Notes on lesson 5 on Artificial Intelligence Fundamentls

Vincenzo Gargano

December 14, 2022

Contents

1 Classical research and beyond **5**

1.1 Local Search and Optimization problems 5

1.1.1 Hill Climbing 5

1.1.2 Simulated Annealing 6

1.1.3 Local Beam Search 7

1.1.4 Genetic Algorithms 8

1.1.5 Evolution vs. Search 9

1.1.6 Continuous State Spaces 9

1.1.7 Search with Non-Deterministic Actions 10

1.2 Constraint Satisfaction Problems 10

1.2.1 Constraints varieties 11

1.2.2 Search Algorithms for CSP 11

1.2.3 Local Search for CSP 13

1.2.4 Exploiting problem strucutre 13

1.2.5 Iterative algoithms for CSP 13

1.3 Game Theory and Optimal Decision Making 14

1.3.1 Minimax Search 15

1.3.2 Alpha-Beta Pruning 15

1.3.3 Monte Carlo Tree Search 15

1.3.4 Nondeterministic games 17

1.3.5 Games with imperfect information 17

2 Knowledge-based agents **19**

Chapter 1

Classical research and beyond

1.1 Local Search and Optimization problems

The path usually is irrelevant in Optimization problems. We want to reach the goal state

Start in a random state and then we loop until we reach the goal state, doesnt́ matter how we get there. The good thing is that:

- Very little memory is needed
- Reasonable solution in large or infinite spaces

Example could be TSP or n-queens problem Basic approach on n-queen is counting conflicts and try to reduce the number of conflicts moving a single queen... Not looking ahead, just working locally that is bad computationally wise. or n-Basic approach on n-queen is counting conflicts and try to reduce the number of conflicts moving a single queen... Not looking ahead, just working locally that is bad computationally wise (in a big volume). We can try to improve it using some rules and some knowledge about the problem.

1.1.1 Hill Climbing

"Like climbing Everest in **thick fog** with **amnesia**".

How does works? Keeping track of current state and after each iteration we move the a neighbor state that is better than the current one, with an highest value, so better solution. Otherwise we stick to the current state and search for a better one.

Algorithm 1 Hill Climbing

Input: Initial state s_0 and problem P
local variables: current state s and neighbor state s'
While s' is better than s $s \leftarrow s'$

We can have some points of failure like shoulders, plateaus like in the picture below, we can get stuck. Or similarly we can have flat local maxima.

Or we can have a promising path leading to a **local mimimum**.

We can have **Random Restart Hill Climbing**, that can overcome local maxima, but it can be very slow in big spaces problems. This is trivially complete but expensive. Random sideways moves can help but get stuck in a loop if on a flat maxima.

1.1.2 Simulated Annealing

Idea: **Accept bad moves sometimes, but gradually decrease size of errors and frequency**.
Another local search algorithm, **Annealing** is the process of heating and gradually cooling metals to change their physical properties, make them tempered and more resistent. Allowing them to rech low-energy crysatline structures.

Example of ping-pong ball that has to reach deepest crevice in a bumpy surface. Sometime if we want to move a ball we have to bump the board game to get the ball jump to a particular place (local minimum) that let us proceed toward better states. Shaking but not too hard, in a controlled way.

Algorithm 2 Simulated Annealing(problem, schedule)

Input: Initial problem P
local variables: current state s and next state s', T a Temperature
$s_0 \leftarrow initial\_state$
for $t \leftarrow 1$ to ∞ **do**
  $T \leftarrow schedule(t)$
  if $T = 0$ **then**
    return s
    $s' \leftarrow random\_successor\_of\_s$
    $\Delta E \leftarrow Value(s') - Value(s)$
  end if
  if $\Delta E > 0$ **then**
    This delta is the value of goodness of the state $s \leftarrow s'$
  else
    $s \leftarrow s'$ with $P(Accept) = e^{\Delta E/T}$ probability of accepting worse state
    Decreasing exponentially probability
  end if
end for

In this case we have an hyperparameter T that is the temperature, that is the amount of shaking we want to do. Temperature decreases overtime At a fixed temperature, T, the probability reaches Boltzmann distribution, that is the probability of accepting a worse state is proportional to the difference in value of the states.

$$p(x) = ae^{\frac{E(x)}{kT}} \tag{1.1}$$

Decreasing slowly the T $\Rightarrow$ aliways reach best state $x*$.

