Rishabh Indoria
21F3001823

AI: Search Methods for Problem-Solving

December 10, 2024

1 Introduction and History

1.1 Intelligent Agents

- **A First Course in Artificial Intelligence** by Deepak Khemani. First 7 chapters will be covered.
- **Intelligent Agents**: An entity that is persistent(it's there all the time), autonomous, proactive(decide what goals to achieve next) and goal directed(once it has goals, it will follow them). Human beings are also agents.
- An intelligent agent in a world carries a model of the world in its "head". the model may be an abstraction. A self-aware agent would model itself in the world model.
- Signal$\rightarrow$Symbol$\rightarrow$Signal
- Sense(Signal Processing) $\rightarrow$ Deliberate(Neuro fuzzy reasoning + Symbolic Reasoning) $\rightarrow$ Act(Output Signal)

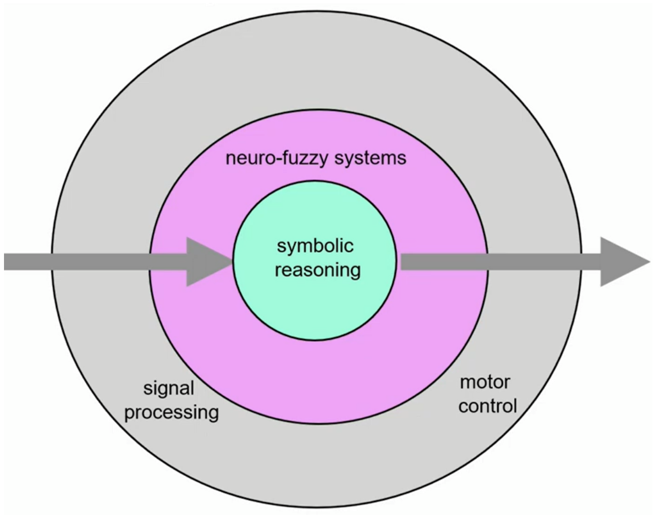


Figure 1: Information Processing View of AI

- **Intelligence**: **Remember the past and learn from it.** Memory and experience(case based reasoning), Learn a model(Machine learning), Recognize objects, faces, patterns(deep neural networks).
  Understand the present. Be aware of the world around you. Create a model of the world(knowledge representation), Make inferences from what you know(logic and reasoning).
  Imagine the future. Work towards your goals. Trial and error(heuristic search), Goals, plans, and actions(automated planning).

1.2 Human Cognitive Architecture

- **Knowledge and Reasoning**: What does the agent know and what else does the agent know as a consequence of what it knows.
- **Semiotics**: A symbol is something that stands for something else. All languages, both spoken and written, are semiotic systems.
- **Biosemiotics**: How complex behaviour emerges when simple systems interact with each other through signs.
- **Reasoning**: The manipulation of symbols in a meaningful manner.

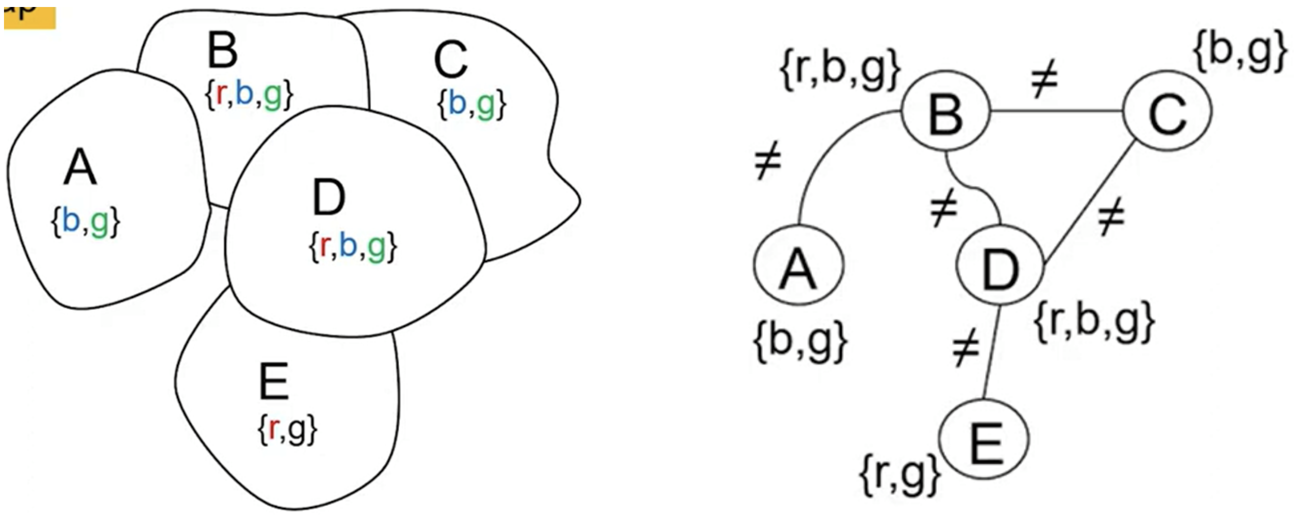
1.3 Problem-Solving

- An autonomous agent in some world has a goal to achieve and a set of actions to choose from to strive for the goal.
- We deal with simple problems first, i.e., the world is static, the world is completely known, only one agent changes the world, action never fail, representation of the world is taken care of.

2 Search Methods

2.1 State Space Search

- We start with the map coloring problem, where we want to color each region in the map with an allowed color such that no two adjacent regions have the same color.



(a) Map with Constraints (b) Constraint Graph

Figure 2: Map Coloring Problem

  Brute Force: Try all combinations of color for all regions.
  Informed Search: Choose a color that does not conflict with neighbors.
  General Search: Pose the problem to serve as an input to a general search algorithm.
  Map coloring can be posed as a constraint satisfaction problem or as a state space search, where the move is to assign a color to a region in a given partially colored state.

- **General Purpose methods**: Instead of writing a custom program for every problem to be solved, these aim to write search algorithms into which individual methods can be plugged in. One of them is State Space Search.
- **State or Solution Space Search**: Describe the given state, devise an operator to choose an action in each state, and then navigate the state space in search of the desired or goal state. Also known as Graph Search.
  A **state** is a representation of a situation.
  A *MoveGen* function captures the moves that can be made in a given state.
  It returns a set of states - *neighbors* - resulting from the moves.
  The state space is an implicit graph defined by the *MoveGen* function.
  A *GoalTest* function checks if the given state is a *goal* state.
  A *search algorithm* navigates the state space using these two functions.
- **The Water Jug Problem**: You have three jugs with capacity 8, 5 and 3 liters. Any state can be written as $[a, b, c]$, where a, b, c are the amount of water present in each jug
  Start state: The 8-liter jug is filled with water, and the other two are empty, $[8, 0, 0]$. It can be seen that at any state the total amount of water remains the same.
  Goal: You are required to measure 4 liters of water. So, our goal state is $[4, x, y]$ or $[x, 4, y]$ or $[x, y, 4]$. A *GoalTest* function can be written as

Algorithm 1 GoalTest for the Water Jug Problem

```
Require: $[a, b, c]$, the state which needs to be checked
if $a = 4$ or $b = 4$ or $c = 4$ then
    return True
end if
return False
```

- A few other examples are **The Eight Puzzle**, **The Man, Goat, Lion, Cabbage** and **The N-Queens Problem**.

Before studying different algorithms, we need to familiarize ourselves with some pseudocode syntax.

2.2 General Search Algorithms

- Searching is like treasure hunting. Our approach would be to generate and test where we traverse the space by generating new nodes and test each node whether it is the goal or not.
- **Simple Search 1**: Simply pick a node N from OPEN and check if it is the goal.

Algorithm 2 Simple Search 1

```
▷ S is the initial state
OPEN ← $\{S\}$
while OPEN is not empty do
    pick some node N from OPEN
    OPEN ← OPEN - $\{n\}$
    if GoalTest(N) = True then
        return N
    else
        OPEN ← OPEN ∪ MoveGen(N)
    end if
end while
return FAILURE
```

This algorithm may run into an infinite loop, to address it we have **Simple Search 2**.

Algorithm 3 Simple Search 2

```
▷ S is the initial state
OPEN ← $\{S\}$
CLOSED ← {}
while OPEN is not empty do
    pick some node N from OPEN
    OPEN ← OPEN - $\{n\}$
    CLOSED ← CLOSED ∪ $\{N\}$
    if GoalTest(N) = True then
        return N
    else
        OPEN ← OPEN ∪ {MoveGen(N) - CLOSED}
    end if
end while
return FAILURE
```

We can modify this a bit further by doing, OPEN ← OPEN ∪ {*MoveGen(N)* - CLOSED - OPEN}, this lowers the state space even more.

2.3 Planning and Configuration Problems

- **Planning Problems**: Goal is known or describe, path is sought. Examples include River crossing problems, route finding etc.
- **Configuration Problems**: A state satisfying a description is sought. Examples include N-queens, crossword puzzle, Sudoku, etc.

- The simple search algorithms will work for configuration problems, but they won't return any path.
- **NodePairs**: Keep track of parent nodes. Each node is a pair (*currentNode*, *parentNode*).
- **Depth First Search**: OPEN is a stack data structure

Algorithm 4 Depth First Search

```
OPEN ← (S, null) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _) ← nodePair
    if GoalTest(N) = True then
        return ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    children ← MoveGen(N)
    newNodes ← RemoveSeen(children, OPEN, CLOSED)
    newPairs ← MakePairs(newNodes, N)
    OPEN ← newPairs ++ tail OPEN
end while
return [ ]

function RemoveSeen(nodeList, OPEN, CLOSED)
    if nodeList = [ ] then
        return [ ]
    end if
    node ← head nodeList
    if OccursIn(node, OPEN) or OccursIn(node, CLOSED) then
        return RemoveSeen(tail nodeList, OPEN, CLOSED)
    end if
    return node : RemoveSeen(tail nodeList, OPEN, CLOSED)
end function

function OccursIn(node, nodePairs)
    if nodePairs = [ ] then
        return FALSE
    else if node = first head nodePairs then
        return TRUE
    end if
    return OccursIn(node, tail nodePairs)
end function

function MakePairs(nodeList, parent)
    if nodeList = [ ] then
        return [ ]
    end if
    return (head nodeList, parent) : MakePairs(tail nodeList, parent)
end function

function ReconstructPath(nodePair, CLOSED)
    (node, parent) ← nodePair
    path ← node : [ ]
    while parent is not null do
        path ← parent : path
        CLOSED ← SkipTo(parent, CLOSED)
        (_, parent) ← head CLOSED
    end while
    return path
end function
```

Algorithm 5 Depth First Search continued

```
function SkipTo(parent, nodePairs)
    if parent = first head nodePairs then
        return nodePairs
    end if
    return SkipTo(parent, tail nodePairs)
end function
```

- **Breadth First Search**: We use a queue for the OPEN set.

