

Comparison of Propagation Techniques

More constraint propagation at each node will result in the search tree containing fewer nodes, but the overall cost may be higher, as the processing at each node will be more expensive. In one extreme, obtaining strong n-consistency for the original problem would completely eliminate the need for search, but as mentioned before, this is usually more expensive than simple backtracking. Actually, in some cases even the full look ahead may be more expensive than simple backtracking. That is the reason why forward checking and simple backtracking are still used in applications.

Some examples of CSP’s are:

Example 1: The n-Queen problem is the problem of putting n chess queens on an n×n chessboard such that none of them is able to capture any other using the standard chess queen's moves. The color of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal. For example, following is a solution for 4 Queen problem:



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.

{0, 1, 0, 0}

{0, 0, 0, 1}

{1, 0, 0, 0}

{0, 0, 1, 0}

Example 2: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using k colors, so that no two neighboring countries have the same color.

Example 4: The Boolean satisfiability problem (SAT) is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? In mathematics, a formula of propositional logic is said to be satisfiable if truth-values can be assigned to its variables in a way that makes the formula true. The class of satisfiable propositional formulas is NP-complete. The propositional satisfiability problem (SAT), which decides whether or not a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmic, artificial intelligence, hardware design and verification.

Example 5: A crypt-arithmetic problem: In the following pattern

```
  S E N D
+ M O R E
=========
M O N E Y
```

We have to replace each letter by a distinct digit so that the resulting sum is correct. All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc. are instances of the same pattern.

QUESTIONS

1. Discuss different types of un-informed search.
2. Discuss various heuristic search.
3. Compare and contrast between DFS and BFS.
4. Describe Hill climbing search method with an example.
5. What is Best First Search?
6. Discuss A* search algorithm.
7. What are the desirable properties of heuristic search algorithms? Explain.
8. Explain AO* algorithm.

SUGGESTED READINGS

1. Introduction to Artificial Intelligence and Expert systems by Dan W. Patterson.

2. Artificial Intelligence: A modern approach by Stuart Russell and Peter Norvig.