Showed in paper Devised by Metropolis et al. in 1953: for problems like VLSI layout, airline scheduling, etc...

1.1.3 Local Beam Search

Until now we had only 1 node at a time in memory for explore the search space. Basic idea is: Keep a set of k states in memory and expand them all at once and then keep the best top % k states. Problem that arise is all k

states end up in the same local hill, an idea could be choose them randomly, with a bias toward the best ones. Analogy with natural selection of genes.

1.1.4 Genetic Algorithms

An example of specialized beam search with a stochastic fashion are Genetic Algorithms. The population of individuals is the states, the best states (fittest) are the one that are likely to be selected for reproduction. We have a recombination process so we can cross the best states to get best results. We can:

- **Crossover**: Combine two parents to get a child
- **Mutation**: Randomly change a gene in a child

Important is consider a fitness function that is a measure of how good a state is. The fitness is an analogous to the $h(x)$ heuristic function.

Example with queens problem: DNA is digit strings representing n-queen states. Fitness is the number of conflicts $\Rightarrow$ seen as a % of how many conflicts are present in the state. When we do crossover we take the genes and we cut the string of position of queen and attach it to another gene.

Cross-over example is $gene1$ = [3,2,7,5,2,4,1,1] and $gene_2$ = [2,4,7,4,8,5,5,2], then we cut first 3 position and we combine obtaining
$gene_{result1}$ = [3,2,7,4,8,5,5,2] and $gene_{result2}$ = [2,4,7,5,2,4,1,1].
We can mutate changing one or more position of the string randomly. $gene_{mutated1}$ = [3,2,7,6,2,4,1,1]

Refer to the book for details on the algorithm. At chapter 4.3.2

Another point is that these algoithm are useful for problems with large spaces and is easy to implement, OpenAI used it for Reinforcement Learning. See blog post: https://blog.openai.com/evolution-strategies/
No need for backpropagation, easier to scale and few hyperparameters. Concept in AI are recurrent and can be used in different ways, mixing them can be useful.

1.1.5 Evolution vs. Search

From the approach of Charles Darwin, we can see that evolution is a process of **survival of the fittest** and **reproduction of the fittest**. But the actual evolution mechanism, biologically speaking is far more complex than most genetic algortihms that we use to "train" agents. In fact genes themselves encode the mechanism that describe how they are reproduced and traslated into an organism. Partner implements the fitness and choose the best fit to reproduce. The worst are discarded. Something that is discussed question like Valiant, Leslie in "Probable Approximately Correct" (suggested book to read), the complexity is too big and learning have to help in some way. Bacteria that is stress in terms of food consumption, create better offspring that can survive in the same conditions.
RNA could be something involved in learning process?

1.1.6 Continuous State Spaces

Suppose we want to site three airport in Romania

- 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
- Objective function $f(x_1, y_2, x_2, y_2, x_3, y_3)$

As the sum of squared distances from each city to nearest airport.

Discretization methods compute: continous state space $\rightarrow$ discrete state space. To a map or a grid. And then we decide where to place the airports, in this way the space is incredibly reduced and we can solve easily.

Gradient methods compute: Remaining in the continous space and define

$$\nabla f = \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \tag{1.2}$$

to increase or reduce f by $x \leftarrow x + \alpha \nabla f$. We know that with the gradient iteratively we are getting better results, using the delta-rule and moving in the opposite direction if we want to minimize the objective function, so downwards in the hill.
Sometime can solve for $\nabla f(x) = 0$ that can be a local minimum or saddle point or anything like that.
Usually we can calculate the Hessian Matrix but at the cost of worst performance. For highly dimensional problems, computing the Hessian is intractable. Local search suffer on plateaus, that is a region of the state space

where the objective function is flat. On ridges, that is a region of the state space where the objective function is flat but the gradient is not zero. So random restarts-like methods are often helpful to solve these problems.