Algorithm 6 Breadth First Search

```
OPEN ← (S, null) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _) ← nodePair
    if GoalTest(N) = True then
        return ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    children ← MoveGen(N)
    newNodes ← RemoveSeen(children, OPEN, CLOSED)
    newPairs ← MakePairs(newNodes, N)
    ▷ This is the only line that is different, we now add newPairs at the end of the list
    OPEN ← tail OPEN ++ newPairs
end while
return [ ]
```

- BFS always generates the shortest path, whereas DFS may not generate the shortest path.
- **Time Complexity**: Assume constant branching factor b and assume the goal occurs somewhere at depth d. In the best case the goal would be the first node scanned at depth d and in the worst case the goal would b3 the last node scanned.
  Best Case: $N_{DFS} = d + 1$ and $N_{BFS} = \frac{b^d - 1}{b-1} + 1$
  Worst Case: $N_{DFS} = N_{BFS} = \frac{b^{d+1}-1}{b-1}$
  Average: $N_{DFS} \approx \frac{b^d}{2}$ and $N_{BFS} \approx \frac{b^d(b+1)}{2(b-1)}$, $N_{BFS} \approx N_{DFS}(\frac{b+1}{b-1})$

	Depth First Search	Breadth First Search
Time	Exponential	Exponential
Space	Linear	Exponential
Quality of Solution	No guarantees	Shortest path
Completeness	Not for infinite search space	Guaranteed to terminate if solution path exists

Table 1: DFS vs BFS

- **Depth Bounded DFS**: Do DFS with a depth bound d. It is not complete and does not guarantee the shortest path. Given below is a modified version that returns the count of nodes visited, to get the original version just remove count.

Algorithm 7 Depth Bounded DFS

```
count ← 0
OPEN ← (S, null, 0) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _, depth) ← nodePair
    if GoalTest(N) = True then
        return count, ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    if depth < depthBound then
        children ← MoveGen(N)
        newNodes ← RemoveSeen(children, OPEN, CLOSED)
        newPairs ← MakePairs(newNodes, N, depth + 1)
        OPEN ← newPairs ++ tail OPEN
        count ← count + length newPairs
    else
        OPEN ← tail OPEN
    end if
end while
return count, [ ]
```

- **Depth First Iterative Deepening**: Iteratively increase depth bound. Combines best of BFS and DFS. This will give the shortest path, but we have to be careful that we take correct nodes from CLOSED. $N_{DFID} \approx N_{BFD}\frac{b}{b-1}$ for large b.

Algorithm 8 Depth First Iterative Deepening

```
count ← −1
path ← [ ]
depthBound ← 0
repeat
    previousCount ← count
    (count, path) ← DB-DFS($S, depthBound$)
    depthBound ← depthBound +1
until (path is not empty) or (previousCount == count)
return path
```

- DFID-C as compared to regular DFID-N adds new neighbors to OPEN list and also adds neighbors that are in CLOSED but not in OPEN list. This allows DFID-C to get the shortest path whereas DFID-N may not get the shortest path.
- The monster that AI fights is Combinatorial Explosion.
- All these algorithms we studied are blind algorithms or uniformed search, they don't know where the goal is.

2.4 Heuristic Search

- Testing the neighborhood and following the steepest gradient identifies which neighbors are the lowest or closest to the bottom.
- The heuristic function $h(N)$ is typically a user defined function, $h(Goal) = 0$.
- **Best First Search**: Instead of having a simple stack or queue we use a priority queue sorted based on heuristic function. Search frontier depends upon the heuristic function. If graph is finite then this is complete and quality of solution will head towards the goal but may not give the shortest path.

Algorithm 9 Best First Search

```
OPEN ← (S, null, h(S)) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _, _) ← nodePair
    if GoalTest(N) = True then
        return ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    children ← MoveGen(N)
    newNodes ← RemoveSeen(children, OPEN, CLOSED)
    newPairs ← MakePairs(newNodes, N)
    ▷ Again, this is pretty much the most important line
    OPEN ← sort_h(newPairs ++ tail OPEN)
end while
return [ ]
```

- For The eight puzzle we can define a few heuristic functions as follows:
  $h_1(n)$ = number of tiles out of place, Hamming distance.
  $h_2(n) = \sum_{\text{for each tile}}$ Manhattan distance to its destination.
- **Hill Climbing**: A local search algorithm, i.e., move to the best neighbor if it is better, else terminate. In practice sorting is not needed, only the best node. This has burnt its bridges by not storing OPEN. We are interested in this because it is a constant space algorithm, which is a vast improvement on the exponential space for BFS and DFS. It's time complexity is linear. It treats the problem as an optimization problem.

Algorithm 10 Hill Climbing

```
node ← Start
newNode ← head(sort_h(moveGen(node)))
while $h(newNode) < h(node)$ do
    node ← newNode
    newNode ← head(sort_h(moveGen(node)))
end while
return node
```

2.5 Escaping Local Optima

- **Solution Space Search**: Formulation of the search problem such that when we find the goal node we have the solution, and we are done.
- **Synthesis Methods**: Constructive methods. Starting with the initial state and build the solution state piece by piece.
- **Perturbation Methods**: Permutation of all possible candidate solutions.
- **SAT problem**: Given a boolean formula made up of a set of propositional variables $V = \{a, b, c, d, e, ...\}$ each of which can be $true$ or $false$, or 1 or 0, to find an assignment of variables such that the given formula evaluates to $true$ or 1.
  A SAT problem with N variables has 2^N candidates.
- **Travelling Salesman Problem**: Given a set of cities and given a distance measure between every pair of cities, the task is to find a Hamiltonian cycle, visiting each city exactly once, having the least cost.
  1. **Nearest Neighbor Heuristic**: Start at some city, move to nearest neighbor as long as it does not close the loop prematurely.
  2. **Greedy Heuristic**: Sort the edges, then add the shortest available edge to the tour as long as it does not close the loop prematurely.
  3. **Savings Heuristic**: Start with $n - 1$ tours of length 2 anchored on a base vertex and performs $n - 2$ merge operations to construct the tour.
  4. **Perturbation operators**: Start with some tour, then choose two cities and interchange, this gives the solution space. Heuristic function is $saving = cost(base, a) + cost(base, b)$ - $cost(a, b)$

5. **Edge Exchange**: Similar to above but now instead of choosing cities we are choosing edges. Can be 2-edge exchange, 3-edge, 4-edge, etc. City exchange is a special case of 4-edge exchange.

- Solution space of TSP grows in order factorial, much faster than SAT problem.
- A collection of problems with some solutions is available here.
- To escape local minima we need something called **exploration**.
- **Beam Search**: Look at more than one option at each level. For a beam width b, look at best b options. Heavily memory dependent.

Algorithm 11 Beam Search

```
BeamSearch($S, w$)
OPEN ← [$S$]
$N$ ← $S$
do
    $bestEver$ ← $N$
    if Goal-Test(OPEN) = True then
        return goal from OPEN
    end if
    $neighbors$ ← MoveGen(OPEN)
    OPEN ← take w $sort_h(neighbors)$
    $N$ ← head(OPEN)
while $h(N)$ is better than $h(bestEver)$
return $bestEver$
```

- **Variable Neighborhood Descent**: Essentially we are doing hill climbing with different neighborhood functions. The idea is to use sparse functions before using denser functions, so the storage is not high. The algorithm assumes that there are N *moveGen* functions sorted according to the density of the neighborhoods produced.

Algorithm 12 Variable Neighborhood Descent

```
VariableNeighbourhoodDescent( )
node ← start
for $i$ ← 1 to $n$ do
    $moveGen$ ← MoveGen($i$)
    $node$ ← HillClimbing($node, moveGen$)
end for
return $node$
```

- **Best Neighbor**: Another variation of Hill Climbing, simply move to the best neighbor regardless of whether it is better than current node or not. This will require an external criterion to terminate. It will not escape local maxima, as once it escapes it will go right back to it in the next iteration.

Algorithm 13 Best Neighbor

```
BestNeighbor( )
$N$ ← $start$
$bestSeen$ ← $N$
while some termination criterion do
    $N$ ← best($moveGen(N)$)
    if $N$ better than $bestSeen$ then
        $bestSeen$ ← $N$
    end if
end while
return $bestSeen$
```

- **Tabu Search**: Similar to Best Neighbor but not allowed back immediately. An aspiration criterion can be added that says that if a tabu move result in a node that is better than bestSeen then it is allowed. To drive this search into newer areas, keep a frequency array and give preference to lower frequency bits or bits that have been changed less.

Algorithm 14 Tabu Search

```
TabuSearch( )
1: $N \leftarrow start$
2: $bestSeen \leftarrow N$
3: while some termination criterion do
4:     $N \leftarrow$ BEST(ALLOWED($moveGen(N)$))
5:     if $N$ better than $bestSeen$ then
6:         $bestSeen \leftarrow N$
7:     end if
8: end while
9: return $bestSeen$
```

3 Stochastic Search

Very often to escape local minima we use Stochastic/Randomized methods instead of Deterministic methods.

3.1 Iterated Hill Climbing

- **2SAT**: Problems where each clause has at most two literals, known to be solved in polynomial time. However, if we add even 1 more literal it would become NP-complete or exponential.
- Iterated hill climbing says that we should perform hill climbing with multiple different starting nodes chosen randomly to have a higher probability of finding the goal node.

Algorithm 15 Iterated Hill Climbing

```
Iterated-Hill-Climbing($N$)
1: $bestNode \leftarrow$ random candidate solution
2: for $i \leftarrow 1$ to $N$ do
3:     $currentBest \leftarrow$ Hill-Climbing(new random candidate solution)
4:     if $h(currentBest)$ is better than $h(bestNode)$ then
5:         $bestNode \leftarrow currentBest$
6:     end if
7: end for
8: return $bestNode$
```

3.2 Stochastic Actions

- To move or not to move is the question.
- Idea is we would like to consider actions that are good in terms of heuristic functions, but sometimes we even want to move when heuristic function is not necessarily good.
- **Random Walk**: Choose a random node and move there, hill climbing of stochastic search.

Algorithm 16 Random Walk

```
RandomWalk( )
1: $node \leftarrow$ random candidate solution or start
2: $bestNode \leftarrow node$
3: for $i \leftarrow 1$ to $n$ do
4:     $node \leftarrow$ RandomChoose(MoveGen($node$))
5:     if $node$ is better than $bestNode$ then
6:         $bestNode = node$
7:     end if
8: end for
```

- **Stochastic Hill Climbing**: Let the algorithm be at the current bode v_c, then we select a random neighbor v_N and calculate $\Delta E = (eval(v_N) - eval(v_c))$.
  If ΔE is positive, it means that v_N is better than v_c, then the algorithm should move to it with a higher probability.
  If ΔE is negative, it means that v_N is better than v_c, then the algorithm should move to it with a lower

probability.
The probability is computed using the sigmoid function, which is given as

$$\text{Probability } P = \frac{1}{1 + e^{-\frac{\Delta E}{T}}}$$

where T is a parameter. The algorithm essentially combines exploration with exploitation.

- **Annealing**: In metallurgy and materials science, **annealing** is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above its recrystallization temperature, maintaining a suitable temperature for an appropriate amount of time and then cooling.
  In annealing, atoms migrate in the crystal lattice and the number of dislocations decreases, leading to a change in ductility and hardness. As the material cools it recrystallizes.

- **Simulated Annealing**: Essentially tries to mimic the annealing process, begins with exploration and ends with exploitation. Annealing is also where ΔE and T come from, essentially meaning Energy and Temperature.

Algorithm 17 Simulated Annealing

```
    SimulatedAnnealing( )
 1: node ← random candidate solution or start
 2: bestNode ← node
 3: T ← some large value
 4: for time ← 1 to numberOfEpochs do
 5:     while some termination criteria do                ▷ M cycles in a sample case
 6:         neighbor ← RandomNeighbor(node)
 7:         $\Delta E$ ← Eval(neighbor) − Eval(node)
 8:         if Random(0, 1) < Sigmoid($\Delta E$, T) then
 9:             node ← neighbor
10:             if Eval(node) > Eval(bestNode) then
11:                 bestNode ← node
12:             end if
13:         end if
14:     end while
15:     T ← CoolingFunction(T, times)
16: end for
17: return bestNode
```

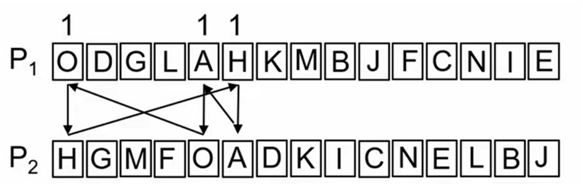
3.3 Genetic Algorithms

- Survival of the fittest. Inspired by the process of natural selection.
- A class of methods for optimization problems, more generally known as Evolutionary Algorithms.
- Implemented on a population of candidates in the solution space. A fitness function evaluates each candidate. The fittest candidates get to mate and reproduce.
- **Artificial Selection**: Given a population of candidate solutions we do a three step iterative process
  1. **Reproduction**: Clone each candidate in proportion to its fitness. Some may get more than one copy, some none.
  2. **Crossover**: Randomly mate the resulting population and mix up the genes.
  3. **Mutation**: Once in a while, in search of a missing gene.
- **Selection**: Imagine a roulette wheel on which each candidate in the parent population $\{P_1, P_2, ..., P_N\}$ is represented as a sector. The angle subtended by each candidate is proportional to its fitness. The roulette wheel is spun N times. Then the selected parents are added one by one to create a new population.
- **Crossover Operators**: A *single point crossover* simply cuts the two parents at a randomly chosen point and recombines them to form two new solution strings. It can be at multiple points as well, idea is to mix up the genes.

Algorithm 18 Genetic Algorithm

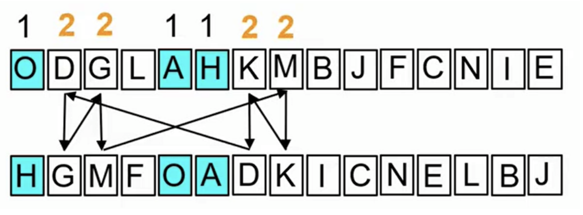
```
    GENETIC-ALGORITHM( )
1:  $P \leftarrow$ create $N$ candidate solutions                                   ▷ initial population
2:  repeat
3:      compute fitness value for each member of $P$
4:      $S \leftarrow$ with probability proportional to fitness value, randomly select $N$ member from $P$.
5:      $offspring \leftarrow$ partition $S$ into two halves, and randomly mate and crossover members to generate $N$ offsprings.
6:      With a low probability mutate some offsprings
7:      Replace $k$ the weakest members of $P$ with $k$ strongest offsprings
8:  until some termination criteria
9:  return the best member of $P$
```

- A large diverse population size is necessary for the performance of genetic algorithm to work.
- One issue is how to represent candidate solutions?, as they can be of any type. One simple solution is to convert the solution into a string this will require an additional function.
- In TSP, one problem is also that during crossover some cities can get repeated.
- **Cycle Crossover**: Identify cycles, look at below image, easier to understand.



1 1 1
P1 O D G L A H K M B J F C N I E
P2 H G M F O A D K I C N E L B J

(a) TSP cycle 1

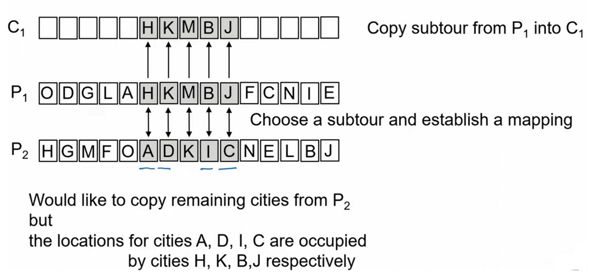


(b) TSP cycle 2

Figure 3: TSP Identifying cycles

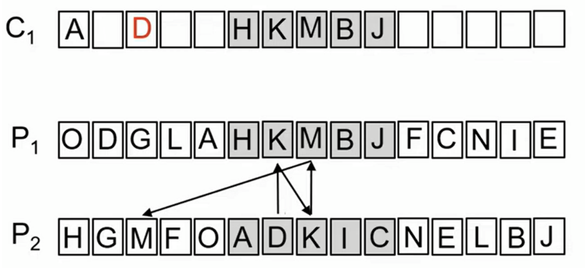
Once cycles are identified, then C_1 gets odd numbered cycles from P_1 and even numbered cycles from P_2 the other go to C_2.

- **Partially Mapped Crossover(PMX)**: Identify some cities that form a subtour and establish a mapping.



C1 H K M B J
Copy subtour from P1 into C1
P1 O D G L A H K M B J F C N I E
Choose a subtour and establish a mapping
P2 H G M F O A D K I C N E L B J
Would like to copy remaining cities from P2 but the locations for cities A, D, I, C are occupied by cities H, K, B,J respectively

(a) Choose Subtour



C1 A D H K M B J
P1 O D G L A H K M B J F C N I E
P2 H G M F O A D K I C N E L B J

(b) Copy occupied spaces

Figure 4: Partially Mapped Crossover

- **Order Crossover**: Copy a subtour from P_1 into C_1 and the remaining from P_2 in the order they occur in P_2.
- For all the above we were using path representation of TSP, there is another representation of TSP.
- **Adjacency Representation**: The cities are arranged based on where they come from in the tour with respect to the index. For example if $A \rightarrow H$, then at index 0 we would have H.
- **Alternating Edges Crossover**: From a given city X choose the next city Y from P_1 and from the city Y choose the next city from P_2 and so on. It is possible to run into cycles, need to be careful.
- **Heuristic Crossover**: For each city choose the next from that parent P_1 or P_2 whichever is closer.

- **Ordinal Representation**: Replace the name of the city is Path Representation one by one to its current numeric index. At the start all cities A, B, C, D, E will have numeric index $1, 2, 3, 4, 5$, now consider if we add C to the representation the new indexing would be for cities A, B, D, E, we have $1, 2, 3, 4$. The **advantage** is that single point crossover produces valid offspring.

3.4 Ant Colony Optimization

- **Emergent Systems**: Collections of simple entities organize themselves and a larger more sophisticated entity emerges. The behavior of this complex system is a property that emerges from interactions amongst its components.
- **Conway's Game of Life**: A cellular automaton in which cells are alive or dead. Each cell obeys the following rules to decide its fate in the next time step.

Cell state	Number of alive neighbors	New cell state
alive	< 2	dead
alive	2 or 3	alive
alive	> 3	dead
dead	3	alive

Table 2: Rules for Game of Life

- Illusion of movement can be seen in an example called **Gosper's Glider Gun**.
- **Chaos and Fractals**: A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop. Drive by recursion, fractals are images of dynamic systems, the pictures of chaos.
- Let 5 ants A, B, C, D, and E go out in search for food. Each ant lays a trail of pheromone where it goes. Each ant lays a trail of pheromone where it goes. More ants that emerge will tend to follow some pheromone trail.
- Let's say A finds some food, then it will follow its pheromone trail back and the other ants will continue their search.
- Eventually, as more ants travel on the trail they deposit more pheromone and the trail gets stronger and stronger, eventually becoming the caravan we might have seen raiding our food.
- They tend to find the shortest path.
- **Ant Colony Optimization**: We try to capitalize on this method. Each ant constructs a solution using a stochastic greedy method using a combination of a heuristic function and pheromone trail following.
- This is related to the class of algorithms known as *Swarm Optimization*.

Algorithm 19 Ant Colony Optimization for TSP

```
TSP-ACO( )
 1: $bestTour \leftarrow$ NIL
 2: repeat
 3:     randomly place $M$ ants on $N$ cities
 4:     for each ant $a$ do
 5:         for $n \leftarrow 1$ to $N$ do
 6:             ant $a$ selects an edge from the distribution $P_n^a$
 7:         end for
 8:     end for
 9:     update $bestTour$
10:     for each ant $a$ do
11:         for each edge $(u, v)$ in the ant's tour do
12:             deposit pheromone $\propto$ 1/tour-length on edge $(u, v)$
13:         end for
14:     end for
15: until some termination criteria
16: return $bestTour$
```

- From a city i the k^{th} ant moves to city j with a probability given by

$$P_{ij}^{k}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} \times [\eta_{ij}]^{\beta}}{\sum_{h \in allowed_k(t)} ([\tau_{ih}(t)]^{\alpha} [\eta_{ih}(t)]^{\beta})}, & \text{if } j \in allowed_k(t) \text{ the cities ant } k \text{ is allowed to move to} \\ 0, & \text{otherwise} \end{cases}$$

  where $\tau_{ij}(t)$ is pheromone on edge$_{ij}$ and η_{ij} is called visibility which is inversely proportional to the distance between cities i and j.

- After constructing a tour in n time steps, each ant k deposits an amount of pheromone $\frac{Q}{L_k}$ on the edges it has traversed, which is inversely proportional to the cost of the tour L_k it found.

- Total pheromone deposited on edge$_{ij}$ is $\Delta\tau_{ij}(t, t+n)$

- The total pheromone on edge$_{ij}$ is updated as

$$\tau_{ij}(t+n) = (1-\rho) \times \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n)$$

  where ρ is the rate of evaporation of pheromone.

4 Finding Optimal TSP Tours

4.1 Finding Optimal Paths

- Breadth First Search finds a solution with the smallest *number* of moves. But if *cost* of all moves is no the same then an **optimal** solution may not be the one with the smallest number of moves.
- **Brute Force**: Simply search the entire search tree, computationally it is mindlessly expensive.
- **Goal**: Search as little of the space as possible while guaranteeing the optimal solution.
- A *Best First* approach to solving problems. Given a set of candidates a search algorithm has to choose from. Each candidate is tagged with an estimated cost of the complete solution.
- Branch and Bound in a refinement space
  1. Initial solution set contains all solutions.
  2. In each step we partition a set into two smaller sets.
  3. Until we can pick a solution that is fully specified.
  4. Each solution set needs to have an estimated cost.
  5. B&B will refine the solution that has the least estimated cost.
- An optimal solution can be guaranteed by ensuring that the estimated cost is a lower bound on actual cost.
- **Lower bound**: A solution will never be cheaper than it is estimated to be.
- Thus if a fully refined solution is cheaper than a partially refined one, the latter need not be explored - **PRUNING**. The higher the estimate, the better the pruning.
- Branch and Bound for TSP
  1. Let the candidate solutions be permutations of the list of city names.
  2. The initial set of candidates includes all permutations.
  3. Refine the cheapest set by specifying a specific segment.
  4. Heuristic: Choose the segment with minimum cost.
  5. For tighter estimates, edges that cause three or more segment cycles are avoided.
  6. We, can also exclude an edge that is a third edge for a node, because a city has only two neighbors in a tour.
  7. Higher estimates require more work.
- A lower bound estimate could be for each row add up the smallest two positive entries and divide by two. This may not feasible.
- The basic idea behind B&B is to prune those parts of a search space which cannot contain a better solution.
- Each candidate is tagged with an estimated cost of the complete solution.

- **Dijkstra's Algorithm**
  1. Begins by assigning infinite cost estimates to all nodes except the start node.
  2. It assigns color white to all the nodes initially.
  3. It picks the cheapest white node and colors it black.
  4. **Relaxation**: Inspect all neighbors of the new black node and check if a cheaper path has been found to them.
  5. If yes, then update cost of that node, and mark the parent node.