1.1.7 Search with Non-Deterministic Actions

Agents that don't know the environment after transition of state, the environment is stochastic or non-determistic. Instead the agent will make a guess of which state will be in, called "**belief state**".

1.2 Constraint Satisfaction Problems

A CSP is a tuple $< X, D, C >$ where:

- X is a set of variables
- D is a set of domains, one for each variable
- C is a set of constraints

We can define a problem as CSP and then solve it by solving the constraints. In the example in the book the constraint are that the regions must be colored with different colors and the neighbors must have different colors. Binary CPS in which we can represent the problem in a graph and the state with more connection show the one with more constraints (arcs.)

Type of CSP: Discrete variables

- **Finite domains**: D is finite and complexity is $O(n^d)$ and example could be boolean CSP, boolean satisfiability problem that are NP-complete
- **Infinite domains**: The domain is integers, strings etc

Solvable if there are linear, non linear ad undecidable.

Continuous variables

- Start-Ends time for Hubble telescope observation
- Linear constrins solvable in poly time by Linear Programming

1.2.1 Constraints varieties

- **Unary constraints**: C is a set of unary constraints, that is a constraint that involves only one variable eg. SA not green
- **Binary constraints**: C is a set of binary constraints, that is a constraint that involves two variables eg. SA not WA
- **N-ary constraints**: C is a set of N-ary constraints, that is a constraint that involves N variables eg. cryptarithmetic problem
- **Soft constraints**: C is a set of soft constraints, that is a constraint that is not mandatory eg. red is better than green with some costs

Example of Cryptoarithmetic is finding solution to CSP for a problem like "TWO"+"TWO"="FOUR", mapping each letter to a digit and then solve the costraints. For example our constraints can be seen like alldiff(F, T, W, R, U, O) where $O + O = R + 10 * X_1$ and so on.

1.2.2 Search Algorithms for CSP

We can define state as value assigned to each variable
Initial state: empty assignment
Successor function assign value to unassigned variable that do not conflict
Goal test: all variables are assigned and all constraints are satisfied This is the same for all CSPs, every solution appears at depth n with $n\ variable \rightarrow DFS$, the path is irrelevant, so can use complete state formulation and $b = (n-1)d$ at depth n, hence $n!d^n$ leaves!

Backtracking Search

We can obsererve that value assignments are commutative so order does not matter. So we need only to check single variable at each node.
DFS with single variable assignment is called **backtracking search** and is

complete and optimal for finite domains. Informally repeatedly choose unassigned variable and try all possible values in order until we find a value that does not violate any constraints and extend the search via recursive calls, if they fail then we backtrack to previous good state and try next value, if no value can be assigne return failure. Backtracking searching is an uninformed algorithm for CSPs, can solve n-queens for $n = 25$

Pseudocode for backtracking

Algorithm 3 Backtracking Search

How do we solve faster this problems, we can have some question What variable should be the next to be assigne?
What order this value should be tried?
Can we detect failure earlier?
Can we take advantage of the structure of the problem?

Minimum Remaining Values (MRV) heuristic: Choose the variable with the fewest legal values so we prune the search tree.
Degree Euristic: Choose the variable with the most constraints, so the most constraining variable.
Least Constraining Value (LCV) heuristic: Choose the value that rules out the fewest values in the remaining variables.

Every variable are to be assigne, choosing the one that fail first we will have fewer backtracking success, so in this case make sense to look for the most promising solution.

Forward Checking

Keepign track of remaining legal values for legal variable. If we select red for WA, NT being the neighbor can't be red, so we can remove red from the domain of NT. Then we keep doing with another variable for example Q is green so all the neighbors can't be green, so we remove green from the domain of SA, NT, NSW.
Forward checking propagates information from assigne to unassigned variables but doesn't detect early failure. Constrainst propagaion repeatedly enforces constraints locally. **Arc Consistency** is a property that states: $X \to Y$ is consistent iff for every value x in X there is some value y in Y

such that $(x, y) \in C$. In the forward checking the algorithm check initially a sort of arc consistency, but more the algoritm goes on more the arc consistency is lost. Algorithm for Arc Consistency is called **AC-3**. This algorithm is $O(n^2d^3)$ and can be reduced to $O(n^2d^2)$