4.2 Algorithm A*

- A* achieves better performance by using heuristics to guide its search.
- For all nodes we will compute $f(n)$ which is made up of two components $g(n)$ and $h(n)$.
- $g(n)$ = actual cost of solution found from the start node to node n.
- $h(n)$ = estimated cost of the path from node n to goal node.
- Maintain a priority queue of nodes sorted on the f values
- Pick the lowest f value node, and check whether it is the goal node.
- Else generate its neighbors, and compute their f values
- Insert new nodes into the priority queue
- For existing nodes check for better path(like Dijkstra's algorithm)

Algorithm 20 A* Algorithm

```
    A*(S)
1:  default value of $g$ for every node is $+\infty$
2:  $parent(S) \leftarrow null$
3:  $g(S) \leftarrow 0$
4:  $f(S) \leftarrow g(S) + h(S)$
5:  $OPEN \leftarrow S$ :[ ]
6:  $CLOSED \leftarrow$ empty list
7:  while $OPEN$ is not empty do
8:      $N \leftarrow$ remove node with the lowest $f$ value from $OPEN$
9:      add $N$ to CLOSED
10:     if GoalTest($N$) then
11:         return ReconstructPath($N$)
12:     end if
13:     for each neighbor $M \in$ MoveGen($N$) do
14:         if $g(N) + k(N, M) < g(M)$ then
15:             $parent(M) \leftarrow N$
16:             $g(M) \leftarrow g(N) + k(N, M)$
17:             $f(M) \leftarrow g(M) + h(M)$
18:             if $M \in OPEN$ then
19:                 continue
20:             end if
21:             if $M \in CLOSED$ then
22:                 Propagate-Improvement($M$)
23:             else
24:                 add $M$ to $OPEN$
25:             end if
26:         end if
27:     end for
28: end whilereturn empty list
```

- Propagate improvement simply does line 13 to 17 recursively.
- A* is admissible if

1. The branching factor is finite, otherwise you cannot even generate the neighbors.
2. Every edge has a cost greater than a small constant ϵ, however it is possible to get stuck in an infinite path with a finite cost.
3. For all nodes $h(n) \leq h^*(n)$

- If a path exists to the goal node, then the OPEN list always contains a node n' from an optimal path. Moreover, the f value of that node is not greater than the optimal value.

4.3 Weighted A*

- $f(n) = g(n) + wh(n)$, where w determines how much weight we give to the heuristic function.
- $w = 0$, with the **pull** of the source being the dominating factor and only shortest path in mind.
- $w \to \infty$, with the **push** towards the goal being the dominating factor and only speed termination in mind.
- Best First Search and Weighted A^* head off towards the goal directly.
- Branch and Bound and A^* keep checking if a cheaper path is available.
- $w = 0$ is essentially branch and bound, and $w \to \infty$ is essentially the best first search.

4.4 A* Space Saving Versions

- **Iterative Deepening** A^*: Similar to DFID, but instead of having a depth parameter, we set the depth as f-values, initially set the bound as $f(S)$ which is equal to $h(S)$. In subsequent cycles it extends the bound to the next unexplored f-value.

Algorithm 21 Iterative Deepening A^*

```
IterativeDeepeningA*($start$)
1: $depthBound \leftarrow f(S)$
2: while $TRUE$ do
3:     DepthBoundedDFS($start, depthBound$)
4:     $depthBound \leftarrow f(N)$ of cheapest unexpanded node on $OPEN$
5: end while
```

- Even when the state space grows quadratically the number of paths to each node grows exponentially.
- The DFS algorithm can spend a lot of time exploring these different paths if a CLOSED list is not maintained. The other problem is that in each cycle it extends the bound only to the next f-value.
- IDA* has linear space requirements however has no sense of direction.
- **Recursive best first search** is a linear space best-first search algorithm, will expand fewer nodes than iterative deepening with a non-decreasing cost function. This is like Hill Climbing with backtracking.
- Backtrack if *no child* is the best node on OPEN, except RBFS *rolls* back search to a node it has marked as second best. While rolling back it backs up the lowest value for each node on from its children to update the value of the parent.
- **The Monotone Condition**: Says that for a node n that is a successor to a node m on a path to the goal being constructed by the algorithm A^* using the heuristic function $h(x)$,

  $$h(m) - h(n) \leq k(m, n)$$

  The heuristic function underestimates the cost of each edge.

  $$h(m) + g(m) \leq k(m, n) + h(n) + g(m) = h(n) + g(n)$$

  For A^* the interesting consequence of searching with a heuristic function satisfying the monotone property is that every time it picks a node for expansion, it has found an optimal path to that node.
  As a result, there is no necessity of improved cost propagation through nodes in CLOSED.

4.5 Pruning in A*

- **Sequence Alignment Problem**: Finding the similarity of two amino acid sequences.
- Given two sequences compose of the characters C, A, G and T the task of sequence alignment is to list the two alongside with the option of inserting a gap in either sequence. The objective is to maximize the similarity between the resulting two sequences with gaps possibility inserted.
- The similarity can be quantified by associating a cost with misalignment. Typically, two kinds of penalties are involved
  Mismatch: If character X is aligned to a different character Y.
  Indel: associated with inserting a gap.
- **Similarity function**: A similarity function is a kind of *inverse* of the distance function. Using a similarity function transforms the sequence alignment into a *maximization* problem.
- Let X and Y be the first characters of the two strings. There are three possibilities
  1. Align X with Y
  2. Insert blank before X
  3. Insert blank before Y

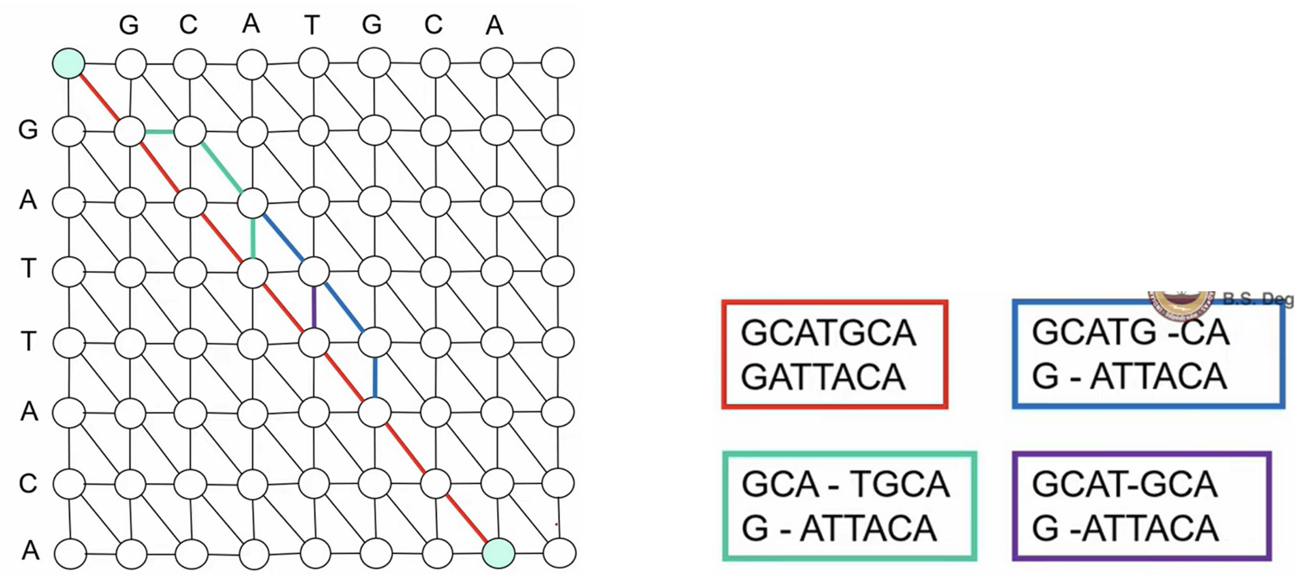


Figure 5: Sequence Alignment Graph

- OPEN grows linearly, whereas CLOSED grows quadratically.
- **Frontier Search**: Only maintain OPEN and throw away CLOSED, nodes in OPEN will be **barred** from generating nodes that would have been in CLOSED. Search only moves forward, without ”leaking back”.It has no means of reconstructing the optimal path it has found.
- **Relay nodes**: at roughly the halfway mark. When a node on OPEN has $g(n) \approx h(n)$ mark it as relay and keep pointers from its descendants on OPEN.
- Now solve two recursive problems, from start to relay and from relay to goal.
- This is known as **Divide and Conquer Frontier Search**(DCFS).
- **Smart Memory Graph Search (SMGS)**: Given that memory is getting cheaper and abundant, one *need not* make recursive calls when A^*(without pruning) could have solved the problem. Keep track of available memory, only when we sense that memory is running out we create a relay layer.
- At all points it identifies a layer of *Boundary nodes* that can be potentially converted into *Relay nodes*. These nodes also stop the search from looking back.
- Find any path from the start to the goal node, could use beam search, but we can prune OPEN by only looking at nodes with $f(n) \leq U$, where U is an upper bound on the cost of the optimal path.

- **Beam Stack search**: Beam Search + Backtracking, Backtracking is **explicit** guided by a Beam Stack that contains pairs of values $[f_{min}, f_{max})$ at each level. The pair of values identify the current nodes in the beam, used to guide regeneration of nodes while backtracking.
- **Breadth First Heuristic Search**: Limit breadth first search to f-values less than U.
- Now we can use the divide and conquer strategy on these algorithms as well.

5 Game Theory

5.1 Introduction

- Game theory is a theoretical framework for conceiving social situations among competing players.
- The focus is the game, which serves as a model of an interactive situation among rational players.
- One player's payoff is contingent on the strategy implemented by the other player.
- It is assumed players within the game are rational and will strive to maximize their payoffs.
- **Nash Equilibrium**: Outcome reached that, once achieved, no player can increase payoff by changing decisions unilaterally.
- Game theory is the study of the ways in which *interacting choices* of *economic agents* produce *outcomes* with respect to the preference of those agents, where the outcomes in question might have been intended by *none* of the agents.
- **The Prisoner's Dilemma**: Two members of a criminal gang are arrested and imprisoned. Each prisoner is given the opportunity either to betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The offer is:
  - If A and B betray each other, each of them serves two years in prison.
  - If A betrays B but B remains silent, A will be set free and B will serve three years in prison, vice versa.
  - If A and B both remain silent, both of them will serve only one year in prison(on a lesser charge).

A, B payoff	Cooperate	Defect
Cooperate	$-1, -1$	$-3, 0$
Defect	$0, -3$	$-2, -2$

- Games can be classified based on payoff as
  1. **Zero sum games**: Total payoff is zero, some players may gain while others lose, competing for payoff.
  2. **Positive sum games**: Total payoff is positive, most players gain, cooperation.
  3. **Negative sum games**: Total payoff is negative, most players lose.
- Games can have two players, more than two players, or can be team games. Team games are often zero-sum.
- Games can have incomplete information, can be stochastic leading to uncertainty.

5.2 Game Tress

- We will consider board games, which are two person, zero-sum games, where we have complete information of the environment and there is no stochastic parameter for them.
- A game tree is a layered tree, Max and Min choose alternately, seen from the perspective of Max.
- A game is a path in the game tree from the root to a leaf node.
- **Minimax backup rule**: The *leaves* are labeled with the *outcome* of the game. When both players are *rational* the *value of the game* is fixed. The *minimax value* is the *outcome* when both play perfectly.

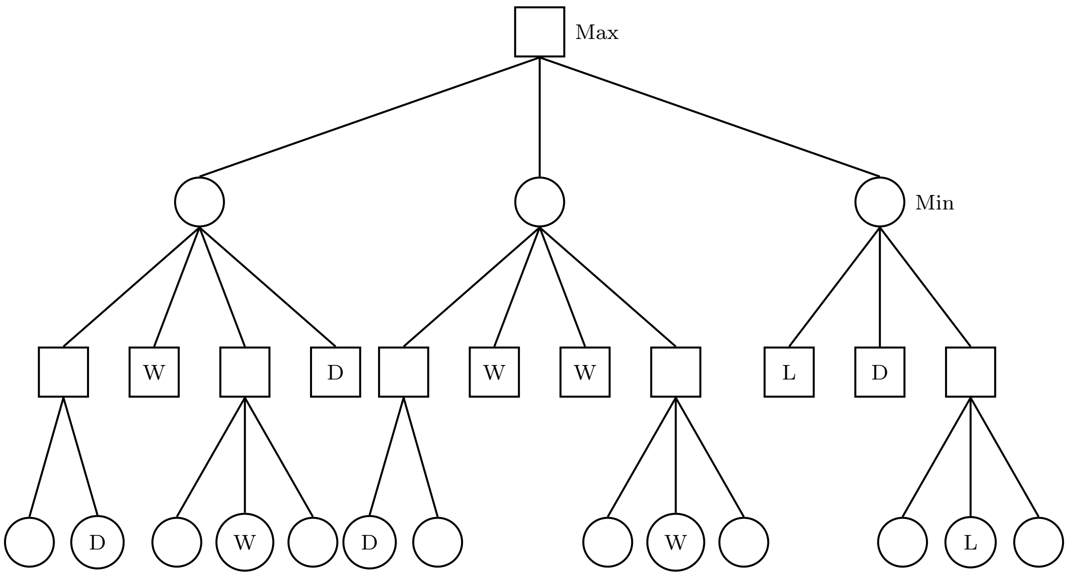


Figure 6: A sample game tree

- A node represents a cluster of partial strategies.
- A strategy for a player is a subtree of the game tree that completely specifies the choices for that player. The algorithm below constructs a strategy for Max

Algorithm 22 Strategy

```
   CONSTRUCT-STRATEGT(MAX)
1: traverse the tree starting at the root
2: if level is MAX then
3:     choose one branch below it
4: else if level is MIN then
5:     choose all branches below it
6: end if
7: return the subtree constructed
```

- **Complexity of Games** = Size of Game Tree
- Most game trees are too big to be analyzed completely.
- Finite Lookahead, limit the depth of the tree to certain number. *k-ply* search.

Algorithm 23 Game Play

```
   GAME-PLAY(MAX)
1: while game not over do
2:     K-PLY-SEARCH
3:     make move
4:     get MIN's move
5: end while
```

- **Evaluation function h**: It's a static function, like a heuristic function, that looks at a board position and returns a value in some range $[-Large, +Large]$.
- For Tic-Tac-Toe, it can be $eval(N)$ = number of free rows, columns, diagonals for MAX − number of free rows, columns, diagonals for MIN.
- We rely on a combination of evaluation and lookahead.

5.3 Algorithm Minimax

- Consider the following pseudocode

Algorithm 24 Minimax Algorithm

```
MINIMAX(N)
if N is a terminal node then
    return EVAL(N)
end if
if N is a MAX node then
    value ← $-\infty$
    for each child C of N do
        value ← max (value,MINIMAX(C))
    end for
else
    value ← $\infty$
    for each child C of N do
        value ← min (value,MINIMAX(C))
    end for
end if
return value
```

- When Max has found a winning move it does not need to explore any further. This notion can even be extended to moves that are not winning moves.
- We call Max nodes as Alpha nodes which store alpha values.
- We call Min nodes as Beta nodes which store beta values.
- Alpha value is the value found so far for the alpha node, and it is a lower bound on the value of the node. It can only be revised upwards, it will reject any lower values subsequently.
- Beta value is the value found so far for the beta node, and it is an upper bound on the value of the node. It can only be revised downwards, it will reject any higher values subsequently.

Algorithm 25 Alpha Beta Pruning

```
ALPHA-BETA($N, \alpha, \beta$)
if $N$ is a terminal node then
    return EVAL($N$)
end if
if $N$ is a MAX node then
    for each child $C$ of $N$ do
        $\alpha \leftarrow \max(\alpha,$ ALPHA-BETA$(C, \alpha, \beta))$
        if $\alpha \geq \beta$ then return $\beta$
        end if
    end for
    return $\alpha$
else
    for each child $C$ of $N$ do
        $\beta \leftarrow \min(\beta,$ ALPHA-BETA$(C, \alpha, \beta))$
        if $\alpha \geq \beta$ then return $\alpha$
        end if
    end for
    return $\beta$
end if
```

5.4 Algorithm SSS*

- A best first search approach to searching the game tree.
- Refine the best looking partial solution till the best solution is fully refined.
- A solution in a game tree is a *strategy.* A partial solution is a partial strategy and stands for a cluster of strategies, these will be intermediate nodes.

- Searches in the solution space.
- First defines clusters to cover all strategies, necessary for completeness.
- The procedure for constructing the initial clusters is as follows

Algorithm 26 Initial Clusters

```
1: Start at the root
2: repeat
3:     At the max level choose all children
4:     At the min level choose one child(left most for simplicity)
5: until the horizon is reached
```

- This covers all strategies, we are essentially finding an upper bound for the strategy.

Algorithm 27 SSS* Iterative Algorithm

```
    SSS*(root)
1:  OPEN ← empty priority queue
2:  add $(root, LIVE, \infty)$ to OPEN
3:  while True do
4:      $(N, status, h)$ ← pop top element from OPEN
5:      if $N$ = root and status is SOLVED then
6:          return h
7:      end if

8:      if status is LIVE then
9:          if N is a terminal node then
10:             add $(N, SOLVED, min(h, eval(N)))$ to OPEN
11:         else if N is a MAX node then
12:             for each child C of N do
13:                 add $(C, LIVE, h)$ to OPEN
14:             end for
15:         else if N is a MIN node then
16:             add (first child of N, $LIVE, h$) to OPEN
17:         end if
18:     end if

19:     if status is SOLVED then
20:         P← parent(N)
21:         if N is a MAX node and N is the last child then
22:             add $(P, SOLVED, h)$ to OPEN
23:         else if N is a MAX node then
24:             add (next child of P, $LIVE, h$) to OPEN
25:         else if N is a MIN node then
26:             add $(P, SOLVED, h)$ to OPEN
27:             remove all successors of P from OPEN
28:         end if
29:     end if
30: end while
```

- We can add another parameter in the tuple that identifies whether the node is MAX or MIN.
- This is very confusing, solve an example by following the algorithm to understand it more clearly.

6 Planning Methods

6.1 Introduction

- The planning community takes an action centric view of problem-solving.
- Solutions are expressed as sequences of actions.
- An autonomous agent in some domain may have certain goals to achieve, may have access to a repository of actions or operators, may use search or other methods to find a plan, executes the plan and monitor it as well.
- Goal driven behavior by an agent: **perceive** or sense its environment, **deliberate** (find a plan to achieve goals) and **act** (execute the planned actions).
- Domains can be modeled with various degrees of expressivity.
- In the simplest domains the agent can perceive the world perfectly.
- In realistic situations the agent may have only partial information.
- The goals or objectives of an agent can be of different kinds
  1. satisfaction goals on end state, these have to be achieved
  2. soft constraints on end state, all goals may not be achieved
  3. hard trajectory constraints, not just the final state on the path too.
  4. soft trajectory constraints
- Actions are also of different kinds
  1. deterministic, always achieve the intended results
  2. stochastic, may or may not achieve the intended results
  3. Instantaneous, no notion of time, once action is chosen result is immediate
  4. durative, have starting and ending time, once action is chosen, agent may have to wait
  5. actions may have an associated cost.
- A planning may be the only one making changes in the world.
- There may be extraneous events, for example rain, shops opening at certain times.
- There may be collaborating agents, or adversarial agents, or agents that may be competing for resources.
- The simplest domain are called STRIPS domain
  1. STandford Research Institute Planning System
  2. Finite, static, completely observable environment.
  3. The set of states and actions is finite.
  4. Changes occur only in response to actions of agents.
  5. The agent has complete information
  6. There are no other agents
  7. The goals are hard constraints on the final state, they have to be achieved.
  8. Actions are instantaneous, there is no explicit notion of time
  9. Actions are deterministic, no accidents, no execution errors or stochastic effects.
- The simplest domains can be modelled as a state-transition system which is defined as a triple (S, A, γ), where S is a finite set of states
  A is a finite set of actions that the actor may perform.
  $\gamma : S \times A \to S$ is a partial function called the state transition function.
  If action a is applicable in state s then $\gamma(s, a)$ is the resulting state.
- Planning Domain Description Languages(PDDL)
  1. The idea is that the researchers working on planning algorithms can use these standardized representations

2. Objects: Things in the world that interest us
3. Predicates: Properties of objects that we are interested in
4. Initial state: The state of the world that we start in
5. Goal specification: Things that we want to be true
6. Actions/Operators: Way of changing the state of the world

- For STRIPS domain we only need PDDL 1.0, further versions add different effects.

6.2 The Block Worlds Domain

- The world is described by a set of statements that conform to the following predicate schema
  1. $on(X, Y)$: block X is on block Y
  2. $ontable(X)$: block X is on the table
  3. $clear(X)$: no block is on block X
  4. $holding(X)$: the robot arm is holding X
  5. *armempty*: the robot arm is not holding anything
- There are no metrics involved, essentially a qualitative description.
- A block can have only one block on it.
- We assume a arbitrarily large table.
- The one-armed robot can hold only one block at a time.
- A planning operator O is defined by
  1. a name (arguments - object types)
  2. a set of preconditions: $pre(O)$
  3. a set of positive effects: $effects^{+}(O)$, ADD list
  4. a set of negative effects: $effects^{-}(O)$, DELETE list
- An action is an instance of an operator with individual objects as arguments, also called a ground operator.

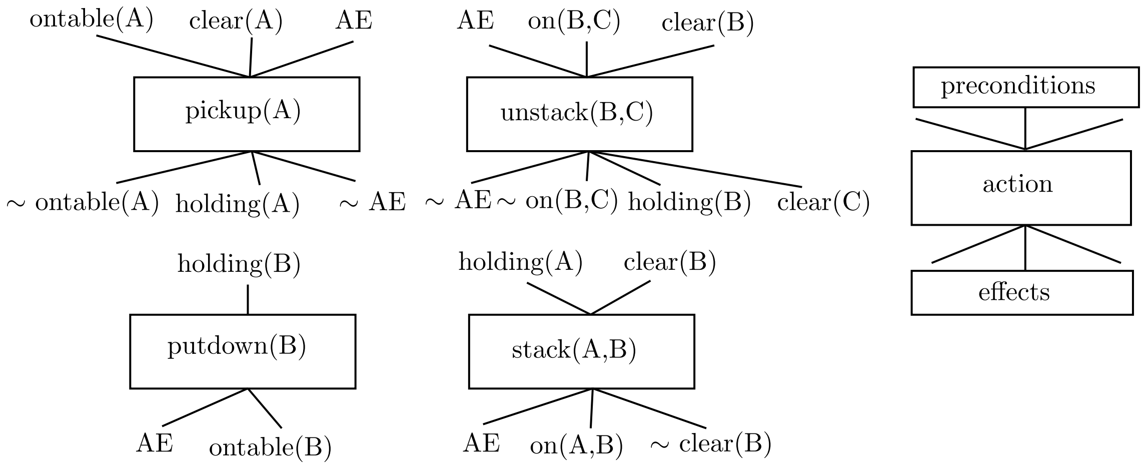


Figure 7: Block Worlds Domain

6.3 Forward State Space Planning

- **Applicable actions**: Given a state S and action a is applicable in the state S if its preconditions are satisfied in the state. That is,
$$pre(a) \subseteq S$$
- **Progression**: If an applicable action a is applied in a state S then the state transitions or progresses to a new state S', defined as,
$$S' = \gamma(S, a) = \{S \cup effects^+(a)\} \backslash effects^-(a)$$
- **Plan**: A plan π is a sequence of actions $\langle a_1, a_2, ..., a_n \rangle$. A plan π is applicable in a state S_0 if there are states $S_1, ..., S_n$ such that $\gamma(S_{i-1}, a_i) = S_i$ for $i = 1, ..., n$.
The final state is $S_n = \gamma(S_0, \pi)$
- **Valid Plan**: Let G be a goal description. Then a plan is a valid plan in a state S_0 if
$$G \subseteq \gamma(S_0, \pi)$$
- FSSP has a high branching factor. This is because the given state is completely described, and many actions are applicable.