1.2.3 Local Search for CSP

Local search use complete-state formulation where each state assign a value to each variable, the search changes the value of a single variable at a time. Common strategies are Min-conflicts heuristic: minimum number of conflicts bringing near the solution, usually has a series of plateau. Plateau search allow sideways moves to another state with same score so that can escape plateau, Constraint weighting aim to do the search on the most important constraints, weight of constraint are adjusted by how much time it violate it Min conflicts can have problem for certain ratios: if we reach the critical ration the CPU time will be much larger, some solution can be add more constraint or more variables so that we change the ratio.

1.2.4 Exploiting problem strucutre

Any tree sturcture CSP can be solved in linear time, if the CSP has no loops we can solve it in $O(nd^2)$ time, with worst case $O(d^n)$.

Proof. Choose a variable A as root so that we linearize it, then from the end to the start we RemoveInconsistent(Parent(X_j), X_j) for each X_j in A. And for j from 1 to n, assign X_j consistently with Parent(X_j) □

1.2.5 Iterative algoithms for CSP

Hill-Climbing, Simulated annealing work with "complete state" so we start with all variables assigned and little by little we change values satisfying constraints. The main function that make all the work is the one that reassigns variable values, we can use the heuristic that we saw before like min-conflicts so i.e. Hill climbing with $h(n)$ = total number of constraints.

1.3 Game Theory and Optimal Decision Making

Game are being used for developing AI strategies for a long time, solving game is like a benchmark and keep tracking our improvement.

> *There is a silver lining bias issue. For example, say you have an algorithm trying to predict who should get promotion. And say there was a supermarket chain that, statistically speaking, didn't promote women as often as men. It might be easier to fix an algorithm than fix the minds of 10000 store managers*
> *Richard Socher*

Algorithms that we will see are used for competition at the highest level for example alpha-go playing Go and chess or poker. In these environments usually we have two or more agents with conflicting goal, rising an adversarial search problem.
For simplicity we take

- Two players: Max min, taking turns
- Moves: actions
- Position: State
- Zero sum: a game in which if one opponent is winning the other is losing no win-win situation

Nomeclature: S_0 initial state, TO-MOVE(s) player move in s, ACTIONS(s) legal actions, RESULT(s,a) next state, TERMINAL-TEST(s) is s terminal, UTILITY(s,p) utility of s for p, that is a payoff or objective, we want for us positive utility function. Opponent is unpredictable, we assume rational opponent (want to win), time limits so we must find approximate plan of attack.

Examples of perfect: chess,go,checkers.
Imperfect: poker, backgammon, bridge, mahjong, scrabble

1.3.1 Minimax Search

For deterministic, pefect-information games IDEA: Max player tries to maximize the utility, Min player tries to minimize the utility. If we start with min player we maximize a min value function that looks over childs, if state is terminal return utility otherwise return recursively the min of a max value function keeping the infinium as base the same with the max player.
This search is complete for finite tree, is optimal again optimal opponent, and is exponential in m, the branching factor: $O(b^m)$, space complexity is a depth first exploration so $O(bm)$, for chess is infeasible.

1.3.2 Alpha-Beta Pruning

IDEA: We can prune the search tree by not exploring the branches that we know that are not optimal. For example if we go from 3 to 2 and we have many nodes to explore, we can simply ignore them assuming that the other player will choose the min node so we can skip the search for an entire subtree. Why $\alpha - \beta$ pruning? α is the best value found in current path, if an hypotetic V is worse max will keep α pruning the branch, similarly for β. Pruning doesn't affect final result improve the time complexity to $O(b^m/2)$ with perfect ordering so we can go double the depth, space complexity is the same. Unfortunately still impossible solving chess 35^{50} Reasoning about what computation will be relevant, **metareasoning**, is the important part of these algorithms.

1.3.3 Monte Carlo Tree Search

MCTS doesn't use any euristics, is based on the idea on averaging utility over many simulations, with trial and error like behaviour.

- **Playout**: Simulation chooses until terminal state
- **Selection**: Start in root and with a policy choose a move and repeat
- **Expansion**: Search grows by generating new child node
- **Simulation**: Playout from generated child

- **Backpropagation**: Use the result from simulation to update the search tree nodes until the root

UCT: Effective selection policy is called "upper confidence bounds applied to trees"
This function ranks each possible move based and the formula on upper confidence bounds applied to trees called UCB1.

$$\text{UCB1} = \frac{U(n)}{N(n)} + C\sqrt{\frac{\ln(Parent(N))}{N(n)}} \tag{1.3}$$

Where the first term in green is the exploitation term, averaging the utility of n, if we stop here the node is ranked based on utility accumulated on average, if we rank all the nodes with this term all the relevant utility will be choosen first, and make sense, we want prioritize the better, we are exploiting what we learned, the second term in red in the square root is the exploration term, this tend to explore only the node that present better utilities, maybe some simulation bring to good utilities at the start and we want to be sure that will converge to the solution; the term in the square root is the true exploration term that make that nodes that are being explored less have higher rank, since we have $N(n)$ the count at denominator, the numerator is the log of number of times we explored the parent of n, this make so that the search becomes less useful, in terms of ranking, more we searched in that specific tree.

Algorithm 4 Monte Carlo Tree Search

Input: s_0 initial state, N number of simulations
$tree \leftarrow NODE(State)$
for N **do**
  $leaf \leftarrow SELECT(tree)$
  $child \leftarrow EXPAND(leaf)$
  $result \leftarrow SIMULATE(child)$
  $BACKPROPAGATE(child, result)$
end for
return move in Actions(States) with highest playout count

If the enviroment if undeterministic we need to keep changing our solution over time in the game, some approaches are: Cutoff-Test adding also

quiescence search, that is another way to explore promising nodes, so we can prefer better nodes, for example in chess would be nice to take out a Knight instead of a pawn early in the game. We can also use Eval function instead utility that estimates desirable position.
Suppose in chess we have 100 second and exploring 10^4 nodes/second per move is 10^6 that can reach up to depth 8 in chess program, could be debatable if more depth solution are better than humans player but that many moves grant high performance, if we think how humans approach chess they have like a built-in evaluation function on bad and better moves, more they play more they are good to recognize these better moves, by feeling sometimes...
Some evaluation function for chess can be number of material, how they are placed on the board, for example if someone control the center of checkboard is winning problably, if the bishops have good lines for attacking and defending, if the towers are near each other and so on... eg. $f_1(s) = (numberofwhitequeens) - (numberofblackqueens)$

1.3.4 Nondeterministic games

Example of nondeterministic games is backgammon that is influenced by chances like card shuffling, dice rolls, etc. In this case as utility we can use expected values or weight with probabilities so that we average the utility over the outcomes.
We change minimax in nondeterministic by adding chance nodes like this:
if state is a chance node then return average of ExpectedMiniMax-ValueOfSuccessors(State)
With chance nodes the tree is much bigger than before and $\alpha - \beta$ pruning is not that much of an help. For example TDGammmon uses depth-2 search and very good Eval function and is very competitive at world champion level.
In non deterministic games exact values do matter, behaviour is preserved only by positive linear transfomation of Eval, more on the book...

1.3.5 Games with imperfect information

Games like poker or Scala 40 typically we calculate a probability for each possible deal and the idea is compute minimax value of each action in each deal, then choose the action with highest expected value over all deals. For example we don't know the cards of the opponent, we have to consider all in a belief state and choose no matter the state the action that bring us to the

expected utility that is better. The current best bridge program generate 100 deals consistent with bidding information then pick the action that wins the most deals. From intuition the value of an action is the average of it's values in all state is wrong, an example could be:
Day one take road A will bring us to some gold and road B can bring us to jewels or being run over by a bus, the next day the other way around, with partial observation, values depends on information state or belief state the agent is in, we can generate these tree that will lead the agent to rational behaviours like acting to obtain information, signalling to partner and acting randomly to minimize information disclosure, that are very typical of adversarial games.

Chapter 2

Knowledge-based agents