6.4 Backward State Space Planning

- **Relevant actions**: Given a goal G an action a is relevant to the goal if it produces some positive effect in the goal, and deletes none. That is,
$$\{effect^+(a) \cap G\} \neq \phi \wedge \{effects^-(a) \cap G\} = \phi$$
- **Regression**: If a relevant action a is applied in a goal G then the goal regresses to a new goal G', also called a subgoal, define as
$$G' = \gamma^{-1}(G, a) = \{G \backslash effects^+(a)\} \cup pre(a)$$
- **Plan**: A plan π is a sequence of actions $\langle a_1, a_2, ..., a_n \rangle$. A plan π is relevant to a goal G_n if there are goals $G_0, ..., G_{n-1}$ such that $G_{i-1} = \gamma^{-1}(G_i, a_i)$ for $i = 1, ..., n$
The final goal is $G_0 = \gamma^{-1}(G_n, \pi)$
- **Valid Plan**: Let S_0 be the start state. Regression ends when $G_0 \subseteq S_0$. The plan π is a valid plan if $G_n \subseteq \gamma(S_0, \pi)$. Validity is still checked by progression.
- BSSP has a lower branching factor. This is because the goal description is often quite small.

	FSSP	BSSP
Start at	Start State	Goal Description
Moves	a is applicable $pre(a) \subseteq S$	a is relevant $\{effect^+(a) \cap G\} \neq \phi \wedge \{effects^-(a) \cap G\} = \phi$
Transition	Progression $S' = \{S \cup effects^+(a)\} \backslash effects^-(a)$ Sound $\pi \leftarrow \pi \circ a$	Regression $G' = \{G \backslash effects^+(a)\} \cup pre(a)$ Not Sound $\pi \leftarrow a \circ \pi$
Goal Test	$G \subseteq \gamma(S_0, \pi)$	$\gamma^{-1}(G_n, \pi) \subseteq S_0$ followed by validity check

Table 3: FSSP vs BSSP

6.5 Goal Stack Planning

- Strives to combine features of FSSP and BSSP
- The basic idea is to break up a compound goal into individual goals and solve them serially one by one, producing a plan that is a sequence of actions, a linear plan.
- It is best suited to domains where goals can be solved serially one at a time.
- Employs a stack in which goals are pushed in a backward manner with actions that could achieve them.
- A basic operation in GSP is $PushSet(G)$
  1. First push a compound goal $G = \{g_1, g_2, ..., g_n\}$ or $\{g_1 \wedge g_2 \wedge ... \wedge g_n\}$
  2. and then the individual goals $g_1, ..., g_n$ in some order
  3. to be solved in last in first out order
  4. possibility of using a heuristic function for ordering the goals or some form of reasoning.
- The reason to insert the compound goal too is to check whether after having solved the individual goals independently the compound goal has indeed been solved.
- The algorithm maintains the current state S at all times.
- When a goal g is popped from the stack, there are two possibilities.
  1. $g \in S$, when the goal is true in the current state nothing needs to be done.
  2. $g \notin S$ when the goal is not true, push relevant action a onto the stack followed by $PushSet(pre(a))$ and the individual goals in $pre(a)$ in some order.
- If an action a is popped then it must be applicable and can be added to the plan.

Algorithm 28 Goal Stack Planning

```
GSP($givenState, givenGoal, actions$)
1: $S \leftarrow givenState$; $plan \leftarrow ()$; $stack \leftarrow emptyStack$
2: PushSet($givenGoal, stack$)
3: while not Empty($stack$) do
4:     $x \leftarrow$ Pop($stack$)
5:     if $x$ is an action $a$ then
6:         $plan \leftarrow (plan \circ a)$
7:         $S \leftarrow$ Progress($S, a$)
8:     else if $x$ is a compound goal $G$ and $G$ is not true then
9:         PushSet($G, stack$)
10:    else if $x$ is a goal $g$ and $g \notin S$ then
11:        CHOOSE a relevant action $a$ that achieves $g$
12:        if $a$ is none then
13:            return FAILURE
14:        end if
15:        Push($a, stack$)
16:        PushSet($pre(a), stack$)
17:    end if
18: end while
19: return $plan$
```

- Goal ordering matters, it is possible to end up in a dead end or get a longer path.
- Certain problems have non-serializable subgoals, there is no optimal order for solving them.
- We may even have to undo previous subgoals to achieve new ones.
- **Sussman's Anomaly**: Neither order of goals will give an optimal plan.
- Sussman showed that there are many problems that have non-serializable subgoals.

6.6 Plan Space Planning

- Consider the space of all possible plans, and search in this space for a plan.
- **Partial plans**: Search space consists of partial plans.
- A partial plan is a 4 tuple, $\pi = \langle A, O, L, B \rangle$
  1. A is a set of partially instantiated operators or actions in the plan
     The set simply identifies the actions that are somewhere in the plan. The actions may be partially instantiated, for example, $Stack(A, ?X)$ - stack block A onto some block.
  2. O is a set of ordering links or relations of the form $(A_i \prec A_j)$
     The partial plan is thus a directed graph, it imposes an order on some actions in the plan.
  3. L is a set of causal links of the form (A_i, P, A_j)
     The causal link represents fact that action A_i has a positive effect P which is a precondition for A_j. A_i is the producer of P, and A_j is the consumer of P.
  4. B is a set of binding constraints of the form
     $$(?X = ?Y), (?X \neq ?Y), (?X = A), (?X \neq B), (?X \neq D_X)$$
- Given a planning problem $\langle S = \{s_1, s_2, ..., s_n\}, G = \{g_1, g_2, ..., g_k\}, O \rangle$
- Planning in plan space always begins with an initial plan $\pi_0 = \langle \{A_0, A_\infty\}, \{(A_0 \prec A_\infty)\}, \{\}, \{\} \rangle$
- A_0 is the initial action with no preconditions and whose positive effects are $s_1, s_2, ..., s_n$
- A_∞ is the final action with preconditions $g_1, g_2, ..., g_k$ and no effects.
- A partial plan may have two kinds of flaws
  1. **Open goals**: any precondition of any action in the partial plan that is not supported by a causal link
  2. **Threats**: A causal link (A_i, P, A_j) is said to have a threat if there exists another action A_t in the plan that potentially deletes P before it can be consumed.
- Plan space planning involves systematically removing flaws.
- A solution plan is a partial plan without any flaws.
- A causal link for P can be found in two ways
  1. If an existing action A_e produces P and it is consistent to add $(A_e \prec A_p)$, then add a causal link (A_e, P, A_p) and add the ordering link $(A_e \prec A_p)$ to the partial plan.
  2. If no such action can be found then insert a new action A_{new} that produces P to the partial plan. Add the corresponding causal link (A_{new}, P, A_p) and the ordering link $(A_{new} \prec A_p)$.
- **Threats Materializing**: Disruption will happen if all three of the following happen
  1. A_{threat} has an effect $\sim Q$ such that P can be unified with Q.
  2. A_{threat} happens after A_i
  3. A_{threat} happens before A_j
- We can resolve a threat by removing one of the conditions
  1. **Separation**: Ensure that P and Q cannot unify. This can be done by adding an appropriate binding constraint to the set B in the partial plan.
  2. **Promotion**: Advance the action A_{threat} to happen before it can disrupt the causal link. Add an ordering $(A_{threat} \prec A_i)$ to the set O in the partial plan.
  3. **Demotion**: Delay the action A_{threat} to happen after both the causal link actions. Add an ordering link $(A_j \prec A_{threat})$ to the set O.

6.7 Multi Armed Robots

- Now we want to modify the STRIPS operators to work with two arms, Arm1 and Arm2.
- Now, instead of $holding(X)$ and, AE we would add $holding1(X)$, $holding2(X)$ and $A1E, AE2$.
- Similarly, we will have different actions for each arm.
- Clearly adding multiple actions would be infeasible, instead we have $pickup(N, A)$, and $holding(N, X)$, where N is the arm number.
- In $unstack(A, B)$ we did not delete $clear(A)$, this is not a problem in a single arm case, but in multi-arm case we will delete it and add it back in $stack(A, B)$.

6.8 Means Ends Analysis

- In their seminal work on *Human Problem Solving*, Newell and Simon proposed a general purpose strategy for problem solving, which they call the **General Purpose Solver (GPS)**. GPS encapsulated the heuristic approach, which they called **Means Ends Analysis**.
- Compare the current state with the desired state, and list the differences between them.
- Evaluate the differences in terms of magnitude in some way.
- Consult an operator-difference table
- Reduce the largest or most important differences first.
- The differences characterize the ends that need to be achieved, the operators define the means of achieving those ends.

6.9 Algorithm Graphplan

- It first constructs a structure called a *planning graph* that potentially captures all possible solutions.
- Then proceeds to search for a solution in the planning graph.
- Starting with Graphplan, a variety of new planning algorithms burst forth
- These algorithms increased the length of plans that could be constructed by an order of magnitude, from tens of actions to hundreds of actions.
- We can also try to convert the planning problem to a satisfiability or constraint satisfaction problem, then use a SAT solver or CSP solver respectively.
- Another direction of research was Heuristic Search planners, domain independent heuristics to guide state space search.
- State space search generates a successor candidate state, which will be a starting point for further exploration.
- The planning graph is a structure which merges the states produced by the different applicable actions.
- The resulting set of propositions forms a layer, as does the set of actions that resulted in this layer.
- The planning graph is a layered graph made up of alternating sets of action layers and proposition layer.
- **Action layer**: Set if all individually applicable actions.
- **Proposition layer** is a union of all states possible.
- We also have a special action no-op, which is always applicable and does nothing. It has one precondition and one positive effect. Thus, every proposition is replicated in every proposition layer.
- We will assume the no-op actions to be implicit in figures.
- Actions will be connected with previous layer with pre-condition links, and will connect to next layer with $effect^+$ and $effect^-$ links.
- The negative effects of actions still persist in the propositional layer because of no-op operators.
- Because of no-op, the actions applicable in previous layer will now be applicable in current layer as well, of course there will be additional actions as well.
- As a result, the action and the proposition layers grow monotonically.
- **Mutual Exclusion (mutex) links**: Two actions $a \in A_i$ and $b \in A_i$ are mutex is one of the following holds
  1. **Competing needs**: There exists $p_a \in pre(a)$ and $p_b \in pre(b)$ such that p_a and p_b are mutex in the preceding layer.
  2. **Inconsistent effects**: There exists a proposition p such that $p \in effects^+(a)$ and $p \in effects^-(b)$ or vice versa. The semantics of these actions in parallel are not defined. And if they are linearized, then the outcome will depend upon the order.
  3. **Interference**: There exists a proposition p such that $p \in pre(a)$ and $p \in effects^-(b)$ or vice versa. Then only one linear ordering of the two actions would be feasible.

4. There exists a proposition p such that $p \in pre(a)$ and $p \in effects^{-}(a)$ and also $p \in pre(b)$ and $p \in effects^{-}(b)$. That is, the proposition is consumed by each action, and hence only one of them can be executed.

- Two propositions $p \in P_i$ and $q \in P_i$ are mutex if all combinations of actions $a, b \in A_i$ with $p \in effects^{+}(a)$ and $q \in effects^{+}(b)$ are mutex.
- The planning graph is made up of the following sets associated with each index i
  1. The set of actions A_i in the i^{th} layer.
  2. The set of propositions P_i in the i^{th} layer.
  3. The set of positive effect links $PostP_i$ of actions in the i^{th} layer.
  4. The set of negative effect links $PostN_i$ of actions in the i^{th} layer.
  5. The set of preconditions links $PreP_i$ of actions in the i^{th} layer from P_{i-1}.
  6. The set of action mutexes in the i^{th} layer.
  7. The set of proposition mutexes in the i^{th} layer.
- If two propositions are non-mutex in a layer, they will be non-mutex in all subsequent layers because of the no-op actions. Likewise for two actions that are non-mutex.
- In P_0 all propositions are non-mutex, since P_0 depicts the start state.
- The process of extending the graph continues till any one of the following two conditions is achieved.
  1. The newest proposition layer contains all the goal propositions, and there is no mutex relation between any of the goal propositions.
  2. The planning graph has leveled off. This means that for two consecutive levels,

     $$P_{i-1} = P \text{ and } MuP_{i-1} = MuP_i$$

     If two consecutive levels have the same set of propositions with the same set of mutex relations between them, it means that no new actions can make an appearance. Hence, if the goal propositions are not present in a levelled planning graph, they can never appear, and the problem has no solution.
- When a planning graph with all goal propositions non-mutex is found, then it is possible, but not necessary, that a valid plan might exist in the graph.
- The algorithm regresses to a set of sub-goals that is non-mutex, continue this process in a depth first fashion.
- If it cannot find a sub-goal set at some level searching backwards, backtrack and try another sub-goal set.
- If it reaches the layer P_0 at some point, then it returns the subgraph that is the shortest makespan plan.
- else it extends the planning graph by one more level, this happens till the planning graph by one more level.
- Graphplan does DFS on the planning graph.

7 Goal Based Reasoning

7.1 Problem Decomposition

- Problem decomposition takes a goal directed view of problem solving.
- The emphasis is on breaking up a problem into smaller problems, like in backward state space planning.
- Primitive problems are labeled SOLVED otherwise they are LIVE and have to be refined, like in the SSS^* game playing algorithm.
- The search space generated for solving And-Or (AO) problems can be seen as a goal tree.
- The nodes in the search space can have a heuristic value that is an estimate of the cost of solving the node.
- Edge costs indicate the cost of transforming a problem.
- An AO graph with mixed nodes can be converted into a graph with pure AND and OR nodes at each level.

7.2 Algorithm AO^*

- At any point, the algorithm AO^* maintains the graph generated so far.
- Every choice point in the graph has a marker, marking the best choice at that point of time.
- The algorithm follows the marked path leading to a set of live nodes. It refines one of the LIVE nodes by expanding it, called the **forward phase**.
- It backs up the cost of the best hyper arc and propagates it upwards, called the **backward phase**.
- If the best choice leads to a SOLVED node, then the parent node is also labeled SOLVED.
- The algorithm terminates when the root is labelled SOLVED.
- The algorithm has the following cycle
  1. Starting at the root, traverse the graph along marked paths till the algorithm reaches a set of unsolved nodes U.
  2. Pick a node n from U and refine it.
  3. Propagate the revised estimate of n up via all ancestors.
  4. If for a node all AND successors along the marked path are marked SOLVED, then mark it SOLVED as well.
  5. If a node has OR edges emanating from it, and the cheapest successor is marked SOLVED, then mark the node SOLVED.
  6. Terminate when the root node is marked SOLVED.
- Like, A^* the algorithm AO^* is also admissible when the heuristic function underestimates the actual cost.
- Refine the best looking partial solution till the best solution is fully refined.

Algorithm 29 And-Or Algorithm Forward Phase

```
$AO^*$(start, Futility)
1: add start to G
2: compute $h(start)$
3: $solved(start) \leftarrow$ FALSE
4: while $solved(start) = FALSE$ and $h(start) \leq$ Futility do
5:     ▷ Forward Phase
6:     $U \leftarrow$ trace marked paths in G to a set of unexpanded nodes
7:     $N \leftarrow$ select a node from $U$
8:     $children \leftarrow$ SUCCESSORS$(N)$
9:     if $children$ is empty then
10:        $h(N) \leftarrow$ Futility
11:    else
12:        check for looping in the members of $children$
13:        remove any looping members from $children$
14:        for each $S \in children$ do
15:            add $S$ to G
16:            compute $h(S)$
17:            if $S$ is primitive then
18:                $solved(S) \leftarrow$ TRUE
19:            end if
20:        end for
21:    end if
```

Algorithm 30 And-Or Algorithm Backward Phase

```
    ▷ Propagate Back
    $M \leftarrow \{N\}$
    ▷ set of modified nodes
    while $M$ is not empty do
        $D \leftarrow$ remove deepest node from $M$
        compute best cost of $D$ from its children
        mark best option at $D$ as MARKED
        if all nodes connected through marked arcs are solved then
            $solved(D) \leftarrow$TRUE
        end if
        if $D$ has changed then
            add all parents of $D$ to $M$
        end if
    end while
end while
if $solved(start) = TRUE$ then
    return the marked subgraph from $start$ node
end if
return null
```

7.3 Deduction in Logic

- Given a knowledge base(KB) which is a set of sentences assumed to be true.
- A move is adding a new sentence in KB by inferring from current sentences.
- Is a given(query) sentence goal α necessarily true.
- **The Greek Syllogism**: Given *All men are mortal* and *Socrates is a man* conclude *Socrates is mortal.* In general, this can be written as, Given *All z are y* and *x is a z* then conclude *x is y.*
- Forward chaining in First order logic, $\forall x(Man(x) \supset Mortal(x))$ and $Man(Socrates)$ then $Mortal(Socrates)$.
- **Backward Chaining**: aka Deductive Retrieval, the goal need not be a specific proposition, it can have variables as well. Formulas with variables can match facts.
- **Conjunctive Antecedents**: A goal $(R\ ?x)$ and a rule $(if(and(P?x)(Q?x))(R?x))$. A goal which matches the consequent of a rule reduces to the goals in the antecedents of the rule. To solve R, we solve both P and Q.
- We solve goal trees in a depth first manner.

8 Rule Based Expert System

- We are looking for patterns in the domain.
- A *rule* looks at a part of a state which matches a pattern and modifies it to result in a new state.
- **State**: A set of sentences in some language.
- **Pattern**: A subset of the sentences.
- We are modifying the state in a piecewise fashion.
- A rule associates an action with a pattern in the state.
  1. also called a production.
  2. unifying format for heuristic knowledge, business rules, and actions.
  3. basis for a Turing complete programming language.
- **Declarative Programming**: The user or programmer only state the rules.
- **Rule Based Production Systems** contain three main components

1. **Working memory (WM)**: Represents the current state, contains a set of records or statements, known as **working memory elements(WMEs)**, it is a model of the short term memory (STM) of the problem solver.
2. **Each rule/production**: Represents a move, has a name or an identifier. Has one or more pattern on LHS and one or more action on RHS. A model of the long term memory (LTM) of the problem solver.
3. **Inference Engine (IE)**: Matches patterns in all rules with all WMEs, picks one matching rule to fire/execute the actions in the rule. Repeat this process till some termination criterion.

- The inference works as, Match $\rightarrow$ Resolve $\rightarrow$ Execute.
- The Match algorithm takes the set of rules and the set of working memory elements and generates the **conflict set**.
- Each element in the conflict set is a rule along with identities of matching WMEs.
- The **conflict** to be resolved is which rule to select from the set of matching rules.
- Match is the most time-consuming step of this process.
- Resolve selects one rule along with the matching working memory elements. It encapsulates the strategy for search.
- Execute implements the actions on the RHS of a rule.
- Actions may make changes in the working memory, which is updated.
- OPS5 syntax for working memory
  - The data structure is a structured record.
  - Class name with a set of attributes, Attribute names are marked by ˆ
  - The data is a collection of attribute-value pairs bunched together, Order of attributes is not important. Default values of attribute is NIL.
- The working memory is a collection of working memory elements, the WMEs are indexed by a time stamp that indicates the order in which they were added to the WM.
- OPS5 syntax for rules
  - The LHS of a rule is a set of patterns, that conform to the syntax of the WMEs.
  - The value field in patterns can have variables and Boolean tests.
  - There may even be negative patterns.
  - The RHS of a rule is a set of actions.
  - Make a new WME, Remove an existing WME, Modify = Remove + Make.
  - Other actions like Read, Write, Load, Halt, ...
  - All actions happen can **concurrently**.
- A rule has a match in the WM if
  - Each positive pattern in the rule has a matching WME. Unsigned patterns are positive by default.
  - There is no WME in the WM that matches a negative pattern in the rule.
- A pattern in a rule matches a WME in the WM if
  - The class name of the pattern = class name of the WME.
  - Each attribute condition in the pattern matches the attribute value in the WME.
  - Attributes not mentioned in the pattern but present in the WME are ignored.
  - {} represent conjunction of tests, all must match.
  - <<>> represent disjunction of tests, some must match.
- Given a large set of WMEs, the rule may match many pairs of elements.
- The set of matching rules with data is called the **conflict set(CS)**.
- The Inference Engine selects one rule from the conflict set.
- **Conflict Resolution**: Choosing which rule to execute or fire next.

- **Refactoriness**: A rule instance may fire only once with a set of matching WMEs. This is particularly relevant when the selected rule does not modify the WMEs matching its preconditions, else only this rule would keep firing.
- **Lexical Order**: All the rules that have matching instances, then choose the first one that the user has stated. And if a rule has multiple instances with different data, then choose the instance that matches the earlier data.
  The strategy places the onus of this choice on the user. The user is more like a programmer. This strategy is used in the programming language Prolog.
  Prolog thus deviates from the idea of declarative programming envisaged by pure logic programming, in which the user would only state the relation between input and output.
- **Specificity**: All the rules that have matching instances, then choose the instance of the rule that is most specific. Specificity can be measured in terms of the number of tests that patterns in rules need.
  The intuition is that the more specific the conditions of a rule are, the more appropriate the rule is likely to be in the given situation. This can facilitate default reasoning.
- **Recency**: All the rules that have matching instances. then choose the instance that has the most recent WME. Recency can be implemented by looking at the time stamps of the matching WMEs.
  The intuition is that when a problem solver adds a new element to the working memory, then any rule that matches that WME should get priority. Recency facilitates "a chain of thought" in reasoning.
  The conflict set can be maintained as a priority queue.
- **Means Ends Analysis**: The idea is to partition the set of rules based on the context and focus on one partition at a time. One can think of each partition as solving a specific subgoal or reducing a specific difference.
  The context is set by the first pattern in a rule. All rules in the same pattern have the same first pattern. The MEA strategy applies Recency to the first pattern in each rule, and Specificity for the remaining patterns.
- **Rete Net**: Looks at changes in WM and outputs changes in conflict set.
- **Discrimination Trees**: A popular approach to work with large amount of data is to use the Divide and Conquer approach and route the query or data token via a sequence of tests, a simple example is Binary Search Trees.
- Alpha nodes serve as the WM of the Rule Based System.
- A WME-token is inserted at the root, $< +WME >$ for an ADD action and $< -WME >$ for a DELETE action.
- The first test, usually, looks at the class-name and separates the tokens. Subsequent tests look at the value of some attribute.
- The key is to sequence the tests in such a manner so that the common tests in different patterns are higher up in the network, and shared between patterns.
- Test at current node, Edges mark test value.
- If a WME passes a test, then it moves to an appropriate alpha node at the next level, else it gets stuck on the parent node.
- Beta nodes have at least one parent, but generally have tow or more parents. All parents must satisfy a join condition, the shared variable must have the same value.
- A rule instance can be attached to any beta node.
- When a positive token is dropped in, it travels down, and may trigger some rules whose other conditions have already been met. If it does, then put these rule instances in a bucket, equal recency.
- For every rule instance in the conflict set, the sum of the lengths of the paths defines specificity. For specificity, we maintain a priority queue on the sum of lengths.
- Once a rule instance is selected to fire, it is removed from the conflict set, refactoriness is automatic.
- When a negative token is dropped in, it may go and exorcise some rules matching earlier.
- Rules with negative patterns need special treatment.

9 Constraint Satisfaction Problems

9.1 Introduction

- A Constraint Satisfaction Problem is a triple ¡X, D, C¿, also called a constraint network.
- X is a set of variable names.
- D is a set of domains, one for each variable.
- C is a set of relations on subsets of variables.
- A solution is an assignment of values for all the variables such that all the constraints are satisfied.
- A network $\mathcal{R}$ is said to express a solution relation ρ.
- Every CSP can be represented as a constraint graph.
- A search algorithm may order variables based on the size of their domain, a heuristic.
- and also do propagation, reasoning, alongside.
- **Interpreting Line Drawings**: Four types of edges
  - Arrow indicates material is on the right, $\rightarrow$ or $\leftarrow$.
  - Convex edge $+$, materials are at an angle of $\leq 90\,\text{deg}$, pops out.
  - Concave edge $-$, materials are at an angle of $\geq 180\,\text{deg}$, like hollow.
- 18 types of joints possible

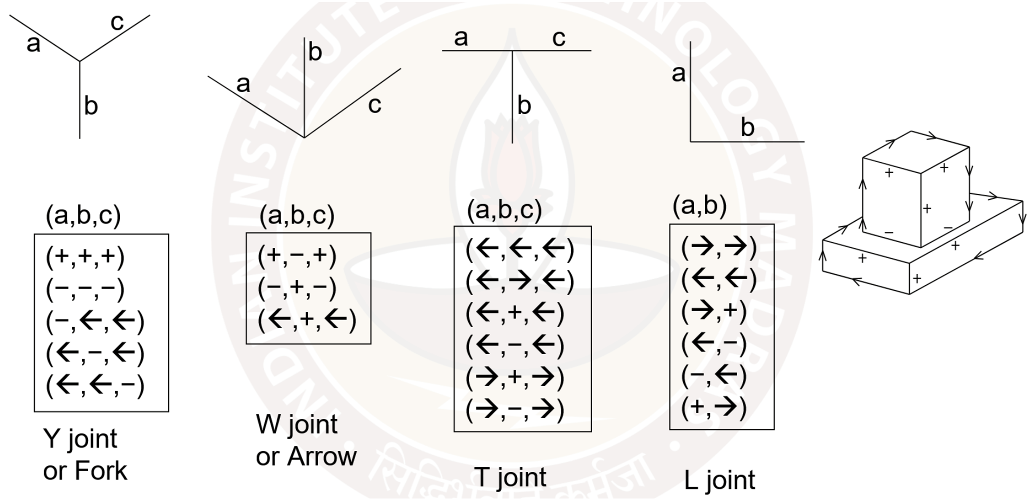


Figure 8: Types of Joints

- Two vertices have the same edge, so one label must be the same.

9.2 Posing and Solving CSPs

- An assignment A_Z assigns values to a subset Z of variables, $Z \subseteq X$
- Let $\mathcal{A} =< a_Z, a_{Z-1}, ..., a_1 >$ be the tuple of values in A.
- Let $\mathcal{A}_S$ be the projection of $\mathcal{A}$ on the set of variables S.
- Then A_Z satisfies a constraint $C = (S, R)$ where S is the scope of R, i.e., if $S \subseteq Z$ and $\mathcal{A}_S \in R$
- An assignment A_Z is consistent, if for every constraint $C = (S, R)$, A_Z satisfies C.
- A solution is a consistent assign for all the variables in X.
- Let $X = (x_1, x_2, ..., x_N)$ be the order in which the N variables are tried.
- Let $D_i = (a_{i1}, a_{i2}, ..., a_{iN})$ be the values in domain D_i in the order they will be tried.

- The search algorithm, Backtracking, is as follows

Algorithm 31 Backtracking

```
BACKTRACKING($X, D, C$)
1:  $\mathcal{A} \leftarrow [\,]$
2:  $i \leftarrow 1$
3:  $D'_i \leftarrow D_i$
4:  while $1 \le i \le N$ do
5:      $a_i \leftarrow$ SELECTVALUE($D'_i, \mathcal{A}, C$)
6:      if $a_i =$ null then
7:          $i \leftarrow i - 1$
8:          $\mathcal{A} \leftarrow$ tail $\mathcal{A}$
9:      else
10:         $\mathcal{A} \leftarrow a_i : \mathcal{A}$
11:         $i \leftarrow i + 1$
12:         if $i \le N$ then
13:             $D'_i \leftarrow D_i$
14:         end if
15:     end if
16: end while
17: return REVERSE($\mathcal{A}$)

SELECTVALUE($D'_i, \mathcal{A}, C$)
18: while $D'_i$ is not empty do
19:     $a_i \leftarrow$ head $D'_i$
20:     $D'_i \leftarrow$ tail $D'_i$
21:     if CONSISTENT($a_i : \mathcal{A}$) then
22:         return $a_i$
23:     end if
24: end while
25: return null
```

- There are various ways to combat combinatorial explosion
    1. Choosing an appropriate ordering of nodes, Min-induced-width ordering of the constraint graph, Select nodes with higher degree first.
    2. Dynamic Variable Ordering, Choose variables with the smallest domains first.
    3. Preprocess the network to prune the search space, Consistency enforcement.
    4. Prune the domains during the search, Lookahead search.
    5. Intelligent Backtracking, Lookback Search, Memoization, i.e., remember combination of nodes that cannot be solved, called nogoods.

- **Arc Consistency**: Let (X, Y) be an edge in the constraint graph of a network. A variable X is said to be arc-consistent w.r.t variable Y iff for every value $a \in D_x$ there is a value $b \in D_Y$ such that $< a, b > \in R_{XY}$ X can be made arc-consistent w.r.t Y by algorithm Revise.

Algorithm 32 Arc Consistency

```
REVISE(($X$), $Y$)
1: for every $a \in D_x$ do
2:     if there is no $b \in D_y$ such that $< a, b > \in R_{XY}$ then
3:         delete $a$ from $D_X$
4:     end if
5: end for
```

- An edge (X, Y) is said to be arc-consistent if both X and Y are arc-consistent w.r.t each other.
- A network is arc-consistent if all its edges are arc-consistent.
- One cycle of calls to Revise is not enough.

- The algorithm $AC1$ cycles through all edges as long as even one domain changes

Algorithm 33 AC1

```
AC-1($X, D, C$)
repeat
    for each edge $(X, Y)$ in the constraint graph do
        Revise($(X), Y$)
        Revise($(Y), X$)
    end for
until no domain changes in the cycle
```

- In the worst case, the network is not arc-consistent and in every cycle exactly one element in one domain is removed.
- For some networks, arc-consistency results in backtrack-free search.
- If the domain of a variable P has changed, then consistency w.r.t P is enforced for the neighbors of P.

Algorithm 34 AC3

```
AC-3($X, D, C$)
$Q \leftarrow [\,]$
for each edge $(N, M)$ in the constraint graph do
    $Q \leftarrow Q ++ (N, M) : [(M, N)]$
end for
while $Q$ is not empty do
    $(P, T) \leftarrow$ head $Q$
    $Q \leftarrow$ tail $Q$
    Revise($(P), T$)
    if $D_P$ has changed then
        for each $R \neq T$ and $(R, P)$ in the constraint graph do
            $Q \leftarrow Q ++ [(R, P)]$
        end for
    end if
end while
```

- A network is said to be i-consistency if an assignment to any $(i-1)$ variables can be consistently extended to i variables.
- 1-consistency is also called node consistency, satisfying that a Boolean variable P to a particular value.
- 2-consistency is called arc consistency.
- 3-consistency is also called path consistency, any edge in the matching diagram can be extended to a triangle.
- If network is strong n-consistent, then it also $n-1$-consistent, $n-2$-consistent,....
- The higher the level of consistency, the lower the amount of backtracking that the search algorithm does.
- Achieving i-consistency before embarking upon search results in a smaller search space being explored.
- Lookahead search variations inspect all values to estimate which one would lead to fewer conflicts in the nature.
- Forward Checking does the least amount of work in looking ahead.
- Partial, Full, Arc Consistency Lookahead do increasingly more work.
- Arc Consistency Lookahead implements full arc consistency between the future variables.
- Backtracking does chronological backtracking.
- Jumpback methods aim to identify culprit variables, the cause of the deadend.
- Culprit can be identified based on graph topology or based on values that cause the conflict.

- Constraint programming offers a unified framework in which search and reasoning can be combined.
- The more the reasoning, the less the search.

Algorithm 35 Forward Checking

```
ForwardChecking($X, D, C$)
1: $\mathcal{A} \leftarrow [\,]$
2: for $k \leftarrow 1$ to $N$ do
3:     $D'_k \leftarrow D_k$
4: end for
5: $i \leftarrow 1$
6: while $1 \leq i \leq N$ do
7:     $a_i \leftarrow$ SelectValue-FC($D'_i, \mathcal{A}, C$)
8:     if $a_i =$ null then
9:         Undo lookahead pruning done while choosing $a_i$
10:        $i \leftarrow i - 1$
11:        $\mathcal{A} \leftarrow$ tail $\mathcal{A}$
12:    else
13:        $\mathcal{A} \leftarrow a_i : \mathcal{A}$
14:        $i \leftarrow i + 1$
15:    end if
16: end while
17: return Reverse($\mathcal{A}$)

SelectValue-FC($D'_i, \mathcal{A}, C$)
18: while $D'_i$ is not empty do
19:     $a_i \leftarrow$ head $D'_i$
20:     $D'_i \leftarrow$ tail $D'_i$
21:     for $k \leftarrow i + 1$ to $N$ do
22:         for each $b$ in $D'_k$ do
23:             if not Consistent($b : a_i : \mathcal{A}$) then
24:                 delete $b$ from $D'_k$
25:             end if
26:         end for
27:     end for
28:     if no $D'_k$ is empty then
29:         return $a_i$
30:     else
31:         for $k \leftarrow i + 1$ to $N$ do
32:             undo deletes in $D'_k$
33:         end for
34:     end if
35: end while
36: return null
```